DEPARTMENT OF MATHEMATICS & COMPUTING, IIT(ISM) DHANBAD

Lab Manual

GPU Computing with CUDA Lab (AMC16202): L-T-P: 0-0-3

VII -M.Tech (M&C)

Winter Semester 2020-21

Course Objective
To understand the concepts of General-Purpose GPU Programming To understand GPU Architecture and Performance To learn parallel programming on Single and Multiple GPUs
Learning Outcomes
To write basic and advanced Programs in CUDA To write program to understand host, device and global functions To write CUDA program for vector addition and Dot Product To develop skill for writing Matrix-Matrix Multiplication To write tiled Matrix-Matrix Multiplication and understand memory management To understand warp divergence through CUDA program To write CUDA porgramme for reduction algorithm To understand Atomic Opreration To demonstrate Graphics-Interoperability To Write a CUDA programme to demonstrate streams To write a programme to demarcated Zero-copy Memory

REQUIREMENTS:
HARDWARE Linux/Windows System with Graphics Card, SOFTWRAE: CUDA C/C++, NVCC

COMPONENTS OF LAB MANUAL:

- Aim
- Programme Logic/Steps and CUDA APIs
- Expected Outcomes

Experiments

Experiment No.1: Write a first CUDA program to display Hello World from GPU!

1. **Aim:** To display message Hello World from GPU
2. **Programme Logic/Steps and CUDA APIs:**

```
__global__ void helloFromGPU()
{
   printf("Hello World from GPU!\n");
}
```

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

   helloFromGPU<<<1, 10>>>();
   CHECK(cudaDeviceReset());
   return 0;
}
```

3. **Outcome:** Hello World from CPU and GPU

Experiment No.2: Write a CUDA C program for Vector Addition

Objective

The purpose of this lab is to introduce the student to the CUDA API by implementing vector addition. The student will implement vector addition by writing the GPU kernel code as well as the associated host code.

Prerequisites

Before starting this lab, make sure that:

- You have completed all of Module 2 in the teaching kit
- You have completed the “Device Query” lab

Instructions

Edit the code in the code tab to perform the following:

- Allocate device memory
- Copy host memory to device
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host
- Free device memory
- Write the CUDA kernel

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

Local Setup Instructions

The most recent version of source code for this lab along with the build-scripts can be found on the Bitbucket repository. A description on how to use the CMake tool in along with how to build the labs for local development found in the README document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./VectorAdd_Template -e <expected.raw> -i <intput1.raw>,<input2.raw> \
  -o <output.raw> -t vector
```

where `<expected.raw>` is the expected output, `<input0.raw>,<input1.raw>` is the input dataset, and `<output.raw>` is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

Questions

(1) How many floating operations are being performed in your vector add kernel? EXPLAIN.

ANSWER: N - one for each pair of input vector elements.

(2) How many global memory reads are being performed by your kernel? EXPLAIN.

ANSWER: 2N - one for each input vector element.

(3) How many global memory writes are being performed by your kernel? EXPLAIN.

ANSWER: N - one for each output vector element.

(4) Describe what possible optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: Split into multiple kernels to overlap data transfer and kernel execution.

(5) Name three applications of vector addition.

ANSWER: Big Data analysis, statistics, FFT.

Code Template

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code is the sections demakated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```
#include <wb.h>

__global__ void vecAdd(float *in1, float *in2, float *out, int len) {
  //@@ Insert code to implement vector addition here
}

int main(int argc, char **argv) {
  wbArg_t args;
```

```
int inputLength;
float *hostInput1;
float *hostInput2;
float *hostOutput;
float *deviceInput1;
float *deviceInput2;
float *deviceOutput;

args = wbArg_read(argc, argv);

wbTime_start(Generic, "Importing data and creating memory on host");
hostInput1 =
    (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
hostInput2 =
    (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
hostOutput = (float *)malloc(inputLength * sizeof(float));
wbTime_stop(Generic, "Importing data and creating memory on host");

wbLog(TRACE, "The input length is ", inputLength);

wbTime_start(GPU, "Allocating GPU memory.");
//@@ Allocate GPU memory here

wbTime_stop(GPU, "Allocating GPU memory.");

wbTime_start(GPU, "Copying input memory to the GPU.");
//@@ Copy memory to the GPU here

wbTime_stop(GPU, "Copying input memory to the GPU.");

