DEPARTMENT OF MATHEMATICS AND COMPUTING
IIT(ISM) DHANBAD
GPU Computing Lab Manual
V -M.Tech (M&C)
Monsoon Semester 2023-24

Badam Singh Kushvah
E-mail:bskush@iitism.ac.in

Course Type	Course Code	Name of Course	L	T	P	Credit
DP	MCC302	GPU Computing Lab	0	0	2	2

Course Objectives

- To understand the concepts of General-Purpose GPU Programming
- To understand GPU Architecture and Performance
- To learn parallel programming on Single and Multiple GPUs
- To understand programming in OpenCL and pyCUDA

Learning Outcomes

Upon successful completion of this course, students will: Parallel programming skills on the GPU with CUDA

Sl. No.	Name of Experiment/Lab	Learning Outcomes
1.	Basic Programming and CUDA implementation	To write basic CUDA Programs
2.	Programs -Hello world, a Kernel Call and Passing Parameters	To write program to understand host, device and global functions
3.	Vector Sum and Dot Product	To write CUDA program for vector addition and Dot Product
4.	Matrix-Matrix Multiplication	To develop skill for writing Matrix-Matrix Multiplication
5.	Use of shared memory to reduce Global Memory Traffic	To write tiled Matrix-Matrix Multiplication and understand memory management

Sl. No.	Name of Experiment/Lab	Learning Outcomes
6.	Program on warp divergence issue	To understand warp divergence through CUDA program
7.	Implementation of Min/Max/Sum reduction algorithm	To write CUDA porgramme for reduction algorithm and warp divergence
8.	General purpose GPU computing with PyCUDA and PyOpenCL	To understand CUDA pyCUDA, OpenCL programming structure
9.	An overview of OpenCL,Important OpenCL concepts and Basic Program Structure, NumbaPro	To Write basic OpenCL program and use of NumbaPro
10.	GPU Computing Applications- A Case Study in Machine Learning	Case study using GPU programming

Text Books

- Programming Massively Parallel Processors: A Hands-on Approach by David B. Kirk, Wen-mei W. Hwu, Elsevier, 2010[1].

Reference Books

1. Shane Cook: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs Applications of GPU computing series Morgan Kaufmann,Newnes, 2012[2].
2. Dr. Brian Tuomanen:Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA, Packt Publishing Ltd, 2018.

1 General Information and Instructions

Requirements

HARDWARE Linux/Windows System with Graphics Card, SOFTWARE: CUDA, OpenCL, pyCUDA [3],[4]

System Access and Other useful commands

1. Open terminal/putty/mobaxterm
2. Use Server IP (172.16.203.23) with ssh command (**ssh -Y userID@ServerIP**)
3. Write CUDA/OpenCL code using any editor (vi, vim, gedit etc), the extension of file may be .cu
4. At the end of file .bashrc write the followings(only first time):

```
export␣PATH=/usr/local/cuda/bin:$PATH
export␣LD_LIBRARY_PATH=/usr/local/cuda/lib:$LD_LIBRARY_PATH
```

5. Save the .bashrc and use command `source .bashrc`
6. Compile the CUDA code using NVCC ($ nvcc helloWorld.cu) or ($ nvcc helloWorld.cu -o hello)
7. Execute the program using ./a.out or ./hello

8. Use docker images for PyCUDA program as:
   - `␣docker␣run␣--runtime=nvidia␣␣-v␣$HOME:$HOME␣-ti␣bryankp/pycuda:latest␣bash␣`
   - Write PyCUDA using your favorite editor (like vim, Emacs etc.)[5]
   - use command `python3` to execute the code.
9. You may use Colaboratory, or "Colab" for short, is a product from Google Research `https://colab.research.google.c` for pyCUDA and PyOpenCL practicals.

Build Applications with Makefile[6]

Makefile for the following completion of a single file with output

```
$nvcc␣-g␣-G␣-Xcompiler␣-Wall␣main.cpp␣-o␣main.exe
```

is given as

```
#Makefile
NVCC␣=␣/usr/local/cuda/bin/nvcc
NVCC_FLAGS␣=␣-g␣-G␣-Xcompiler␣-Wall
main.exe:␣main.cpp
␣␣␣␣␣␣$(NVCC)␣$(NVCC_FLAGS)␣$<␣-o␣$@
```

For more information please visit `https://www.gnu.org/software/make`

Components of Lab Manual

1. Sample Experiment
2. Aim
3. Program Logic/Steps and CUDA Code
4. Expected Outcomes

Components of Lab Report

1. Cover page (Name of Student, Admission No. with Lab Date,and Submission Date)
2. Title
3. Objectives
4. Experiments
5. Algorithm/ Flowchart
6. CUDA Program with source code file
7. Input(if any)/Output
8. Discussion and Conclusion

2 Experiments

Experiment 2.1 ***Programs -Hello world, a Kernel Call and Passing Parameters***

Objectives: display Hello world on terminal from CPU & GPU threada
CUDA Sample Program

```
#include <stdio.h>
__global__ void helloFromGPU()
{
    printf("Hello World from GPU!\n");
}

int main(int argc, char **argv)
{
    printf("Hello World from CPU!\n");

    helloFromGPU<<<1, 5>>>();
    cudaDeviceReset();
    return 0;
}
```

helloWorld.cu

Output

```
[badam@isrohpc GPULAB]$ nvcc helloWorld.cu
[badam@isrohpc GPULAB]$ ./a.out
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
Hello World from GPU!
```

Lab Exercise 2.1 *Display information from CPU and GPU as per the followings:*

1. *Write a CUDA C program to display your 10-10 times name from CPU and GPU respectively.*
2. *Write a CUDA C program to display your 4 times Course Name, Name of Experiment and Date from CPU and GPU respectively.*

Experiment 2.2 ***Check Device Information***

Objectives: Display information on the first CUDA device, including driver version, runtime version, compute capability, bytes of global memory
CUDA Sample Program

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

```

```
int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    int deviceCount = 0;
    cudaGetDeviceCount(&deviceCount);

    if (deviceCount == 0)
    {
        printf("There are no available device(s) that support CUDA\n");
    }
    else
    {
        printf("Detected %d CUDA Capable device(s)\n", deviceCount);
    }
    int dev = 0, driverVersion = 0, runtimeVersion = 0;
    cudaSetDevice(dev);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    printf("Device %d: \"%s\"\n", dev, deviceProp.name);

    cudaDriverGetVersion(&driverVersion);
    cudaRuntimeGetVersion(&runtimeVersion);

    printf("  CUDA Driver Version / Runtime Version  %d.%d / %d.%d\n", driverVersion / 1000,
     (driverVersion % 100) / 10, runtimeVersion / 1000, (runtimeVersion % 100) / 10);
    printf("  CUDA Capability Major/Minor version number: %d.%d\n", deviceProp.major,
    deviceProp.minor);
    printf("  Total amount of global memory:  %.2f GBytes (%llu " "bytes)\n", (float)
    deviceProp.totalGlobalMem / pow(1024.0, 3),(unsigned long long)deviceProp.totalGlobalMem
    );
    printf("  GPU Clock rate: %.0f MHz (%0.2f " "GHz)\n", deviceProp.clockRate * 1e-3f,
    deviceProp.clockRate * 1e-6f);
    printf("  Memory Clock rate:  %.0f Mhz\n", deviceProp.memoryClockRate * 1e-3f);
    printf("  Memory Bus Width: %d-bit\n", deviceProp.memoryBusWidth);

    if (deviceProp.l2CacheSize)
    {
        printf("  L2 Cache Size: %d bytes\n", deviceProp.l2CacheSize);
    }

    printf("  Max Texture Dimension Size (x,y,z)  1D=(%d), "
            "2D=(%d,%d), 3D=(%d,%d,%d)\n", deviceProp.maxTexture1D,
            deviceProp.maxTexture2D[0], deviceProp.maxTexture2D[1],
            deviceProp.maxTexture3D[0], deviceProp.maxTexture3D[1],
            deviceProp.maxTexture3D[2]);
    printf("  Max Layered Texture Size (dim) x layers  1D=(%d) x %d, "
            "2D=(%d,%d) x %d\n", deviceProp.maxTexture1DLayered[0],
            deviceProp.maxTexture1DLayered[1], deviceProp.maxTexture2DLayered[0],
            deviceProp.maxTexture2DLayered[1], deviceProp.maxTexture2DLayered[2]);
    printf("  Total amount of constant memory:  %lu bytes\n",
            deviceProp.totalConstMem);
    printf("  Total amount of shared memory per block:  %lu bytes\n",deviceProp.
    sharedMemPerBlock);
    printf("  Total number of registers available per block: %d\n", deviceProp.regsPerBlock)
    ;
    exit(EXIT_SUCCESS);
}
```

DeviceInfo.cu

Expected output could be similar to the followings

```
 [badam@isrohpc GPULAB]$ nvcc DeviceInfo.cu
 [badam@isrohpc GPULAB]$ ./a.out
./a.out Starting...
Detected 2 CUDA Capable device(s)
Device 0: "GeForce GTX TITAN X"
  CUDA Driver Version / Runtime Version          10.0 / 9.1
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 11.93 GBytes (12806062080 bytes)
  GPU Clock rate:                                1076 MHz (1.08 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 3145728 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65536), 3D=(4096,4096,4096)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
```

Lab Exercise 2.2 *Write a CUDA program to display the following device information on the terminal:*

1. *Warp size:*
2. *Maximum number of threads per multiprocessor:*
3. *Maximum number of threads per block:*
4. *Maximum sizes of each dimension of a block:*
5. *Maximum sizes of each dimension of a grid:*
6. *Maximum memory pitch:*

The the heading related to above points are: warpSize, maxThreadsPerMultiProcessor, maxThreadsPerBlock, maxThreadsDim[0],deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2], maxGridSize[0], , maxGridSize[1], maxGridSize[2], and memPitch

Expected output could be similar to the followings:

```
  Warp size:  32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:   1024
  Maximum sizes of each dimension of a block:  1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch: 2147483647 bytes
```

Experiment 2.3 *Display the dimensions of grid and a thread block*

Objectives: Display the dimensions number of threads in block and number of block in the grid
CUDA Sample Program

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

__global__ void checkIndex(void)
{
    printf("threadIdx:(%d, %d, %d)\n", threadIdx.x, threadIdx.y, threadIdx.z);
    printf("blockIdx:(%d, %d, %d)\n", blockIdx.x, blockIdx.y, blockIdx.z);

    printf("blockDim:(%d, %d, %d)\n", blockDim.x, blockDim.y, blockDim.z);
    printf("gridDim:(%d, %d, %d)\n", gridDim.x, gridDim.y, gridDim.z);

}

int main(int argc, char **argv)
{
    // define total data element
    int nElem = 3;

    // define grid and block structure
    dim3 block(3);
    dim3 grid((nElem + block.x - 1) / block.x);

    // check grid and block dimension from host side
    printf("grid.x %d grid.y %d grid.z %d\n", grid.x, grid.y, grid.z);
    printf("block.x %d block.y %d block.z %d\n", block.x, block.y, block.z);

    // check grid and block dimension from device side
    checkIndex<<<grid, block>>>();

    // reset device before you leave
    cudaDeviceReset();

    return(0);
}
```

DimensionsGridBlock.cu

Expected output could be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc  DimensionsGridBlock.cu
[badam@isrohpc GPULAB]$ ./a.out
grid.x 1 grid.y 1 grid.z 1
block.x 3 block.y 1 block.z 1
threadIdx:(0, 0, 0)
threadIdx:(1, 0, 0)
threadIdx:(2, 0, 0)
blockIdx:(0, 0, 0)
blockIdx:(0, 0, 0)
blockIdx:(0, 0, 0)
blockDim:(3, 1, 1)
blockDim:(3, 1, 1)
blockDim:(3, 1, 1)
gridDim:(1, 1, 1)
gridDim:(1, 1, 1)
gridDim:(1, 1, 1)
```

Experiment 2.4 ***Define grid and Blocks***

Objectives: Display grid and block structure
CUDA Sample Program

```
#include <cuda_runtime.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    // define total data element
    int nElem = 1024;

    // define grid and block structure
    dim3 block (1024);
    dim3 grid  ((nElem + block.x - 1) / block.x);
    printf("grid.x %d block.x %d \n", grid.x, block.x);

    // reset block
    block.x = 512;
    grid.x  = (nElem + block.x - 1) / block.x;
    printf("grid.x %d block.x %d \n", grid.x, block.x);

    // reset block
    block.x = 256;
    grid.x  = (nElem + block.x - 1) / block.x;
    printf("grid.x %d block.x %d \n", grid.x, block.x);

    // reset block
    block.x = 128;
    grid.x  = (nElem + block.x - 1) / block.x;
    printf("grid.x %d block.x %d \n", grid.x, block.x);

    // reset device before you leave
    cudaDeviceReset();

    return(0);
}
```

GridBlock.cu

Expected output could be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc  GridBlock.cu
[badam@isrohpc GPULAB]$ ./a.out
grid.x 1 block.x 1024
grid.x 2 block.x 512
grid.x 4 block.x 256
grid.x 8 block.x 128
```

Experiment 2.5 ***Vector Addition on GPU***

Objectives: Element wise sum of vector
CUDA Sample Program

```
#include <cuda_runtime.h>
#include <stdio.h>
#define N  10
__global__ void VecAddGPU(int *a,int *b, int *c)
{
  int i=threadIdx.x;
  if(i<N)
  {
    c[i]=a[i]+b[i];
  }
}

int main(int argc, char **argv)
{
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// allocate the memory on device
cudaMalloc((void**)&dev_a, N*sizeof(int));
cudaMalloc((void**)&dev_b, N*sizeof(int));
cudaMalloc((void**)&dev_c, N*sizeof(int));
for(int i=0; i<N;i++){
a[i] = -i;
b[i] = i * i;
}

//Copy data from host to device
//cudaError_t=
cudaError_t err=cudaMemcpy(&dev_a, a, N*sizeof(int),cudaMemcpyHostToDevice);
if(err!=cudaSuccess)
{
printf("%s in %s at line %d\n",cudaGetErrorString(err),__FILE__,__LINE__);
exit(1);
}
cudaMemcpy(dev_b, b, N*sizeof(int),cudaMemcpyHostToDevice);
//kernel launch
VecAddGPU<<<1,N>>>(dev_a,dev_b,dev_c);
//Copy result from  device to host
cudaMemcpy(c,dev_c, N*sizeof(int),cudaMemcpyDeviceToHost);

for (int i=0; i<N;i++)
{
printf("%d+%d=%d\n",a[i],b[i],c[i]);
}
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
return 0;
}
```

VecAddGPU.cu

Expected output could be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc  VecAddGPU.cu
[badam@isrohpc GPULAB]$ ./a.out
0+0=0
-1+1=0
-2+4=2
-3+9=6
-4+16=12
```

```
-5+25=20
-6+36=30
-7+49=42
-8+64=56
-9+81=72
```

Lab Exercise 2.3 *Write a CUDA program to display*

1. *Display grid, block and thread details for a block of size (256,3,1):*

Expected output could be similar to that of in 2.4

Lab Exercise 2.4 *Write a CUDA program to display*

1. *Distance between two vectors* x *and* y *where* $x = \{i^2\}_{i=1}^n$, $y = \{(2i+1)\}_{i=1}^n$ *and* $n = 1024$. *Also find the Euclidean norms of* x *and* y *respectively.*
2. *Find the standard deviation of* $y = \{(2i+1)\}_{i=1}^n$ *and* $n = 1024$.

Experiment 2.6 *Matrix-Matrix sum on GPU*

Objectives: Sum of two matrices
CUDA Sample Program

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

void initialData(float *ip, const int size)
{
    int i;

    for(i = 0; i < size; i++)
    {
        ip[i] = i;
    }

    return;
}
void displayMatrix(float *A, int nx,int ny)
{
int idx;
        for (int i = 0; i < nx; i++)
  {
        for (int j = 0; j < ny; j++)
        {
         idx = i*ny + j;
         printf(" %f ",A[idx]);
        }
            printf("\n");
        }
    return;
```

```
}


// grid 1D block 1D
__global__ void sumMatrixOnGPU(float *MatA, float *MatB, float *MatC, int nx,
                                    int ny)
{
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;

    if (ix < nx )
        for (int iy = 0; iy < ny; iy++)
        {
            int idx = iy * nx + ix;
            MatC[idx] = MatA[idx] + MatB[idx];
        }


}

int main(int argc, char **argv)
{

    // set up data size of matrix
    int nx = 4;
    int ny = 5;

    int nxy = nx * ny;
    int nBytes = nxy * sizeof(float);
    printf("Matrix size: nx %d ny %d\n", nx, ny);

    // malloc host memory
    float *h_A, *h_B,*h_C;
    h_A = (float *)malloc(nBytes);
    h_B = (float *)malloc(nBytes);
    h_C = (float *)malloc(nBytes);

    // initialize data at host side
    initialData(h_A, nxy);
    initialData(h_B, nxy);

    // malloc device global memory
    float *d_MatA, *d_MatB, *d_MatC;
   cudaMalloc((void **)&d_MatA, nBytes);
   cudaMalloc((void **)&d_MatB, nBytes);
   cudaMalloc((void **)&d_MatC, nBytes);

    // transfer data from host to device
    cudaMemcpy(d_MatA, h_A, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_MatB, h_B, nBytes, cudaMemcpyHostToDevice);

    // invoke kernel at host side
    int dimx = 32;
    dim3 block(dimx, 1);
    dim3 grid((nx + block.x - 1) / block.x, 1);

    sumMatrixOnGPU<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny);
    cudaDeviceSynchronize();


    // copy kernel result back to host side
    cudaMemcpy(h_C, d_MatC, nBytes, cudaMemcpyDeviceToHost);
    displayMatrix(h_C, nx, ny);


    // free device global memory
```

```
    cudaFree(d_MatA);
    cudaFree(d_MatB);
    cudaFree(d_MatC);

    // free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    // reset device
    cudaDeviceReset();

    return (0);
}
```

sumMatrix1D.cu

Expected output could be similar to the followings

```
[bsk@gr02 ]$ nvcc sumMatrix1D.cu
[bsk@gr02 ]$ ./a.out
Matrix size: nx 4 ny 5
 0.000000  2.000000  4.000000  6.000000  8.000000
 10.000000  12.000000  14.000000  16.000000  18.000000
 20.000000  22.000000  24.000000  26.000000  28.000000
 30.000000  32.000000  34.000000  36.000000  38.000000
```

Lab Exercise 2.5 *Write a CUDA program to demonstrate the followings*

1. *Allocate Device Memory*
2. *Transfer Data(Matrices A and B) from host to device*
3. *Sum two matrices using 2D grid*
4. *Transfer result(Matrix C) from device to host*
5. *Print the result in matrix format*

