```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```

```
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-repo-ubuntu
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
!apt-get update
!apt-get install cuda-9.2
```

```
!nvcc --version
```

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

```
%load_ext nvcc_plugin
```

ARUNIMA SINGH THAKUR

SECTION C

ROLL NO. 31

180905218

PP LAB 8

2nd JUNE 2021

1) 1D convolution

```
%%cu
#include<stdio.h>
#include<stdlib.h>
#include"cuda_runtime.h"
#include"device_launch_parameters.h"

__global__ void convolution_1D(int *N,int *M,int *P,int Mask_width,int Width)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int p=0;
  int NStartPoint=i-(Mask_width/2); //calculating starting point for ith point in P
  for(int j=0;j<Mask_width;j++)
  {
    if(NStartPoint+j>=0 && NStartPoint+j<Width)//calculating only for non ghost elements
    {
      p+=N[NStartPoint+j]*M[j];
    }
  }
```

```
    }
    P[i]=p;//save in output array
  }

  int main()
  {
      int Width=7;//width of input and output
      int Mask_width=5;//width of mask array
      //declarations
      int N[]={100,200,300,400,500,600,700};
      int M[]={1,2,3,4,5};
      int P[Width];
      int size=Width*sizeof(int);
      int *d_N,*d_M,*d_P;
      cudaEvent_t begin,end;
      //memory allocations
      cudaMalloc((void**)&d_N,size);
      cudaMalloc((void**)&d_M,Mask_width*sizeof(int));
      cudaMalloc((void**)&d_P,size);
      //copy from host to devices
      cudaMemcpy(d_N,N,size,cudaMemcpyHostToDevice);
      cudaMemcpy(d_M,M,Mask_width*sizeof(int),cudaMemcpyHostToDevice);
      cudaEventCreate(&begin);
      cudaEventCreate(&end);
      cudaEventRecord(begin,0);
      //run kernel
      convolution_1D<<<1,Width>>>(d_N,d_M,d_P,Mask_width,Width);
      cudaEventRecord(end,0);
      cudaEventSynchronize(end);
      float elapsedTime;
      cudaEventElapsedTime(&elapsedTime,begin,end);
      //copy from device to host
      cudaMemcpy(P,d_P,size,cudaMemcpyDeviceToHost);

      for(int i=0;i<Width;i++)
      {
          printf("%d\t",P[i]);
      }
      printf("\nTime taken by kernel= %f",elapsedTime);
      //free memory
      cudaFree(d_N);
      cudaFree(d_M);
      cudaFree(d_P);
      return 0;
  }
```

OUTPUT:

```
2600 4000 5500 7000 8500 6000 3800
Time taken by kernel= 0.028672
```

2) 1D convolution with constant memory for mask array(kernel1)and shared memory for N (kernel2)

```
%%cu
#include<stdio.h>
#include<stdlib.h>
#include"cuda_runtime.h"
#include"device_launch_parameters.h"
#define MAX_MASK_WIDTH 10

__constant__ float M[MAX_MASK_WIDTH];//declarations of mask array in constant memory

__global__ void convo1Dconstmem(float *N,float *P,int Mask_width,int Width)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int p=0;
  int NStartPoint=i-(Mask_width/2); //calculating starting point for ith point in P
  for(int j=0;j<Mask_width;j++)
  {
    if(NStartPoint+j>=0 && NStartPoint+j<Width)//calculating only for non ghost elements
    {
      p+=N[NStartPoint+j]*M[j];
    }
  }
  P[i]=p;//save in output array
}

__global__ void convo1Dsharedmem(float *N,float *P,int Mask_width,int Width)
{
  extern __shared__ float temp_array[];//declaring a array in shared memory
  int id=blockIdx.x*blockDim.x+threadIdx.x;
  temp_array[threadIdx.x]=N[id];
  __syncthreads();
  float p=0;
  for(int i=0;i<Mask_width;i++)
  {
    if(threadIdx.x+i>=blockDim.x)
    {
      p+=N[id+i]*M[i];
    }
    else
    {
      p+=temp_array[threadIdx.x+i]*M[i];
    }
  }
  P[id]=p;//save in output array
}