//@@ Initialize the grid and block dimensions here

wbTime_start(Compute, "Performing CUDA computation");
//@@ Launch the GPU Kernel here

cudaDeviceSynchronize();
wbTime_stop(Compute, "Performing CUDA computation");

wbTime_start(Copy, "Copying output memory to the CPU");
//@@ Copy the GPU memory back to the CPU here

wbTime_stop(Copy, "Copying output memory to the CPU");

wbTime_start(GPU, "Freeing GPU Memory");
//@@ Free the GPU memory here

wbTime_stop(GPU, "Freeing GPU Memory");
```

```
  wbSolution(args, hostOutput, inputLength);

  free(hostInput1);
  free(hostInput2);
  free(hostOutput);

  return 0;
}
```

Code Solution

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```
#include <wb.h>

__global__ void vecAdd(float *in1, float *in2, float *out, int len) {
  //@@ Insert code to implement vector addition here
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < len) {
    out[index] = in1[index] + in2[index];
  }
}

int main(int argc, char **argv) {
  wbArg_t args;
  int inputLength;
  float *hostInput1;
  float *hostInput2;
  float *hostOutput;
  float *deviceInput1;
  float *deviceInput2;
  float *deviceOutput;

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput1 =
      (float *)wbImport(wbArg_getInputFile(args, 0), &inputLength);
  hostInput2 =
      (float *)wbImport(wbArg_getInputFile(args, 1), &inputLength);
  hostOutput = (float *)malloc(inputLength * sizeof(float));
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The input length is ", inputLength);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here
```

```
  cudaMalloc((void **)&deviceInput1, inputLength * sizeof(float));
  cudaMalloc((void **)&deviceInput2, inputLength * sizeof(float));
  cudaMalloc((void **)&deviceOutput, inputLength * sizeof(float));
  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here
  cudaMemcpy(deviceInput1, hostInput1, inputLength * sizeof(float),
             cudaMemcpyHostToDevice);
  cudaMemcpy(deviceInput2, hostInput2, inputLength * sizeof(float),
             cudaMemcpyHostToDevice);
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here
  dim3 blockDim(32);
  dim3 gridDim(ceil(((float)inputLength) / ((float)blockDim.x)));

  wbLog(TRACE, "Block dimension is ", blockDim.x);
  wbLog(TRACE, "Grid dimension is ", gridDim.x);

  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Launch the GPU Kernel here
  vecAdd<<<gridDim, blockDim>>>(deviceInput1, deviceInput2,
deviceOutput,
                                inputLength);
  cudaDeviceSynchronize();
  wbTime_stop(Compute, "Performing CUDA computation");

  wbTime_start(Copy, "Copying output memory to the CPU");
  //@@ Copy the GPU memory back to the CPU here
  cudaMemcpy(hostOutput, deviceOutput, inputLength * sizeof(float),
             cudaMemcpyDeviceToHost);
  wbTime_stop(Copy, "Copying output memory to the CPU");

  wbTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here
  cudaFree(deviceInput1);
  cudaFree(deviceInput2);
  cudaFree(deviceOutput);
  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, inputLength);

  free(hostInput1);
  free(hostInput2);
  free(hostOutput);
```

```
  return 0;
}
```


Experiment. No.3 : Write a CUDA program to implements element-wise addition of matrices on the host and GPU.