Expected output could be similar to that of in 2.6

Lab Exercise 2.6 *Write a CUDA program to demonstrate*

1. *Allocate Device Memory*
2. *Transfer Data(Matrices A and B) from host to device*
3. *Sum two matrices using 2D grid with different block sizes*
4. *Transfer result(Matrix C) from device to host*
5. *Print the result in matrix format*

6. *Show the effect of block size and grid size in terms of total run time.*

Experiment 2.7 *Matrix-Matrix Multiplication on GPU*

Objectives: Multiply of two matrices
CUDA Sample Program

```
#include <cuda_runtime.h>
#include <stdio.h>
#define N  3
__global__ void MatrixMulKernel(float* MatA, float* MatB, float* MatC,
int Width) {
// Calculate the row index of the P element and M
int Row = blockIdx.y*blockDim.y+threadIdx.y;
// Calculate the column index of P and N
int Col = blockIdx.x*blockDim.x+threadIdx.x;
if ((Row < Width) && (Col < Width)) {
float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k) {
Pvalue += MatA[Row*Width+k]*MatB[k*Width+Col];
}
MatC[Row*Width+Col] = Pvalue;
}
}

void initialData(float *ip, const int size)
{
    int i;

    for(i = 0; i < size; i++)
    {
        ip[i] = ((float)rand()/(float)(RAND_MAX));
    }

    return;
}
void displayMatrix(float *A, int nx,int ny)
{
int idx;
        for (int i = 0; i < nx; i++)
  {
        for (int j = 0; j < ny; j++)
        {
         idx = i*ny + j;
         printf(" %f ",A[idx]);
        }
            printf("\n");
        }
    return;
}

int main(int argc, char **argv)
{

    // set up data size of matrix
    int Width=N;
    int nx = Width;
    int ny = Width;

    int nxy = nx * ny;
```

```
    int nBytes = nxy * sizeof(float);
    printf("Matrix size: nx %d ny %d\n", nx, ny);

    // malloc host memory
    float *h_A, *h_B,*h_C;
   h_A = (float *)malloc(nBytes);
   h_B = (float *)malloc(nBytes);
   h_C = (float *)malloc(nBytes);

    // initialize data at host side
    initialData(h_A, nxy);
    initialData(h_B, nxy);

    // malloc device global memory
    float *d_MatA, *d_MatB, *d_MatC;
  cudaMalloc((void **)&d_MatA, nBytes);
  cudaMalloc((void **)&d_MatB, nBytes);
  cudaMalloc((void **)&d_MatC, nBytes);

    // transfer data from host to device
    cudaMemcpy(d_MatA, h_A, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_MatB, h_B, nBytes, cudaMemcpyHostToDevice);

    // invoke kernel at host side
    int bdimx = 16;
    int bdimy=16;
    dim3 block(bdimx, bdimy);
    dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y,1);

    MatrixMulKernel<<<grid, block>>>(d_MatA, d_MatB, d_MatC,Width);
    cudaDeviceSynchronize();


    // copy kernel result back to host side
    cudaMemcpy(h_C, d_MatC, nBytes, cudaMemcpyDeviceToHost);
    printf("Matrix A is=\n");
    displayMatrix(h_A, nx, ny);
     printf("Matrix B is=\n");
    displayMatrix(h_B, nx, ny);
    printf("The product of Matrix A and Matrix B is=\n");
    displayMatrix(h_C, nx, ny);


    // free device global memory
    cudaFree(d_MatA);
    cudaFree(d_MatB);
    cudaFree(d_MatC);

    // free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    // reset device
    cudaDeviceReset();

    return (0);
}
```

matrix–matrix–multGPU.cu

Expected output could be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc matrix-matrix-multGPU.cu
[badam@isrohpc GPULAB]$ ./a.out
Matrix size: nx 3 ny 3
Matrix A is=
 0.000000  1.000000  2.000000
 3.000000  4.000000  5.000000
 6.000000  7.000000  8.000000
Matrix B is=
 0.000000  1.000000  2.000000
 3.000000  4.000000  5.000000
 6.000000  7.000000  8.000000
The product of Matrix A and Matrix B is=
 15.000000  18.000000  21.000000
 42.000000  54.000000  66.000000
 69.000000  90.000000  111.000000
```

Lab Exercise 2.7 *Write a CUDA program to demonstrate the followings*

1. *Allocate Device Memory*
2. *Transfer Data(Matrices A, B and C) from host to device*
3. *Find the Product of three matrices A*B*C using 2D grid*
4. *Transfer result from device to host*
5. *Print the result in matrix format*

Expected output could be similar to that of in 2.7

Lab Exercise 2.8 *Write a CUDA program to demonstrate*

1. *Allocate Device Memory*
2. *Transfer Data(Matrices A and B) from host to device*
3. *Find the transpose (TA and TB) of matrices A and B in parallel on GPU*
4. *Find the product of A and B and TA and TB*
5. *Transfer results from device to host*
6. *Print the result matrices and their differences*
7. *Show the effect of block size and grid size in terms of total run time.*

Experiment 2.8 *Use of Makefile with Main program, Distance Kernel and Header Kernel*

Objectives: Use of Makefile
CUDA Sample Program

```
#include "DistKernel.h"
#include <stdlib.h>
#define N 16
float scale(int i, int n)
{
return ((float)i)/(n - 1);
}
int main()
{
const float ref = 0.5f;

float *in = (float*)calloc(N, sizeof(float));

float *out = (float*)calloc(N, sizeof(float));

// Compute scaled input values

for (int i = 0; i < N; ++i)
{
in[i] = scale(i, N);
}

// Compute distances for the entire array

distanceArray(out, in, ref, N);
free(in);
free(out);
return 0;
}
```

distanceMain.cpp

```
#include "DistKernel.h"
#include <stdio.h>
#define TPB 16
__device__ float distance(float x1, float x2)
{
return sqrt((x2 - x1)*(x2 - x1));
}

__global__ void distanceKernel(float *d_out, float *d_in, float ref)
{
const int i = blockIdx.x*blockDim.x + threadIdx.x;
const float x = d_in[i];
d_out[i] = distance(x, ref);
printf("i = %2d: dist from %f to %f is %f.\n", i, ref, x, d_out[i]);
}
void distanceArray(float *out, float *in, float ref, int len)
{
// Declare pointers to device arrays
float *d_in = 0;
float *d_out = 0;
// Allocate memory for device arrays
cudaMalloc(&d_in, len*sizeof(float));
cudaMalloc(&d_out, len*sizeof(float));
// Copy input data from host to device
cudaMemcpy(d_in, in, len*sizeof(float), cudaMemcpyHostToDevice);
// Launch kernel to compute and store distance values
distanceKernel<<<len/TPB, TPB>>>(d_out, d_in, ref);
// Copy results from device to host
```

```
cudaMemcpy(out, d_out, len*sizeof(float), cudaMemcpyDeviceToHost);
// Free the memory allocated for device arrays
cudaFree(d_in);
cudaFree(d_out);
}
```

DistKernel.cu

```
#ifndef KERNEL_H
#define KERNEL_H
// Kernel wrapper for computing array of distance values
void distanceArray(float *out, float *in, float ref, int len);
#endif
```

DistKernel.h

```
NVCC = /usr/local/cuda/bin/nvcc
NVCC_FLAGS = -g -G -Xcompiler -Wall

all: distanceMain.exe
distanceMain.exe: distanceMain.o DistKernel.o
	$(NVCC) $^ -o $@

distanceMain.o: distanceMain.cpp DistKernel.h
	$(NVCC) $(NVCC_FLAGS) -c $< -o $@

DistKernel.o: DistKernel.cu DistKernel.h
	$(NVCC) $(NVCC_FLAGS) -c $< -o $@
```

Makefile

Use `make` to compile the files: Output will be similar to the following

```
[badam@isrohpc GPULAB]$ make
/usr/local/cuda/bin/nvcc -g -G -Xcompiler -Wall -c distanceMain.cpp -o distanceMain.o
/usr/local/cuda/bin/nvcc  -g -G -Xcompiler -Wall -c DistKernel.cu -o DistKernel.o
/usr/local/cuda/bin/nvcc distanceMain.o DistKernel.o -o distanceMain.exe
```

Run `distanceMain.exe` the output will be similar to the following

```
 [badam@isrohpc GPULAB]$ ./distanceMain.exe
i =  0: dist from 0.500000 to 0.000000 is 0.500000.
i =  1: dist from 0.500000 to 0.066667 is 0.433333.
i =  2: dist from 0.500000 to 0.133333 is 0.366667.
i =  3: dist from 0.500000 to 0.200000 is 0.300000.
i =  4: dist from 0.500000 to 0.266667 is 0.233333.
i =  5: dist from 0.500000 to 0.333333 is 0.166667.
i =  6: dist from 0.500000 to 0.400000 is 0.100000.
i =  7: dist from 0.500000 to 0.466667 is 0.033333.
i =  8: dist from 0.500000 to 0.533333 is 0.033333.
i =  9: dist from 0.500000 to 0.600000 is 0.100000.
i = 10: dist from 0.500000 to 0.666667 is 0.166667.
i = 11: dist from 0.500000 to 0.733333 is 0.233333.
```