int main()
{
    //declarations
    int Width=7;
    int Mask_width=5;
    float h_N[]={100,200,300,400,500,600,700};
    float h_M[]={1,2,3,4,5};
    float h_P[Width];
    int size=Width*sizeof(float);
    float *d_N,*d_P;
    float eTime1,eTime2;
    cudaEvent_t begin,end,begin1,end1;
    //memory allocations
    cudaMalloc((void**)&d_N,size);
    cudaMalloc((void**)&d_P,size);
    //copy from host to device
    cudaMemcpy(d_N,h_N,size,cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(M,h_M,Mask_width*sizeof(float));
    cudaEventCreate(&begin);
    cudaEventCreate(&end);
    cudaEventCreate(&begin1);
    cudaEventCreate(&end1);
    cudaEventRecord(begin,0);
    //run kernel
    convo1Dconstmem<<<1,Width>>>(d_N,d_P,Mask_width,Width);
    cudaEventRecord(end,0);
    cudaEventSynchronize(end);
    cudaEventElapsedTime(&eTime1,begin,end);
    //copy from device to host
    cudaMemcpy(h_P,d_P,size,cudaMemcpyDeviceToHost);
    for(int i=0;i<Width;i++)
    {
        printf("%f\t",h_P[i]);
    }
    printf("\nTime taken by kernel 1= %f\n",eTime1);
    cudaEventRecord(begin1,0);
    //run kernel
    convo1Dsharedmem<<<1,Width>>>(d_N,d_P,Mask_width,Width);
    cudaEventRecord(end1,0);
    cudaEventSynchronize(end1);
    cudaEventElapsedTime(&eTime2,begin1,end1);
    //copy from device to host
    cudaMemcpy(h_P,d_P,size,cudaMemcpyDeviceToHost);
    for(int i=0;i<Width;i++)
    {
        printf("%f\t",h_P[i]);
    }
    printf("\nTime taken by kernel 2= %f",eTime2);
    //free memory
    cudaFree(d_N);
    cudaFree(d_P);
    return 0;
}
```

OUTPUT:

```
2600.000000 4000.000000 5500.000000 7000.000000 8500.000000 6000.000000 3800.000000
Time taken by kernel 1= 0.022176 2600.000000 4000.000000 5500.000000 7000.000000
8500.000000 6000.000000 3800.000000
Time taken by kernel 2= 0.000000
```

3) Sparse Matrix vector multiplication (CSR format)

```
%%cu
#include<stdio.h>
#include<stdlib.h>
#include"cuda_runtime.h"
#include"device_launch_parameters.h"

__global__ void SpMV_CSR(int num_rows,int *data,int *col_index,int *row_ptr,int *x,int *y)
{
  int row=threadIdx.x;
  if(row<num_rows)
  {
    int dot=0;
    int row_start=row_ptr[row];
    int row_end=row_ptr[row+1];
    for(int i=row_start;i<row_end;i++)
    {
      dot+= data[i]*x[col_index[i]];
    }
    y[row]=dot;
  }
}

int main()
{
    //declarations
    int n=4;
    int y[n],row_ptr[n+1];
    int ipmat[n][n]={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
    int x[]={100,200,300,400};
    int nonzerocount=0;
    //finding number of non zero elements and row ptr array
    for(int i=0;i<n;i++)
    {
      row_ptr[i]=nonzerocount;
      for(int j=0;j<n;j++)
      {
        if(ipmat[i][j]!=0)
        {
          nonzerocount++;
        }
        printf("%d\t",ipmat[i][j]);
      }
      printf("\n");
    }
    row_ptr[n]=nonzerocount;
    int data[nonzerocount],col_index[nonzerocount];
    int k=0;
    //finding data and col_index array
    for(int i=0;i<n;i++)
    {
      for(int j=0;j<n;j++)
      {
        if(ipmat[i][j]!=0)
```