1. **Aim:** To implement Sum of Matrices on HOST and GPU
2. **Programme Logic/Steps and CUDA APIs:**

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

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

   for(i = 0; i < size; i++)
   {
      ip[i] = (float)( rand() & 0xFF ) / 10.0f;
   }
}

void sumMatrixOnHost(float *A, float *B, float *C, const int nx, const int ny)
{
   float *ia = A;
   float *ib = B;
   float *ic = C;

   for (int iy = 0; iy < ny; iy++)
   {
      for (int ix = 0; ix < nx; ix++)
      {
         ic[ix] = ia[ix] + ib[ix];
      }

      ia += nx;
      ib += nx;
      ic += nx;
   }

   return;
}

void checkResult(float *hostRef, float *gpuRef, const int N)
{
   double epsilon = 1.0E-8;
```

```
    for (int i = 0; i < N; i++)
    {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon)
        {
            printf("host %f gpu %f ", hostRef[i], gpuRef[i]);
            printf("Arrays do not match.\n\n");
            break;
        }
    }
}

// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *A, float *B, float *C, int NX, int NY)
{
    unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int iy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int idx = iy * NX + ix;

    if (ix < NX && iy < NY)
    {
        C[idx] = A[idx] + B[idx];
    }
}

int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    CHECK(cudaSetDevice(dev));

    // set up data size of matrix
    int nx = 1 << 14;
    int ny = 1 << 14;

    int nxy = nx * ny;
    int nBytes = nxy * sizeof(float);

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

    // initialize data at host side
    size_t iStart = seconds();
```

```
initialData(h_A, nxy);
initialData(h_B, nxy);
size_t iElaps = seconds() - iStart;

memset(hostRef, 0, nBytes);
memset(gpuRef, 0, nBytes);

// add matrix at host side for result checks
iStart = seconds();
sumMatrixOnHost (h_A, h_B, hostRef, nx, ny);
iElaps = seconds() - iStart;

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

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

// invoke kernel at host side
int dimx = 32;
int dimy = 32;

if(argc > 2)
{
   dimx = atoi(argv[1]);
   dimy = atoi(argv[2]);
}

dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);

// execute the kernel
CHECK(cudaDeviceSynchronize());
iStart = seconds();
sumMatrixOnGPU2D<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("sumMatrixOnGPU2D <<<(%d,%d), (%d,%d)>>> elapsed %d ms\n", grid.x,
      grid.y,
      block.x, block.y, iElaps);
CHECK(cudaGetLastError());

// copy kernel result back to host side
CHECK(cudaMemcpy(gpuRef, d_MatC, nBytes, cudaMemcpyDeviceToHost));
```

```
    // check device results
    checkResult(hostRef, gpuRef, nxy);

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

    // free host memory
    free(h_A);
    free(h_B);
    free(hostRef);
    free(gpuRef);

    // reset device
    CHECK(cudaDeviceReset());

    return EXIT_SUCCESS;
}
```

Outcome: Provide the sum of Matrices on HOST and GPU

Experiment No. 5. Write a program to display a variety of information on the first CUDA device in this system

1. **Aim: display the device information**
2. **Major Steps:**

```
#include "../common/common.h"
#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;
    CHECK(cudaSetDevice(dev));
    cudaDeviceProp deviceProp;
```

```
CHECK(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);
printf("  Warp size:                                     %d\n",
      deviceProp.warpSize);
printf("  Maximum number of threads per multiprocessor:  %d\n",
      deviceProp.maxThreadsPerMultiProcessor);
printf("  Maximum number of threads per block:           %d\n",
      deviceProp.maxThreadsPerBlock);
```

```
printf("  Maximum sizes of each dimension of a block:    %d x %d x %d\n",
      deviceProp.maxThreadsDim[0],
      deviceProp.maxThreadsDim[1],
      deviceProp.maxThreadsDim[2]);
printf("  Maximum sizes of each dimension of a grid:     %d x %d x %d\n",
      deviceProp.maxGridSize[0],
      deviceProp.maxGridSize[1],
      deviceProp.maxGridSize[2]);
printf("  Maximum memory pitch:                          %lu bytes\n",
      deviceProp.memPitch);

exit(EXIT_SUCCESS);
```

3. **Output: Device information as per the fields**

Experiment No. 6. Write a CUDA C program for MATRIX MULTIPLICATION

1. **Aim: MATRIX MULTIPLICATION on GPU**
2. **Multiplication Kernel :**

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
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 += M[Row*Width+k]*N[k*Width+Col];
}
P[Row*Width+Col] = Pvalue;
}
}
```