```
i = 12: dist from 0.500000 to 0.800000 is 0.300000.
i = 13: dist from 0.500000 to 0.866667 is 0.366667.
i = 14: dist from 0.500000 to 0.933333 is 0.433333.
i = 15: dist from 0.500000 to 1.000000 is 0.500000.
```

Lab Exercise 2.9 *Write a CUDA program to demonstrate the followings*

1. *Write a header file for declaring add and multiply functions*
2. *Write a device functions to add the two Matrices in GPU*
3. *Then find the Square of resultant Matrix using global function*
4. *Transfer result from device to host*
5. *Print the result*

Expected output could be a single matrix

Lab Exercise 2.10 *Write a CUDA program to demonstrate the followings*

1. *Write a header file for declaring functions(device and global)*
2. *Write a device functions to transpose of matrix A in GPU*
3. *Then find the product of A and A^T using global function*
4. *Transfer result from device to host*
5. *Print the result*

Expected output could be a single matrix

Experiment 2.9 *Tiled Matrix-Matrix Multiplication*

Objectives: Use tiled algorithm for Matrix-Matrix multiplication
CUDA Sample Program

```
#include <cuda_runtime.h>
#include <stdio.h>
#include "funcDef.h"
#define N  8
#define TILE_WIDTH 2
__global__ void MatrixMulKernel(float* MatA, float* MatB, float* MatC,
int Width) {
//Shared memory allocation
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
int bx = blockIdx.x; int by = blockIdx.y;
```

```
int tx = threadIdx.x; int ty = threadIdx.y;
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;

float Pvalue = 0;
 for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
// Collaborative loading of A and B  tiles into shared memory
Mds[ty][tx] = MatA[Row*Width + ph*TILE_WIDTH + tx];
Nds[ty][tx] = MatB[(ph*TILE_WIDTH + ty)*Width + Col];
__syncthreads();
//dot product using shared memory
for (int k = 0; k < TILE_WIDTH; ++k) {
 Pvalue += Mds[ty][k] * Nds[k][tx];
}
__syncthreads();
}
MatC[Row*Width+Col] = Pvalue;
}


void displayMatrix(float *A, int nx,int ny)
{
int idx;
        for (int i = 0; i < nx; i++)
  {
        for (int j = 0; j < ny; j++)
        {
         idx = i*ny + j;
         printf(" %f ",A[idx]);
        }
            printf("\n");
        }
    return;
}

int main(int argc, char **argv)
{

    // set up data size of matrix
    int Width=N;
    int nx = Width;
    int ny = Width;

    int nxy = nx * ny;
    int nBytes = nxy * sizeof(float);
    printf("Matrix size:  %d  by  %d\n", nx, ny);
 printf("Tile size:  %d by  %d\n", TILE_WIDTH,  TILE_WIDTH);

    // malloc host memory
    float *h_A, *h_B,*h_C;
    h_A = (float *)malloc(nBytes);
    h_B = (float *)malloc(nBytes);
    h_C = (float *)malloc(nBytes);

    // initialize data at host side
    initialData(h_A, nxy);
    initialData(h_B, nxy);

    // malloc device global memory
    float *d_MatA, *d_MatB, *d_MatC;
   cudaMalloc((void **)&d_MatA, nBytes);
   cudaMalloc((void **)&d_MatB, nBytes);
   cudaMalloc((void **)&d_MatC, nBytes);

    // transfer data from host to device
```

```
    cudaMemcpy(d_MatA, h_A, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_MatB, h_B, nBytes, cudaMemcpyHostToDevice);

    // invoke kernel at host side
    int bdimx = TILE_WIDTH;
    int bdimy=TILE_WIDTH;
    dim3 block(bdimx, bdimy);
    dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y,1);

    MatrixMulKernel<<<grid, block>>>(d_MatA, d_MatB, d_MatC,Width);
    cudaDeviceSynchronize();


    // copy kernel result back to host side
    cudaMemcpy(h_C, d_MatC, nBytes, cudaMemcpyDeviceToHost);
    printf("Matrix A is=\n");
    displayMatrix(h_A, nx, ny);
     printf("Matrix B is=\n");
    displayMatrix(h_B, nx, ny);
    printf("The product of Matrix A and Matrix B is=\n");
    displayMatrix(h_C, nx, ny);


    // free device global memory
    cudaFree(d_MatA);
    cudaFree(d_MatB);
    cudaFree(d_MatC);

    // free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    // reset device
    cudaDeviceReset();

    return (0);
}
```

Tiled–Mat–multGPU.cu

```
#include "funcDef.h"
void initialData(float *ip, const int size)
{
    int i;

    for(i = 0; i < size; i++)
    {
        ip[i] = ((float)rand()/(float)(RAND_MAX));
    }

    return;
}
```

initialDataMatAB.cu

```
 #ifndef FUNCDEF_H
 #define FUNCDEF_H
void initialData(float *ip, const int size);
 #endif
```

funcDef.h

```
NVCC = /usr/local/cuda/bin/nvcc
#NVCC_FLAGS = -g -G -Xcompiler -Wall

#all: distanceMain.exe
Tiled-Mat-multGPU.exe: Tiled-Mat-multGPU.o initialDataMatAB.o funcDef.h
	$(NVCC) Tiled-Mat-multGPU.cu initialDataMatAB.o  -o Tiled-Mat-multGPU.exe
Tiled-Mat-multGPU.o: Tiled-Mat-multGPU.cu funcDef.h initialDataMatAB.o
	$(NVCC) -c Tiled-Mat-multGPU.cu  initialDataMatAB.o
initialDataMatAB.o: funcDef.h initialDataMatAB.cu
	$(NVCC) -c initialDataMatAB.cu
clean:
	rm -f Tiled-Mat-multGPU.exe
```

Makefile2

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ make -f Makefile2
/usr/local/cuda/bin/nvcc -c Tiled-Mat-multGPU.cu  initialDataMatAB.o
/usr/local/cuda/bin/nvcc Tiled-Mat-multGPU.cu initialDataMatAB.o  -o Tiled-Mat-multGPU.exe
[badam@isrohpc GPULAB]$ ./Tiled-Mat-multGPU.exe
Matrix size:   8  by  8
Tile size:   2 by  2
Matrix A is=
 0.840188  0.394383  0.783099  0.798440  0.911647  0.197551  0.335223  0.768230
 0.277775  0.553970  0.477397  0.628871  0.364784  0.513401  0.952230  0.916195
 0.635712  0.717297  0.141603  0.606969  0.016301  0.242887  0.137232  0.804177
 0.156679  0.400944  0.129790  0.108809  0.998924  0.218257  0.512932  0.839112
 0.612640  0.296032  0.637552  0.524287  0.493583  0.972775  0.292517  0.771358
 0.526745  0.769914  0.400229  0.891529  0.283315  0.352458  0.807725  0.919026
 0.069755  0.949327  0.525995  0.086056  0.192214  0.663227  0.890233  0.348893
 0.064171  0.020023  0.457702  0.063096  0.238280  0.970634  0.902208  0.850920
Matrix B is=
 0.266666  0.539760  0.375207  0.760249  0.512535  0.667724  0.531606  0.039280
 0.437638  0.931835  0.930810  0.720952  0.284293  0.738534  0.639979  0.354049
 0.687861  0.165974  0.440105  0.880075  0.829201  0.330337  0.228968  0.893372
 0.350360  0.686670  0.956468  0.588640  0.657304  0.858676  0.439560  0.923970
 0.398437  0.814767  0.684219  0.910972  0.482491  0.215825  0.950252  0.920128
 0.147660  0.881062  0.641081  0.431953  0.619596  0.281059  0.786002  0.307458
 0.447034  0.226107  0.187533  0.276235  0.556444  0.416501  0.169607  0.906804
 0.103171  0.126075  0.495444  0.760475  0.984752  0.935004  0.684445  0.383188
The product of Matrix A and Matrix B is=
 1.836571  2.588725  2.984560  3.674901  3.222222  2.906765  2.833551  3.107897
 1.606581  2.257570  2.642685  2.914744  3.035270  2.768578  2.426692  2.922653
 0.980174  1.811519  2.140080  2.251573  2.072760  2.403044  1.876315  1.488523
 1.090756  1.782398  1.928511  2.370534  2.102371  1.812182  2.199538  2.137506
 1.465810  2.494980  2.685837  3.086129  3.034677  2.511495  2.702769  2.496949
 1.685890  2.520235  2.969749  3.164906  3.116278  3.173994  2.568542  2.928237
 1.434502  2.054889  2.120121  2.223845  2.140972  1.920228  1.896632  2.210005
 1.092188  1.533195  1.680568  2.035834  2.515924  1.758628  1.904237  2.138672
```

Lab Exercise 2.11 *Write a CUDA program to demonstrate the followings*

1. *Write a header file for declaring Matrix-Matrix Multiplication*
2. *Write a device functions to multiply two matrices using tiled algorithm and without tiled algorithm in GPU*
3. *Transfer result from device to host*
4. *Print the result*