```
          {
            data[k]=ipmat[i][j];
            col_index[k++]=j;
          }
        }
      }
      printf("\ndata array\t");
      for(int i=0;i<nonzerocount;i++)
      {
        printf("%d\t",data[i]);
      }
      printf("\ncol_index array\t");
      for(int i=0;i<nonzerocount;i++)
      {
        printf("%d\t",col_index[i]);
      }
      printf("\nrow_ptr array\t");
      for(int i=0;i<=n;i++)
      {
        printf("%d\t",row_ptr[i]);
      }
      printf("\nvector X\t");
      for(int i=0;i<n;i++)
      {
        printf("%d\t",x[i]);
      }
      int *d_data,*d_col_index,*d_row_ptr,*d_x,*d_y;
      //memory allocations
      cudaMalloc((void**)&d_data,nonzerocount*sizeof(int));
      cudaMalloc((void**)&d_col_index,nonzerocount*sizeof(int));
      cudaMalloc((void**)&d_row_ptr,(n+1)*sizeof(int));
      cudaMalloc((void**)&d_x,n*sizeof(int));
      cudaMalloc((void**)&d_y,n*sizeof(int));
      //copy from host to device
      cudaMemcpy(d_data,data,nonzerocount*sizeof(int),cudaMemcpyHostToDevice);
      cudaMemcpy(d_col_index,col_index,nonzerocount*sizeof(int),cudaMemcpyHostToDevice);
      cudaMemcpy(d_row_ptr,row_ptr,(n+1)*sizeof(int),cudaMemcpyHostToDevice);
      cudaMemcpy(d_x,x,n*sizeof(int),cudaMemcpyHostToDevice);
      //run kernel
      SpMV_CSR<<<1,n>>>(n,d_data,d_col_index,d_row_ptr,d_x,d_y);
      //copy from device to host
      cudaMemcpy(y,d_y,n*sizeof(int),cudaMemcpyDeviceToHost);

      printf("\nresult\t");
      for(int i=0;i<n;i++)
      {
          printf("%d\t",y[i]);
      }
      //free memory
      cudaFree(d_data);
      cudaFree(d_col_index);
      cudaFree(d_row_ptr);
      cudaFree(d_x);
      cudaFree(d_y);
      return 0;
```

```
}
```

OUTPUT:

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

data array 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
col_index array 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
row_ptr array 0 4 8 12 16
vector X 100 200 300 400
result 3000 7000 11000 15000
```

4) Matrix multiplication using 2D grids and 2D blocks

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

__global__ void matrixMul(const int *a, const int *b, int *c, int m,int n,int o)
{
  //row and col calculations
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  c[row * o + col] = 0;
  //calculating one element
  for (int k = 0; k < n; k++) {
    c[row * o + col] += a[row * n + k] * b[k * o + col];
  }
}

int main()
{
  //declarations
  int size =sizeof(int);
  int m=4,n=2,o=4;
  int a[m][n];
  int b[n][o];
  int c[m][o];
  for(int i=0;i<m;i++)
  {
    for(int j=0;j<n;j++)
    {
      a[i][j]=1+i+j;
    }
  }
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<o;j++)
    {
      b[i][j]=2+i+j;
    }
  }
  int *d_a, *d_b, *d_c;
  //memory allocations
  cudaMalloc(&d_a,m*n*size);
  cudaMalloc(&d_b,n*o*size);
  cudaMalloc(&d_c,m*o*size);
  //copy from host to device
  cudaMemcpy(d_a,a,m*n*size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b,b,n*o*size, cudaMemcpyHostToDevice);
  //dimensions for grid and block
  int thread=2;
  dim3 threads(thread,thread);
  dim3 blocks((m*o)/(4*thread),(m*o)/(4*thread));

  //run kernel
  matrixMul<<<blocks, threads>>>(d_a, d_b, d_c,m,n,o);

  // copy from device to host
  cudaMemcpy(c, d_c,m*o*size, cudaMemcpyDeviceToHost);
  printf("Matrix A\n");
  for(int i=0;i<m;i++)
  {
    for(int j=0;j<n;j++)
    {
      printf("%d\t",a[i][j]);
    }
    printf("\n");
  }
  printf("Matrix B\n");
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<o;j++)
    {
      printf("%d\t",b[i][j]);
    }
    printf("\n");
  }
  printf("Matrix C\n");
  for(int i=0;i<m;i++)
  {
    for(int j=0;j<o;j++)
    {
      printf("%d\t",c[i][j]);
    }
    printf("\n");
  }
  //free memory
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
```

```
}
```

OUTPUT:

```
Matrix A 1 2
2 3
3 4
4 5
Matrix B 2 3 4 5
3 4 5 6
Matrix C 8 11 14 17
13 18 23 28
18 25 32 39
23 32 41 50
```

✓ 0s completed at 1:48 PM