3. **Output: Matrix Mulication**

Experiment No. 7. A TILED MATRIX MULTIPLICATION KERNEL

1. **Aim:** To display results after titled matrix-matrix multiplication
2. **Major Steps:**

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
int Width) {
1.
2. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
3.
4. int bx = blockIdx.x; int by = blockIdx.y;
```

```
int tx = threadIdx.x; int ty = threadIdx.y;
5.
6. // Identify the row and column of the d_P element to work on
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;
7.
8. float Pvalue = 0;
// Loop over the d_M and d_N tiles required to compute d_P element
for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
9.
10.
11. // Collaborative loading of d_M and d_N tiles into shared memory
Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
Nds[ty][tx] = d_N[(ph*TILE_WIDTH + ty)*Width + Col];
__syncthreads();
12.
13.
for (int k = 0; k < TILE_WIDTH; ++k) {
Pvalue += Mds[ty][k] * Nds[k][tx];
}
__syncthreads();
14.
}
d_P[Row*Width + Col] = Pvalue;
15.
}
```

Tiled matrix multiplication kernel with boundary condition checks:

```
// Loop over the M and N tiles required to compute P element
8.
for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ++ph) {
9.
10.
// Collaborative loading of M and N tiles into shared memory
if ((Row< Width) && (ph*TILE_WIDTH+tx)< Width)
Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
if ((ph*TILE_WIDTH+ty)<Width && Col<Width)
Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11. __syncthreads();
12.
13. for (int k = 0; k < TILE_WIDTH; ++k) {
Pvalue += Mds[ty][k] * Nds[k][tx];
}
__syncthreads();
14.
15.
}
if ((Row<Width) && (Col<Width)P[Row*Width + Col] = Pvalue;
```

3. **Output: Display Matrix -Matrix Multiplication result**

Experiment No. 8. **Write a CUDA programme to demonstrate Warp divergence**

1. **Aim: To Demonstrate Warp Divergence**
2. **Major Steps:**

```
S#include <cuda_runtime.h>
#include <stdio.h>
#inclide<sys/time.h>

/*
 * simpleDivergence demonstrates divergent code on the GPU and its impact on
 * performance and CUDA metrics.
 */
#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);                                                         \
   }                                                                   \
}


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);
}

__global__ void mathKernel1(float *c)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   float ia, ib;
   ia = ib = 0.0f;

   if (tid % 2 == 0)
   {
      ia = 100.0f;
   }
   else
   {
      ib = 200.0f;
   }
```

```
    c[tid] = ia + ib;
}

__global__ void mathKernel2(float *c)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float ia, ib;
    ia = ib = 0.0f;

    if ((tid / warpSize) % 2 == 0)
    {
        ia = 100.0f;
    }
    else
    {
        ib = 200.0f;
    }

    c[tid] = ia + ib;
}

__global__ void mathKernel3(float *c)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float ia, ib;
    ia = ib = 0.0f;

    bool ipred = (tid % 2 == 0);

    if (ipred)
    {
        ia = 100.0f;
    }

    if (!ipred)
    {
        ib = 200.0f;
    }

    c[tid] = ia + ib;
}

__global__ void mathKernel4(float *c)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float ia, ib;
    ia = ib = 0.0f;
```

```
    int itid = tid >> 5;

    if (itid & 0x01 == 0)
    {
        ia = 100.0f;
    }
    else
    {
        ib = 200.0f;
    }

    c[tid] = ia + ib;
}

__global__ void warmingup(float *c)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float ia, ib;
    ia = ib = 0.0f;

    if ((tid / warpSize) % 2 == 0)
    {
        ia = 100.0f;
    }
    else
    {
        ib = 200.0f;
    }

    c[tid] = ia + ib;
}


int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s using Device %d: %s\n", argv[0], dev, deviceProp.name);