Expected output could be two matrices

Experiment 2.10 *Sum reduction*

Objectives: Write a CUDA program for sum reduction
CUDA Sample Program

```
#include <cuda_runtime.h>
#include <stdio.h>
#define N  1000
#define BD 256
#define CHECK(call)                                                     \
{                                                                       \
    const cudaError_t error = call;                                     \
    if (error != cudaSuccess)                                           \
    {                                                                   \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);         \
        fprintf(stderr, "code: %d, reason: %s\n", error,               \
                cudaGetErrorString(error));                             \
        exit(1);                                                        \
    }                                                                   \
}
__global__ void sumReduce(float *dev_a, float *dev_b)
{
//unsigned int i=blockIdx.x*blockDim.x+threadIdx.x;
__shared__ float partialSum[BD];
//for (unsigned int ph = 0; ph < N/BD; ++ph)
{
// Collaborative loading
//if(ph==blockIdx.x)
//{
partialSum[threadIdx.x] = dev_a[blockIdx.x*blockDim.x+threadIdx.x];
//}
//__syncthreads();
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
{
__syncthreads();
if (t % (2*stride) == 0)
{
partialSum[t]+= partialSum[t+stride];
}
}
dev_b[0]=partialSum[0];
}
}
```

```
int main(int argc, char **argv)
{
float  a[N],b[N];
float *dev_a,*dev_b;
int bdimx = BD;
float elapsedTime;
dim3 block(bdimx);
dim3 grid((N + block.x - 1) / block.x,1,1);
cudaEvent_t  start, stop;
CHECK(cudaEventCreate( &start));
CHECK(cudaEventCreate(&stop));
printf("Array Size is=%d\n",N);
// allocate the memory on device
CHECK(cudaMalloc((void**)&dev_a, N*sizeof(float)));
CHECK(cudaMalloc((void**)&dev_b, N*sizeof(float)));
for(int i=0; i<N;i++){
a[i] = ((float)rand()/(float)(RAND_MAX));
}
//Cuda events for time measure
CHECK(cudaEventRecord(start,0));
cudaMemcpy(dev_a, a, N*sizeof(float),cudaMemcpyHostToDevice);
CHECK(cudaEventRecord(stop,0));
CHECK(cudaEventSynchronize(stop));
cudaEventElapsedTime(&elapsedTime,start,stop);
printf("Time to do memory transfer of array a, from host to device is %8.6f ms\n",
    elapsedTime);
CHECK(cudaEventRecord(start,0));
sumReduce<<<grid,block>>>(dev_a,dev_b);
//Copy result from   device to host
CHECK(cudaMemcpy(b,dev_b, N*sizeof(float),cudaMemcpyDeviceToHost));
CHECK(cudaEventRecord(stop,0));
CHECK(cudaEventSynchronize(stop));
cudaEventElapsedTime(&elapsedTime,start,stop);
printf("Time to do sum reducation  is %8.6f ms\n",elapsedTime);
printf("Sum=%f\n",b[0]);
cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaFree(dev_a);
cudaFree(dev_b);
return 0;
}
```

SumReduction.cu

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc  SumReduction.cu
[badam@isrohpc GPULAB]$ ./a.out
Array Size is=1000
Time to do memory transfer of array a, from host to device is 0.0 ms
Time to do sum reducation  is 0.1 ms
Sum=112.796783
```

Lab Exercise 2.12 *Write a CUDA program to demonstrate the followings*

1. *Write a header file for declaring Error function*
2. *Write a device functions to do the sum reduction with less warp divergence*

3. *Print the execution time of the kernel and compare with classical sum reduction as given in 2.11*
4. *Print the result*

Expected output could be similar to 2.11

Experiment 2.11 *Numerical accuracy of fusing a multiply-add*

Objectives: Write a CUDA program for to illustrates the effect on numerical accuracy of fusing and multiply-add into a single MAD instruction.
CUDA Sample Program

```
#include <stdio.h>
#include <stdlib.h>
#define CHECK(call)                                                    \
{                                                                      \
    const cudaError_t error = call;                                    \
    if (error != cudaSuccess)                                          \
    {                                                                  \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);         \
        fprintf(stderr, "code: %d, reason: %s\n", error,               \
                cudaGetErrorString(error));                            \
        exit(1);                                                       \
    }                                                                  \
}
/*A fused multiply add (FMA or fmadd https://en.wikipedia.org/wiki/Multiply%E2%80%93
    accumulate_operation) is a floating-point multiply add (MAD) operation performed in one
     step, with a single rounding. That is, where an unfused multiply add would compute
    the product b   c, round it to N significant bits, add the result to a, and round back
    to N significant bits, a fused multiply add would compute the entire expression a + (b
        c) to its full precision before rounding the final result down to N significant bits
    . */

__global__ void fmad_kernel(double x, double y, double *out)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid == 0)
    {
        *out = x * x + y;
    }
}

double host_fmad_kernel(double x, double y)
{
    return x * x + y;
}

int main(int argc, char **argv)
{
    double *d_out, h_out;
    double x = 2.891903;
    double y = -3.980364;

    double host_value = host_fmad_kernel(x, y);
    CHECK(cudaMalloc((void **)&d_out, sizeof(double)));
    fmad_kernel<<<1, 32>>>(x, y, d_out);
```

```
    CHECK(cudaMemcpy(&h_out, d_out, sizeof(double),
                     cudaMemcpyDeviceToHost));

    if (host_value == h_out)
    {
        printf("The device output the same value as the host.\n");
        printf("The device output is %.20f and the host output is=%.20f\n",h_out, host_value
);
    }
    else
    {
        printf("The device output  and host values are different,  (host-device) is =%e.\n"
,
               fabs(host_value - h_out));
      printf("The device output is %.20f and the host output is=%.20f\n",h_out, host_value);
    }

    return 0;
}
```

fp–mad.cu

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc fp-mad.cu
[badam@isrohpc GPULAB]$ ./a.out
The device output  and host values are different,  (host-device) is =8.881784e-16.
The device output is 4.38273896140900109941 and the host output is=4.38273896140900021123
```

Experiment 2.12 *Floating-point's inability*

Objectives: Write a CUDA program for to illustrates the effect of floating-point's inability due to single and double precision.
CUDA Sample Program

```
#include <stdio.h>
#include <stdlib.h>
#define CHECK(call)                                                     \
{                                                                       \
    const cudaError_t error = call;                                     \
    if (error != cudaSuccess)                                           \
    {                                                                   \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);          \
        fprintf(stderr, "code: %d, reason: %s\n", error,                \
                cudaGetErrorString(error));                             \
        exit(1);                                                        \
    }                                                                   \
}
__global__ void kernel(float *F, double *D)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid == 0)
    {
        *F = 128.1;
        *D = 128.1;
```

```
    }
}

int main(int argc, char **argv)
{
    float *deviceF;
    float h_deviceF;
    double *deviceD;
    double h_deviceD;

    float hostF = 128.1;
    double hostD = 128.1;

    CHECK(cudaMalloc((void **)&deviceF, sizeof(float)));
    CHECK(cudaMalloc((void **)&deviceD, sizeof(double)));
    kernel<<<1, 32>>>(deviceF, deviceD);
    CHECK(cudaMemcpy(&h_deviceF, deviceF, sizeof(float),
                     cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(&h_deviceD, deviceD, sizeof(double),
                     cudaMemcpyDeviceToHost));

    printf("Host single-precision representation of 128.1   = %.20f\n", hostF);
    printf("Host double-precision representation of 128.1   = %.20f\n", hostD);
    printf("Device single-precision representation of 128.1 = %.20f\n", h_deviceF);
    printf("Device double-precision representation of 128.1 = %.20f\n", h_deviceD);
    printf("Device and host single-precision representation equal? %s\n",
           hostF == h_deviceF ? "yes" : "no");
    printf("Device and host double-precision representation equal? %s\n",
           hostD == h_deviceD ? "yes" : "no");

    return 0;
}
```

fpaccuracy.cu

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ ./a.out
Host single-precision representation of 128.1   = 128.10000610351562500000
Host double-precision representation of 128.1   = 128.09999999999999431566
Device single-precision representation of 128.1 = 128.10000610351562500000
Device double-precision representation of 128.1 = 128.09999999999999431566
Device and host single-precision representation equal? yes
Device and host double-precision representation equal? yes
```

Lab Exercise 2.13 *Write a CUDA program to demonstrate the followings:*

1. *Kahan summation algorithm*
2. *Write a header file for declaring Error function*
3. *Write a device functions to do sum of all the element of an arrays with and without sorting.*
4. *Write a program do demonstrate Atomic Sum and Atomic Subtraction of two numbers after the one increment of their values by two threads.*
5. *Print the execution time of the kernel and compare the accuracy of results2.12.*

Algorithm 1 Kahan summation algorithm

Require: *input array*, $n = input\ array\ length$
Ensure: *sum*

$var\ sum \leftarrow 0.0$
$var\ c \leftarrow 0.0$
$i \leftarrow 1$
while $i \leq n$ **do**
  $var\ y \leftarrow input[i] - c$
  $vart \leftarrow sum + y$
  $c \leftarrow (t - sum) - y$
  $sum \leftarrow t$
  $i \leftarrow i + 1$
end while

Expected output could be similar to 2.11

Experiment 2.13 *Single Stream*

Objectives: Write a CUDA program to create a single stream with the use of paged-locked memory and asynchronous data transfer.
CUDA Sample Program

```
#include <stdio.h>
#include <stdlib.h>
#define N    (1024*1024)
#define FULL_DATA_SIZE    (N*20)
#define CHECK(call)                                                          \
{                                                                            \
    const cudaError_t error = call;                                          \
    if (error != cudaSuccess)                                                \
    {                                                                        \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);               \
        fprintf(stderr, "code: %d, reason: %s\n", error,                     \
                cudaGetErrorString(error));                                  \
        exit(1);                                                             \
    }                                                                        \
}

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
       c[idx] = (a[idx] + b[idx]) / 2.0;
    }
}


int main( void ) {
    cudaDeviceProp  prop;
    int whichDevice;
    CHECK( cudaGetDevice( &whichDevice ) );
    CHECK( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no speed up from streams\n" );
        return 0;
```