    // set up data size
    int size = 64;
    int blocksize = 64;

    if(argc > 1) blocksize = atoi(argv[1]);

    if(argc > 2) size      = atoi(argv[2]);
```

```
printf("Data size %d ", size);

// set up execution configuration
dim3 block (blocksize, 1);
dim3 grid  ((size + block.x - 1) / block.x, 1);
printf("Execution Configure (block %d grid %d)\n", block.x, grid.x);

// allocate gpu memory
float *d_C;
size_t nBytes = size * sizeof(float);
CHECK(cudaMalloc((float**)&d_C, nBytes));

// run a warmup kernel to remove overhead
size_t iStart, iElaps;
CHECK(cudaDeviceSynchronize());
iStart = seconds();
warmingup<<<grid, block>>>(d_C);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("warmup      <<< %4d %4d >>> elapsed %d sec \n", grid.x, block.x,
      iElaps );
CHECK(cudaGetLastError());

// run kernel 1
iStart = seconds();
mathKernel1<<<grid, block>>>(d_C);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("mathKernel1 <<< %4d %4d >>> elapsed %d sec \n", grid.x, block.x,
      iElaps );
CHECK(cudaGetLastError());

// run kernel 3
iStart = seconds();
mathKernel2<<<grid, block>>>(d_C);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("mathKernel2 <<< %4d %4d >>> elapsed %d sec \n", grid.x, block.x,
      iElaps );
CHECK(cudaGetLastError());

// run kernel 3
iStart = seconds();
mathKernel3<<<grid, block>>>(d_C);
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("mathKernel3 <<< %4d %4d >>> elapsed %d sec \n", grid.x, block.x,
      iElaps);
CHECK(cudaGetLastError());
```

```
    // run kernel 4
    iStart = seconds();
    mathKernel4<<<grid, block>>>(d_C);
    CHECK(cudaDeviceSynchronize());
    iElaps = seconds() - iStart;
    printf("mathKernel4 <<< %4d %4d >>> elapsed %d sec \n", grid.x, block.x,
        iElaps);
    CHECK(cudaGetLastError());

    // free gpu memory and reset divece
    CHECK(cudaFree(d_C));
    CHECK(cudaDeviceReset());
    return EXIT_SUCCESS;
}
```

3. **Output: Warp divergence results**

Experiment No. 9. Write a CUDA program for sum reduction

1. **Aim: To demonstrate sum reduction**
2. **Major Steps:**

Serial Reduction:

```
float sum(float *data, int n)
{
float result = 0;
for(int i = 0; i < n; ++i)
{
result += data[i];
}
return result;
}
```

Parallel Reduction:

```
__global__ void block_sum(float *input, float *results,size_t n)
{
extern __shared__ float sdata[];
int i = ..., int tx = threadIdx.x;
// load input into __shared__ memory
float x = 0;
if(i < n)
x = input[i];
sdata[tx] = x;
__syncthreads();

// block-wide reduction in __shared__ mem
for(int offset = blockDim.x / 2;
offset > 0;
offset >>= 1)
{
if(tx < offset)
```

```
{
// add a partial sum upstream to our own
sdata[tx] += sdata[tx + offset];
}
__syncthreads();
}

// finally, thread 0 writes the result
if(threadIdx.x == 0)
{
// note that the result is per-block
// not per-thread
results[blockIdx.x] = sdata[0];
}
}
```

3. **Output: Sum reduction**

Experiment No. 10. Write a CUDA programme to demonstrates floating-point's accuracy

1. **Aim: Single and double-precision floating point accuracy**
2. **Major Steps:**

```
S#include "../common/common.h"
#include <stdio.h>
#include <stdlib.h>

__global__ void kernel(float *F, double *D)
{
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   if (tid == 0)
   {
      *F = 12.1;
      *D = 12.1;
   }
}

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

   float hostF = 12.1;
   double hostD = 12.1;

   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 12.1   = %.20f\n", hostF);
    printf("Host double-precision representation of 12.1   = %.20f\n", hostD);
    printf("Device single-precision representation of 12.1 = %.20f\n", hostF);
    printf("Device double-precision representation of 12.1 = %.20f\n", hostD);
    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;
}
```

3. **Output: Accuracy in floating-point representation**

Experiment No. 11.Write a progmamme to illustrate implementation of custom atomic operations using atomicCAS

1. **Aim: To develop the Ping-Pong Game**
2. **Major Steps:**

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

__device__ int myAtomicAdd(int *address, int incr)
{
    // Create an initial guess for the value stored at *address.
    int guess = *address;
    int oldValue = atomicCAS(address, guess, guess + incr);

    // Loop while the guess is incorrect.
    while (oldValue != guess)
    {
        guess = oldValue;
        oldValue = atomicCAS(address, guess, guess + incr);
    }

    return oldValue;
}

__global__ void kernel(int *sharedInteger)
{
    myAtomicAdd(sharedInteger, 1);
}

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

```
{
   int h_sharedInteger;
   int *d_sharedInteger;
   CHECK(cudaMalloc((void **)&d_sharedInteger, sizeof(int)));
   CHECK(cudaMemset(d_sharedInteger, 0x00, sizeof(int)));

   kernel<<<4, 128>>>(d_sharedInteger);

   CHECK(cudaMemcpy(&h_sharedInteger, d_sharedInteger, sizeof(int),
            cudaMemcpyDeviceToHost));
   printf("4 x 128 increments led to value of %d\n", h_sharedInteger);

   return 0;
}
```

3. **Output: Results related to Automic addition**

Experiment No. 12. Write a CUDA Heat Transfer with Graphics Inter-operability

1. **Aim: To demonstrate CUDA-OpenGL Graphics Inter-operability**
2. **Major Steps:**

```
static void HandleError( cudaError_t err,  const char *file,  int line ) {
   if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ), file, line );
      exit( EXIT_FAILURE );
   }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))


#define HANDLE_NULL( a ) {if (a == NULL) { \
                        printf( "Host memory failed in %s at line %d\n", \
                              __FILE__, __LINE__ ); \
                        exit( EXIT_FAILURE );}}


glDrawPixels( bitmap->x,bitmap->y,GL_RGBA,GL_UNSIGNED_BYTE,bitmap-
>pixels );

void generate_frame( DataBlock *d, int ticks ) {
dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( d->dev_bitmap, ticks );
HANDLE_ERROR( cudaMemcpy( d→bitmap→get_ptr(),d→dev_bitmap,d→bitmap→i
mage_size(), cudaMemcpyDeviceToHost ) );
}

void anim_gpu( uchar4* outputBitmap, DataBlock *d, int ticks ) {
HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
dim3 blocks(DIM/16,DIM/16);
```

```
dim3 threads(16,16);
// since tex is global and bound, we have to use a flag to
// select which is in/out per iteration
volatile bool dstOut = true;
for (int i=0; i<90; i++) {
*in, *out;
float
if (dstOut) {
in= d->dev_inSrc;
out = d->dev_outSrc;
} else {
out = d->dev_inSrc;
in= d->dev_outSrc;
}
copy_const_kernel<<<blocks,threads>>>( in );
blend_kernel<<<blocks,threads>>>( out, dstOut );
dstOut = !dstOut;
}
float_to_color<<<blocks,threads>>>( outputBitmap,
d->dev_inSrc );
HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
float
elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
d->start, d->stop ) );
d->totalTime += elapsedTime;
++d->frames;
printf( "Average Time per frame:d->totalTime/d->frames%3.1f ms\n",
);
}

__global__ void float_to_color( unsigned char *optr,const float *outSrc ) {
// convert floating-point value to 4-component color
optr[offset*4 + 0] = value( m1, m2, h+120 );
optr[offset*4 + 1] = value( m1, m2, h );
optr[offset*4 + 2] = value( m1, m2, h -120 );
optr[offset*4 + 3] = 255;
}
__global__ void float_to_color( uchar4 *optr,const float *outSrc ) {
// convert floating-point value to 4-component color
optr[offset].x = value( m1, m2, h+120 );
optr[offset].y = value( m1, m2, h );
optr[offset].z = value( m1, m2, h -120 );
optr[offset].w = 255;
}
int main( void ) {
DataBlock data;
GPUAnimBitmap bitmap( DIM, DIM, &data );
```