```
}

cudaEvent_t      start, stop;
float            elapsedTime;

cudaStream_t     stream;
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;

// start the timers
CHECK( cudaEventCreate( &start ) );
CHECK( cudaEventCreate( &stop ) );

// initialize the stream
CHECK( cudaStreamCreate( &stream ) );

// allocate the memory on the GPU
CHECK( cudaMalloc( (void**)&dev_a,
                          N * sizeof(int) ) );
CHECK( cudaMalloc( (void**)&dev_b,
                          N * sizeof(int) ) );
CHECK( cudaMalloc( (void**)&dev_c,
                          N * sizeof(int) ) );

// allocate host locked memory, used to stream
CHECK( cudaHostAlloc( (void**)&host_a,
                          FULL_DATA_SIZE * sizeof(int),
                          cudaHostAllocDefault ) );
CHECK( cudaHostAlloc( (void**)&host_b,
                          FULL_DATA_SIZE * sizeof(int),
                          cudaHostAllocDefault ) );
CHECK( cudaHostAlloc( (void**)&host_c,
                          FULL_DATA_SIZE * sizeof(int),
                          cudaHostAllocDefault ) );

for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}

CHECK( cudaEventRecord( start, 0 ) );
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // copy the locked memory to the device, async
    CHECK( cudaMemcpyAsync( dev_a, host_a+i,
                                   N * sizeof(int),
                                   cudaMemcpyHostToDevice,
                                   stream ) );
    CHECK( cudaMemcpyAsync( dev_b, host_b+i,
                                   N * sizeof(int),
                                   cudaMemcpyHostToDevice,
                                   stream ) );

    kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );

    // copy the data from device to locked memory
    CHECK( cudaMemcpyAsync( host_c+i, dev_c,
                                   N * sizeof(int),
                                   cudaMemcpyDeviceToHost,
                                   stream ) );

}
// copy result chunk from locked to full buffer
CHECK( cudaStreamSynchronize( stream ) );
```

```
    CHECK( cudaEventRecord( stop, 0 ) );

    CHECK( cudaEventSynchronize( stop ) );
    CHECK( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "The single stream  with ID  %d was created and  the  total time taken  for (data transfer, computation) is   %8.6f ms\n", stream,elapsedTime );

    // cleanup the streams and memory
    CHECK( cudaFreeHost( host_a ) );
    CHECK( cudaFreeHost( host_b ) );
    CHECK( cudaFreeHost( host_c ) );
    CHECK( cudaFree( dev_a ) );
    CHECK( cudaFree( dev_b ) );
    CHECK( cudaFree( dev_c ) );
    CHECK( cudaStreamDestroy( stream ) );

    return 0;
}
```

SingleStream.cu

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ ./a.out
The single stream  with ID  27547856 was created and
the  total time taken  for (data transfer, computation) is   21.894527 ms
```

Lab Exercise 2.14 *Write a CUDA program to demonstrate the followings:*

1. *Write a header file for declaring Error function*
2. *Write a CUDA program to perform the sum of arrays and to find the maximum of array using double-streams with paged-locked memory and asynchronous data transfer.*
3. *Print the execution time of the memory transfers and computations*

Expected output could be similar to 2.13

Experiment 2.14 *Device information through a PyCUDA Program*

Objectives: Write a PyCUDA Program for displaying GPU Device information
PyCUDA Sample Program

```
import pycuda.driver as drv
drv.init()
print ("%d device(s) found." % drv.Device.count())
for ordinal in range(drv.Device.count()):
        dev = drv.Device(ordinal)
        print ("Device #%d: %s" % (ordinal, dev.name()))
        print (" Compute Capability: %d.%d" % dev.compute_capability())
        print (" Total Memory: %s KB" % (dev.total_memory()//(1024)))
```

pycudaDevInfo.py

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ python3 pycudaDevInfo.py
2 device(s) found.
Device #0: GeForce GTX TITAN X
 Compute Capability: 5.2
 Total Memory: 12505920 KB
Device #1: GeForce GTX 680
 Compute Capability: 3.0
 Total Memory: 2047552 KB
[badam@isrohpc GPULAB]$
```

Experiment 2.15 *Simple PyCUDA Program*

Objectives:Demonstrate workflow with PyCUDA program for computing 2 times of all the elements of a Matrix
PyCUDA Sample Program

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy
a=numpy.random.randn(5,5)
a=a.astype(numpy.float32)

a_gpu=cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu,a)

mod=SourceModule("""
         __global__ void doubleMatrix(float *a)
         {
         int idx=threadIdx.x+threadIdx.y*4;
         a[idx]*=2;
         }
         """)
func=mod.get_function("doubleMatrix")
func(a_gpu,block=(5,5,1))
a_doubled=numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled,a_gpu)
print("Original Matrix")
print(a)
print("Double Matrix After PyCUDA Execution")
print(a_doubled)
```

SimpleProg.py

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ python3 SimpleProg.py
Original Matrix
[[ 0.03760774  1.604201   -0.39076883  0.30589864 -1.1251544 ]
 [-1.1846496  -0.29308596 -1.1174204  -0.24432212  2.4030788 ]
 [ 0.10457493 -0.5216858  -0.30098775  1.8247517   0.22829506]
```

```
 [-0.34854513  1.0348203  -0.2788698  -0.69622207 -1.5858915 ]
 [-0.03784035  0.7266018   0.36599657 -0.7192867  -1.5846182 ]]
Double Matrix After PyCUDA Execution
[[ 0.07521549  3.208402   -0.78153765  0.6117973  -2.2503088 ]
 [-2.3692992  -0.5861719  -2.2348409  -0.48864424  4.8061576 ]
 [ 0.20914985 -1.0433716  -0.6019755   3.6495035   0.45659012]
 [-0.69709027  2.0696406  -0.5577396  -1.3924441  -3.171783  ]
 [-0.0756807   0.7266018   0.36599657 -0.7192867  -1.5846182 ]]
```

Lab Exercise 2.15 ***Write a PyCUDA program to demonstrate the followings:***

1. ***Allocate host and device memories for three matrices*** A, B, C
2. ***Transfer data of matrices*** A, B ***from host to device***
3. ***Performance Matrix and Matrix multiplication***

Expected output could be similar to 2.13

Experiment 2.16 *PyCUDA with GPUArray and NumbaPro*

Objectives: (1) PyCUDA GPUArray program for computing 2 times of all the elements of a Matrix
Objectives: (2) PyCUDA NumbaPro program for Matrix-Matrix Multiplication
PyCUDA Sample Program

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
from numba import guvectorize
import numpy as np
a_gpu = gpuarray.to_gpu(np.random.randn(5,5).astype(np.float32))
a_doubled = (2*a_gpu).get()
print ("ORIGINAL MATRIX")
print (a_gpu)
print ("DOUBLED MATRIX AFTER PyCUDA EXECUTION USING GPUARRAY CALL")
print (a_doubled)


@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'],
             '(m,n),(n,p)->(m,p)')
def matmul(A, B, C):
    m, n = A.shape
    n, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]

dim = 10
A = np.random.randint(dim,size=(dim, dim))
B = np.random.randint(dim,size=(dim, dim))
```

```
C = matmul(A, B)
print("INPUT MATRIX A")
print(":\n%s" % A)
print("INPUT MATRIX B")
print(":\n%s" % B)
print("RESULT MATRIX C = A*B")
print(":\n%s" % C)
```

GPUArrayNumbaPro.py

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ python3 GPUArrayNumbaPro.py
ORIGINAL MATRIX
[[-0.9645335   0.68201447 -0.1265066  -0.4648311   0.04720533]
 [-0.16570863  1.1681297   0.4138771   1.9788967   0.8305167 ]
 [-0.46075127  1.0986797  -0.08944886  1.4339496   1.0959916 ]
 [-1.1752167   1.1341211  -1.0027288   0.38856977 -0.49455154]
 [-0.51226526  0.3633518  -0.44432122  0.33385575  0.3319665 ]]
DOUBLED MATRIX AFTER PyCUDA EXECUTION USING GPUARRAY CALL
[[-1.929067    1.3640289  -0.2530132  -0.9296622   0.09441066]
 [-0.33141726  2.3362594   0.8277542   3.9577935   1.6610334 ]
 [-0.92150253  2.1973593  -0.17889772  2.8678992   2.1919832 ]
 [-2.3504333   2.2682421  -2.0054576   0.77713954 -0.9891031 ]
 [-1.0245305   0.7267036  -0.88864243  0.6677115   0.663933  ]]
INPUT MATRIX A
:
[[1 4 9 4 0 4 6 8 2 0]
 [6 6 1 6 7 9 0 6 5 3]
 [1 6 3 0 6 4 8 7 0 4]
 [2 4 8 2 3 8 6 7 4 9]
 [7 0 2 4 2 6 3 9 9 6]
 [3 2 7 4 2 8 3 4 9 7]
 [9 6 1 0 2 5 4 9 5 4]
 [4 5 6 2 8 7 9 2 1 4]
 [8 8 0 8 2 7 0 2 9 4]
 [2 1 9 5 7 7 5 5 2 0]]
INPUT MATRIX B
:
[[4 1 0 2 3 0 2 0 9 6]
 [2 8 9 3 1 0 1 0 2 7]
 [5 7 3 9 2 1 4 0 9 0]
 [0 1 7 9 4 5 4 8 6 5]
 [8 4 5 2 7 3 6 9 2 0]
 [4 1 9 1 1 8 8 9 4 4]
 [3 3 6 1 6 1 0 6 5 7]
 [7 9 3 2 1 2 6 3 6 4]
 [7 7 7 9 4 2 4 0 7 4]
 [4 7 2 6 3 9 9 4 2 7]]
RESULT MATRIX C = A*B
```