```
data.totalTime = 0;
data.frames = 0;
HANDLE_ERROR( cudaEventCreate( &data.start ) );
HANDLE_ERROR( cudaEventCreate( &data.stop ) );
int imageSize = bitmap.image_size();
// assume float == 4 chars in size (i.e., rgba)
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL,
texConstSrc,data.dev_constSrc,imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texIn,data.dev_inSrc,imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texOut,data.dev_outSrc,imageSize ) );
// initialize the constant data
float *temp = (float*)malloc( imageSize );
for (int i=0; i<DIM*DIM; i++) {
temp[i] = 0;
int x = i % DIM;
int y = i / DIM;
if ((x>300) && (x<600) && (y>310) && (y<601))
temp[i] = MAX_TEMP;
}
temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
for (int x=400; x<500; x++) {
temp[x+y*DIM] = MIN_TEMP;
}
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,imageSize,
cudaMemcpyHostToDevice ) );
// initialize the input data
for (int y=800; y<DIM; y++) {
for (int x=0; x<200; x++) {
temp[x+y*DIM] = MAX_TEMP;
}
}
HANDLE_ERROR( cudaMemcpy( data.dev_inSrc,
temp,imageSize,cudaMemcpyHostToDevice ) );
free( temp );
bitmap.anim_and_exit( (void (*)(uchar4*,void*,int))anim_gpu,(void (*)
(void*))anim_exit );
}
```

3. **Output: Graphics Interoperability**

Experiment No. 13. Write a CUDA programme to illustrate the use of Streams

1. **Aim:** To demonstrate use of streams
2. **Major Steps:**

```
#define N
(1024*1024)
#define FULL_DATA_SIZE
(N*20)
__global__ void kernel( int *a, int *b, int *c ) {
int idx = threadIdx.x + blockIdx.x * blockDim.x;
if (idx < N) {
int idx1 = (idx + 1) % 256;
int idx2 = (idx + 2) % 256;
float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
c[idx] = (as + bs) / 2;
}
}

int main( void ) {
cudaDeviceProp
prop;
int whichDevice;
HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
HANDLE_ERROR( 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;
// start the timers
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// initialize the stream
cudaStream_t
stream;
HANDLE_ERROR( cudaStreamCreate( &stream ) );
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
```

```
N * sizeof(int) ) );

// allocate page-locked memory, used to stream
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,
FULL_DATA_SIZE * sizeof(int),
cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,
FULL_DATA_SIZE * sizeof(int),
cudaHostAllocDefault ) );
HANDLE_ERROR( 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();
}

// 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
HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i,
N * sizeof(int),
cudaMemcpyHostToDevice,
stream ) );
HANDLE_ERROR( 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
HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c,
N * sizeof(int),
cudaMemcpyDeviceToHost,
stream ) );
}

// copy result chunk from locked to full buffer
HANDLE_ERROR( cudaStreamSynchronize( stream ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
start, stop ) );
printf( "Time taken:
%3.1f ms\n", elapsedTime );
// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
```

```
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
HANDLE_ERROR( cudaStreamDestroy( stream ) );
return 0;
}
```

3. **Output:** Use of **CUDA Streams**

Experiment No. 13. ZERO-COPY DOT PRODUCT

1. **Aim: To demonstrate ZERO-COPY DOT PRODUCT**
2. **Major Steps:**

```
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
__global__ void dot( int size, float *a, float *b, float *c ) {
__shared__ float cache[threadsPerBlock];
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;
float
temp = 0;
while (tid < size) {
temp += a[tid] * b[tid];
tid += blockDim.x * gridDim.x;
}
// set the cache values
cache[cacheIndex] = temp;
// synchronize threads in this block
__syncthreads();
// for reductions, threadsPerBlock must be a power of 2
// because of the following code
int i = blockDim.x/2;
while (i != 0) {
if (cacheIndex < i)
cache[cacheIndex] += cache[cacheIndex + i];
__syncthreads();
i /= 2;
}
if (cacheIndex == 0)
c[blockIdx.x] = cache[0];
}


float malloc_test( int size ) {
cudaEvent_t start, stop;
float *a, *b, c, *partial_c;
float *dev_a, *dev_b, *dev_partial_c;
```

```
float elapsedTime;
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
// allocate memory on the CPU side
a = (float*)malloc( size*sizeof(float) );
b = (float*)malloc( size*sizeof(float) );
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
blocksPerGrid*sizeof(float) ) );
// fill in the host memory with data
for (int i=0; i<size; i++) {
a[i] = i;
b[i] = i*2;
}
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, size*sizeof(float),
cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, size*sizeof(float),
cudaMemcpyHostToDevice ) );
dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
dev_partial_c );
// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
blocksPerGrid*sizeof(float),
cudaMemcpyDeviceToHost ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
start, stop ) );
// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
c += partial_c[i];
}
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );
// free memory on the CPU side
free( a );
free( b );
free( partial_c );
// free events
HANDLE_ERROR( cudaEventDestroy( start ) );
```

```
HANDLE_ERROR( cudaEventDestroy( stop ) );
printf( "Value calculated:
%f\n", c );
return elapsedTime;
}
float cuda_host_alloc_test( int size ) {
cudaEvent_t start, stop;
float *a, *b, c, *partial_c;
float *dev_a, *dev_b, *dev_partial_c;
float elapsedTime;
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
// allocate the memory on the CPU
HANDLE_ERROR( cudaHostAlloc( (void**)&a,
size*sizeof(float),
cudaHostAllocWriteCombined |
cudaHostAllocMapped ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&b,
size*sizeof(float),
cudaHostAllocWriteCombined |
cudaHostAllocMapped ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&partial_c,
blocksPerGrid*sizeof(float),
cudaHostAllocMapped ) );
// fill in the host memory with data
for (int i=0; i<size; i++) {
a[i] = i;
b[i] = i*2;
}
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_a, a, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_b, b, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_partial_c,
partial_c, 0 ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
dev_partial_c );
HANDLE_ERROR( cudaThreadSynchronize() );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
start, stop ) );
// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
c += partial_c[i];
}
HANDLE_ERROR( cudaFreeHost( a ) );
HANDLE_ERROR( cudaFreeHost( b ) );
HANDLE_ERROR( cudaFreeHost( partial_c ) );
```

```
// free events
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
printf( "Value calculated:
%f\n", c );
return elapsedTime;
}
int main( void ) {
cudaDeviceProp
prop;
int whichDevice;
HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
if (prop.canMapHostMemory != 1) {
printf( "Device cannot map memory.\n" );
return 0;
}
HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
float elapsedTime = malloc_test( N );
printf( "Time using cudaMalloc:
%3.1f ms\n",
elapsedTime );
elapsedTime = cuda_host_alloc_test( N );
printf( "Time using cudaHostAlloc:
%3.1f ms\n",
elapsedTime );
}
```

3. **Output:** Provide results related to the u**se of Zero-Copy Memory**

Text Books: David B. Kirk:Programming Massively Parallel Processors: A Hands-on Approac, Wen-mei W. Hwu, Elsevier, 2016

Reference Books:

1. John Cheng, Max Grossman, Ty McKercher: Professional CUDA C Programming,John Wiley & Sons, 2014
2. Shane Cook: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs *Applications of GPU computing series Morgan Kaufmann,Newnes, 2012*
3. Jason Sanders:CUDA by Example: An Introduction to General-Purpose GPU Programming, Edward Kandrot, publisher Addison-Wesley Professional, 2010

Note:1. Lab assignments per week
2. Some Weightage will be given to Attendance/Assignments

(B.S. Kushvah)
Instructors

Date:04/01/2021