```
:
[[161 208 201 175  97  87 146 128 230 152]
 [222 214 274 191 143 173 243 222 238 209]
 [184 213 206 109 128 111 166 175 166 176]
 [243 284 264 230 149 200 269 208 266 231]
 [237 234 218 216 142 169 242 173 277 221]
 [227 245 256 257 142 189 251 180 270 206]
 [215 233 206 145 119 115 190 130 243 228]
 [212 207 255 164 172 147 201 227 228 200]
 [185 204 278 232 135 160 208 167 251 244]
 [203 187 228 187 144 130 197 211 242 135]]
```

Experiment 2.17 *PyCUDA with GPUArray and NumbaPro*

Objectives: Device information using pyOpenCL
PyOpenCL Sample Program

```python
import pyopencl as cl


def print_device_info() :
    print('\n' + '=' * 60 + '\nOpenCL Platforms and Devices')
    for platform in cl.get_platforms():
        print('=' * 60)
        print('Platform - Name:  ' + platform.name)
        print('Platform - Vendor:  ' + platform.vendor)
        print('Platform - Version:  ' + platform.version)
        print('Platform - Profile:  ' + platform.profile)

        for device in platform.get_devices():
            print('    ' + '-' * 56)
            print('    Device - Name:  ' \
                  + device.name)
            print('    Device - Type:  ' \
                  + cl.device_type.to_string(device.type))
            print('    Device - Max Clock Speed:  {0} Mhz'\
                  .format(device.max_clock_frequency))
            print('    Device - Compute Units:  {0}'\
                  .format(device.max_compute_units))
            print('    Device - Local Memory:  {0:.0f} KB'\
                  .format(device.local_mem_size/1024.0))
            print('    Device - Constant Memory:  {0:.0f} KB'\
                  .format(device.max_constant_buffer_size/1024.0))
            print('    Device - Global Memory: {0:.0f} GB'\
                  .format(device.global_mem_size/1073741824.0))
            print('    Device - Max Buffer/Image Size: {0:.0f} MB'\
                  .format(device.max_mem_alloc_size/1048576.0))
            print('    Device - Max Work Group Size: {0:.0f}'\
                  .format(device.max_work_group_size))
    print('\n')

if __name__ == "__main__":
    print_device_info()
```

DeviceInfoPyCL.py

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ python3 DeviceInfoPyCL.py

============================================================
OpenCL Platforms and Devices
============================================================
Platform - Name:  NVIDIA CUDA
Platform - Vendor:  NVIDIA Corporation
Platform - Version:  OpenCL 1.2 CUDA 10.0.292
Platform - Profile:  FULL_PROFILE
    --------------------------------------------------------
    Device - Name:  GeForce GTX TITAN X
    Device - Type:  ALL | GPU
    Device - Max Clock Speed:  1076 Mhz
    Device - Compute Units:  24
    Device - Local Memory:  48 KB
    Device - Constant Memory:  64 KB
    Device - Global Memory: 12 GB
    Device - Max Buffer/Image Size: 3053 MB
    Device - Max Work Group Size: 1024
    --------------------------------------------------------
    Device - Name:  GeForce GTX 680
    Device - Type:  ALL | GPU
    Device - Max Clock Speed:  1058 Mhz
    Device - Compute Units:  8
    Device - Local Memory:  48 KB
    Device - Constant Memory:  64 KB
    Device - Global Memory: 2 GB
    Device - Max Buffer/Image Size: 500 MB
    Device - Max Work Group Size: 1024
```

Lab Exercise 2.16 ***Write a PyOpen program to demonstrate the followings:***

1. ***Allocate host and device memories for three matrices*** A, B, C
2. ***Transfer data of matrices*** A, B ***from host to device***
3. ***Performance addition of two Matrices***

Expected output could be similar to 2.17

Experiment 2.18 *NBody simulation*

Objectives: To implement a very simple two-stage NBody simulation
CUDA Sample Program

```
#include "common.h"
#include <sys/time.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <omp.h>
#ifndef SINGLE_PREC
#ifndef DOUBLE_PREC
#define SINGLE_PREC
#endif
#endif

#ifdef SINGLE_PREC

typedef float real;
#define MAX_DIST     200.0f
#define MAX_SPEED    100.0f
#define MASS         2.0f
#define DT           0.00001f
#define LIMIT_DIST   0.000001f
#define POW(x,y)     powf(x,y)
#define SQRT(x)      sqrtf(x)

#else // SINGLE_PREC

typedef double real;
#define MAX_DIST     200.0
#define MAX_SPEED    100.0
#define MASS         2.0
#define DT           0.00001
#define LIMIT_DIST   0.000001
#define POW(x,y)     pow(x,y)
#define SQRT(x)      sqrt(x)

#endif // SINGLE_PREC

#ifdef VALIDATE

/**
 * Host implementation of the NBody simulation.
 **/
static void h_nbody_update_velocity(real *px, real *py,
                                    real *vx, real *vy,
                                    real *ax, real *ay,
                                    int N, int *exceeded_speed, int id)
{
    real total_ax = 0.0f;
    real total_ay = 0.0f;

    real my_x = px[id];
    real my_y = py[id];

    int i = (id + 1) % N;

    while (i != id)
    {
        real other_x = px[i];
        real other_y = py[i];

        real rx = other_x - my_x;
        real ry = other_y - my_y;

        real dist2 = rx * rx + ry * ry;

        if (dist2 < LIMIT_DIST)
        {
            dist2 = LIMIT_DIST;
        }
```

```
        real dist6 = dist2 * dist2 * dist2;
        real s = MASS * (1.0f / SQRT(dist6));
        total_ax += rx * s;
        total_ay += ry * s;

        i = (i + 1) % N;
    }

    ax[id] = total_ax;
    ay[id] = total_ay;

    vx[id] = vx[id] + ax[id];
    vy[id] = vy[id] + ay[id];

    real v = SQRT(POW(vx[id], 2.0) + POW(vy[id], 2.0));

    if (v > MAX_SPEED)
    {
        *exceeded_speed = *exceeded_speed + 1;
    }
}

static void h_nbody_update_position(real *px, real *py,
                                    real *vx, real *vy,
                                    int N, int *beyond_bounds, int id)
{

    px[id] += (vx[id] * DT);
    py[id] += (vy[id] * DT);

    real dist = SQRT(POW(px[id], 2.0) + POW(py[id], 2.0));

    if (dist > MAX_DIST)
    {
        *beyond_bounds = 1;
    }
}
#endif // VALIDATE

/**
 * CUDA implementation of simple NBody.
 **/
__global__ void d_nbody_update_velocity(real *px, real *py,
                                        real *vx, real *vy,
                                        real *ax, real *ay,
                                        int N, int *exceeded_speed)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    real total_ax = 0.0f;
    real total_ay = 0.0f;

    if (tid >= N) return;

    real my_x = px[tid];
    real my_y = py[tid];

    int i = (tid + 1) % N;

    while (i != tid)
    {
        real other_x = px[i];
        real other_y = py[i];

        real rx = other_x - my_x;
        real ry = other_y - my_y;
```

```
        real dist2 = rx * rx + ry * ry;

        if (dist2 < LIMIT_DIST)
        {
            dist2 = LIMIT_DIST;
        }

        real dist6 = dist2 * dist2 * dist2;
        real s = MASS * (1.0f / SQRT(dist6));
        total_ax += rx * s;
        total_ay += ry * s;

        i = (i + 1) % N;
    }

    ax[tid] = total_ax;
    ay[tid] = total_ay;

    vx[tid] = vx[tid] + ax[tid];
    vy[tid] = vy[tid] + ay[tid];

    real v = SQRT(POW(vx[tid], 2.0) + POW(vy[tid], 2.0));

    if (v > MAX_SPEED)
    {
        atomicAdd(exceeded_speed, 1);
    }
}

__global__ void d_nbody_update_position(real *px, real *py,
                                        real *vx, real *vy,
                                        int N, int *beyond_bounds)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid >= N) return;

    px[tid] += (vx[tid] * DT);
    py[tid] += (vy[tid] * DT);

    real dist = SQRT(POW(px[tid], 2.0) + POW(py[tid], 2.0));

    if (dist > MAX_DIST)
    {
        *beyond_bounds = 1;
    }
}

static void print_points(real *x, real *y, int N)
{
    int i;

    for (i = 0; i < N; i++)
    {
        printf("%.20e %.20e\n", x[i], y[i]);
    }
}

int main(int argc, char **argv)
{
    int i;
    int N = 30720;
    int block = 256;
    int iter, niters = 50;
```

```
    real *d_px, *d_py;
    real *d_vx, *d_vy;
    real *d_ax, *d_ay;
    real *h_px, *h_py;
    int *d_exceeded_speed, *d_beyond_bounds;
    int exceeded_speed, beyond_bounds;
#ifdef VALIDATE
    int id;
    real *host_px, *host_py;
    real *host_vx, *host_vy;
    real *host_ax, *host_ay;
    int host_exceeded_speed, host_beyond_bounds;
#endif // VALIDATE

#ifdef SINGLE_PREC
    printf("Using single-precision floating-point values\n");
#else // SINGLE_PREC
    printf("Using double-precision floating-point values\n");
#endif // SINGLE_PREC

#ifdef VALIDATE
    printf("Running host simulation. WARNING, this might take a while.\n");
#endif // VALIDATE

    h_px = (real *)malloc(N * sizeof(real));
    h_py = (real *)malloc(N * sizeof(real));

#ifdef VALIDATE
    host_px = (real *)malloc(N * sizeof(real));
    host_py = (real *)malloc(N * sizeof(real));
    host_vx = (real *)malloc(N * sizeof(real));
    host_vy = (real *)malloc(N * sizeof(real));
    host_ax = (real *)malloc(N * sizeof(real));
    host_ay = (real *)malloc(N * sizeof(real));
#endif // VALIDATE

    for (i = 0; i < N; i++)
    {
        real x = (rand() % 200) - 100;
        real y = (rand() % 200) - 100;

        h_px[i] = x;
        h_py[i] = y;
#ifdef VALIDATE
        host_px[i] = x;
        host_py[i] = y;
#endif // VALIDATE
    }

    CHECK(cudaMalloc((void **)&d_px, N * sizeof(real)));
    CHECK(cudaMalloc((void **)&d_py, N * sizeof(real)));

    CHECK(cudaMalloc((void **)&d_vx, N * sizeof(real)));
    CHECK(cudaMalloc((void **)&d_vy, N * sizeof(real)));

    CHECK(cudaMalloc((void **)&d_ax, N * sizeof(real)));
    CHECK(cudaMalloc((void **)&d_ay, N * sizeof(real)));

    CHECK(cudaMalloc((void **)&d_exceeded_speed, sizeof(int)));
    CHECK(cudaMalloc((void **)&d_beyond_bounds, sizeof(int)));

    CHECK(cudaMemcpy(d_px, h_px, N * sizeof(real), cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_py, h_py, N * sizeof(real), cudaMemcpyHostToDevice));

    CHECK(cudaMemset(d_vx, 0x00, N * sizeof(real)));
```

```
    CHECK(cudaMemset(d_vy, 0x00, N * sizeof(real)));
#ifdef VALIDATE
    memset(host_vx, 0x00, N * sizeof(real));
    memset(host_vy, 0x00, N * sizeof(real));
#endif // VALIDATE

    CHECK(cudaMemset(d_ax, 0x00, N * sizeof(real)));
    CHECK(cudaMemset(d_ay, 0x00, N * sizeof(real)));
#ifdef VALIDATE
    memset(host_ax, 0x00, N * sizeof(real));
    memset(host_ay, 0x00, N * sizeof(real));
#endif // VALIDATE

    double start = seconds();

    for (iter = 0; iter < niters; iter++)
    {
        CHECK(cudaMemset(d_exceeded_speed, 0x00, sizeof(int)));
        CHECK(cudaMemset(d_beyond_bounds, 0x00, sizeof(int)));

        d_nbody_update_velocity<<<N / block, block>>>(d_px, d_py, d_vx, d_vy,
                d_ax, d_ay, N, d_exceeded_speed);
        d_nbody_update_position<<<N / block, block>>>(d_px, d_py, d_vx, d_vy,
                N, d_beyond_bounds);

    }

    CHECK(cudaDeviceSynchronize());
    double exec_time = seconds() - start;

#ifdef VALIDATE

    for (iter = 0; iter < niters; iter++)
    {
        printf("iter=%d\n", iter);
        host_exceeded_speed = 0;
        host_beyond_bounds = 0;

        #pragma omp parallel for
        for (id = 0; id < N; id++)
        {
            h_nbody_update_velocity(host_px, host_py, host_vx, host_vy,
                                    host_ax, host_ay, N, &host_exceeded_speed,
                                    id);
        }

        #pragma omp parallel for
        for (id = 0; id < N; id++)
        {
            h_nbody_update_position(host_px, host_py, host_vx, host_vy,
                                    N, &host_beyond_bounds, id);
        }
    }

#endif // VALIDATE

    CHECK(cudaMemcpy(&exceeded_speed, d_exceeded_speed, sizeof(int),
                     cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(&beyond_bounds, d_beyond_bounds, sizeof(int),
                     cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_px, d_px, N * sizeof(real), cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_py, d_py, N * sizeof(real), cudaMemcpyDeviceToHost));

    print_points(h_px, h_py, 10);
    printf("Any points beyond bounds? %s, # points exceeded velocity %d/%d\n",
```

```
            beyond_bounds > 0 ? "true" : "false", exceeded_speed,
            N);
    printf("Total execution time %f s\n", exec_time);

#ifdef VALIDATE
    double error = 0.0;

    for (i = 0; i < N; i++)
    {
        double dist = sqrt(pow(h_px[i] - host_px[i], 2.0) +
                           pow(h_py[i] - host_py[i], 2.0));
        error += dist;
    }

    error /= N;
    printf("Error = %.20e\n", error);
#endif // VALIDATE

    return 0;
}
```

nbodyGPU.cu

```
#include <sys/time.h>

#ifndef _COMMON_H
#define _COMMON_H

#define CHECK(call)
    \
{
    \
    const cudaError_t error = call;
    \
    if (error != cudaSuccess)
    \
    {
    \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
    \
        fprintf(stderr, "code: %d, reason: %s\n", error,
    \
                cudaGetErrorString(error));
    \
        exit(1);
    \
    }
    \
}

#define CHECK_CUBLAS(call)
    \
{
    \
    cublasStatus_t err;
    \
    if ((err = (call)) != CUBLAS_STATUS_SUCCESS)
    \
    {
    \
        fprintf(stderr, "Got CUBLAS error %d at %s:%d\n", err, __FILE__,
    \
                __LINE__);
    \
```

```
        exit(1);                                                        \
    }                                                                   \
}

#define CHECK_CURAND(call)                                              \
{                                                                       \
    curandStatus_t err;                                                 \
    if ((err = (call)) != CURAND_STATUS_SUCCESS)                        \
    {                                                                   \
        fprintf(stderr, "Got CURAND error %d at %s:%d\n", err, __FILE__, \
                __LINE__);                                              \
        exit(1);                                                        \
    }                                                                   \
}

#define CHECK_CUFFT(call)                                               \
{                                                                       \
    cufftResult err;                                                    \
    if ( (err = (call)) != CUFFT_SUCCESS)                               \
    {                                                                   \
        fprintf(stderr, "Got CUFFT error %d at %s:%d\n", err, __FILE__, \
                __LINE__);                                              \
        exit(1);                                                        \
    }                                                                   \
}

#define CHECK_CUSPARSE(call)                                            \
{                                                                       \
    cusparseStatus_t err;                                               \
    if ((err = (call)) != CUSPARSE_STATUS_SUCCESS)                      \
    {                                                                   \
        fprintf(stderr, "Got error %d at %s:%d\n", err, __FILE__, __LINE__); \
        cudaError_t cuda_err = cudaGetLastError();                      \
        if (cuda_err != cudaSuccess)                                    \
        {                                                               \
            fprintf(stderr, "  CUDA error \"%s\" also detected\n",
```

```
    \
                        cudaGetErrorString(cuda_err));
    \
            }
    \
            exit(1);
    \
        }
    \
}

inline double seconds()
{
    struct timeval tp;
    struct timezone tzp;
    int i = gettimeofday(&tp, &tzp);
    return ((double)tp.tv_sec + (double)tp.tv_usec * 1.e-6);
}

#endif // _COMMON_H
```

common.h

The output will be similar to the followings

```
[badam@isrohpc GPULAB]$ nvcc nbodyGPU.cu
[badam@isrohpc GPULAB]$ ./a.out
Using single-precision floating-point values
8.29334487915039062500e+01 -1.40215482711791992188e+01
7.68599090576171875000e+01 1.49849863052368164062e+01
9.29465179443359375000e+01 3.49770202636718750000e+01
8.58866806030273437500e+01 -8.03452301025390625000e+00
-5.09801368713378906250e+01 -7.88040084838867187500e+01
6.19454917907714843750e+01 -7.29803695678710937500e+01
-9.94156074523925781250e+00 -4.09241943359375000000e+01
6.29818801879882812500e+01 2.60619792938232421875e+01
3.99804077148437500000e+01 -7.39264450073242187500e+01
7.19351501464843750000e+01 3.59459609985351562500e+01
Any points beyond bounds? true, # points exceeded velocity 28529/30720
Total execution time 1.251625 s
```

Lab Exercise 2.17 ***Write a CUDA program to demonstrate the followings:***

1. ***Allocate host and device memories***
2. ***Transfer data from host to device***
3. ***Perform N-body problem for large number of bodies(such a Solar System with thousands of Asteroids)***

Expected output could be similar to 2.18

References

[1] Kirk, D. & Hwu, W. *Programming Massively Parallel Processors: A Hands-on Approach* (Elsevier Science, 2016).

[2] Sanders, J. & Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents* (Pearson Education, 2010).

[3] Stroustrup, B. *The C++ Programming Language: 4th Edition*. Always learning (Addison-Wesley, 2013).

[4] Chapman, S. *FORTRAN FOR SCIENTISTS & ENGINEERS* (McGraw-Hill Education, 2017).

[5] Zaccone, G. *Python Parallel Programming Cookbook* (Packt Publishing, 2015).

[6] Stevens, W. & Rago, S. *Advanced Programming in the UNIX Environment*. Addison-Wesley professional computing series (Addison-Wesley, 2008).