Name: Sahil Saini

Reg No. 180905048

Roll NO 11C

```
!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
!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
```

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

%load_ext nvcc_plugin
```

Google Colabs Link

Question1

Write and execute a program in CUDA to add two vectors of length N to meet the following requirements using 3 different kernels

a) block size as N

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include<cuda.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
__global__ void blockSizeN(int * A, int * B, int *C, int N)
{
    int id=blockIdx.x;

    if(id<N)
```

```
      C[id]=A[id]+B[id];
}

int main()
{

    // Here N can be larger
    int N=10;
    int size=sizeof(int)*N;

    int * d_A, * d_B, * d_C;

    int A[N];
    int B[N];
    int C[N];

    for(int i=0;i<N;i++)
    {
        A[i]=B[i]=i;
    }


    cudaMalloc((void**)&d_A,size);
    cudaMalloc((void**)&d_B,size);
    cudaMalloc((void**)&d_C,size);

    cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,size,cudaMemcpyHostToDevice);

    blockSizeN<<<N,1>>>(d_A,d_B,d_C,N);

    cudaMemcpy(C,d_C,size,cudaMemcpyDeviceToHost);

    printf("Resultant array after adding A and B arrays:\n");

    for(int i=0;i<N;i++)
    {
        printf("%d ",C[i]);
    }

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

 return 0;

}
```

```
Resultant array after adding A and B arrays:
0 2 4 6 8 10 12 14 16 18
```

b) N threads within a block

```
 %%cu
#include <stdio.h>
#include <stdlib.h>
#include<cuda.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
__global__ void blockSizeN(int * A, int * B, int *C, int N)
{
    int id=threadIdx.x;

    if(id<N)
      C[id]=A[id]+B[id];
}

int main()
{
    // if N is increased the value is not computed

    int N=10;
    int size=sizeof(int)*N;

    int * d_A, * d_B, * d_C;

    int A[N];
    int B[N];
    int C[N];

    for(int i=0;i<N;i++)
    {
        A[i]=B[i]=i;
    }


    cudaMalloc((void**)&d_A,size);
    cudaMalloc((void**)&d_B,size);
    cudaMalloc((void**)&d_C,size);

    cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,size,cudaMemcpyHostToDevice);

    blockSizeN<<<1,N>>>(d_A,d_B,d_C,N);

    cudaMemcpy(C,d_C,size,cudaMemcpyDeviceToHost);

    printf("Resultant array after adding A and B arrays:\n");

    for(int i=0;i<N;i++)
    {
        printf("%d ",C[i]);
    }

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
```

```
 return 0;

}
```

```
Resultant array after adding A and B arrays:
0 2 4 6 8 10 12 14 16 18
```

c) Keep the number of threads per block as 256 (constant) and vary the number of blocks to handle N elements.

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

__global__ void blockSizeN(int * A, int * B, int *C, int N)
{
    int id=blockIdx.x*blockDim.x+threadIdx.x;

    if(id<N)
      C[id]=A[id]+B[id];
}

int main()
{
    // if N is increased the value is not computed

    int N=1000;
    int size=sizeof(int)*N;

    int * d_A, * d_B, * d_C;

    int A[N];
    int B[N];
    int C[N];

    for(int i=0;i<N;i++)
    {
        A[i]=B[i]=i;
    }


    cudaMalloc((void**)&d_A,size);
    cudaMalloc((void**)&d_B,size);
    cudaMalloc((void**)&d_C,size);

    cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,size,cudaMemcpyHostToDevice);

    blockSizeN<<<ceil(N/256.0),256>>>(d_A,d_B,d_C,N);
```

```
    cudaMemcpy(C,d_C,size,cudaMemcpyDeviceToHost);

    printf("Resultant array after adding A and B arrays:\n");

    for(int i=0;i<N;i++)
    {
        printf("%d ",C[i]);
    }

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

 return 0;

}
```

```
Resultant array after adding A and B arrays:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 5
```

Question2

Write and execute a CUDA program to read an array of N integer values. Sort the array in parallel using parallel selection sort and store the result in another array.

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


__global__ void sortKernel(int *A, int *B, int N)
{
    int id=blockDim.x*blockIdx.x+threadIdx.x;

    if(id>N)
     return ;

    int pos=0;

    for(int i=0;i<N;i++)
    {
        if(A[i]<A[id] || ( A[i]==A[id] &&i<id))
        {
            pos++;
        }
```

```
        }
        B[pos]=A[id];
    }

    int main()
    {
        int N=13;
        int size=sizeof(int)*N;

        int A[N];
        int B[N];

        for(int i=0;i<N;i++)
        {
            A[i]=N-i;
        }

        printf("Array before sorting:\n");

        for(int i=0;i<N;i++)
        {
            printf("%d ",A[i]);
        }
        printf("\n");



        int *d_A, *d_B;

        cudaMalloc((void**)&d_A,size);
        cudaMalloc((void**)&d_B,size);

        //Only that mem which kernel needs to read
        cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);

        sortKernel<<<ceil(N/5.0),5>>>(d_A,d_B,N);

        cudaMemcpy(B,d_B,size,cudaMemcpyDeviceToHost);

        printf("Array after sorting:\n");

        for(int i=0;i<N;i++)
        {
            printf("%d ",B[i]);
        }

        return 0;
    }
```

```
 Array before sorting:
 13 12 11 10 9 8 7 6 5 4 3 2 1
 Array after sorting:
 1 2 3 4 5 6 7 8 9 10 11 12 13
```

Question3

Write a execute a CUDA program to read an integer array of size N. Sort this array using odd-even transposition sorting. Use 2 kernels.

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


__global__ void oddEven(int *A, int N)
{
    int id=blockDim.x*blockIdx.x+threadIdx.x;
    if(id>N)
      return ;

   if(id%2==1 &&id+1<N)
   {
       if(A[id]>A[id+1])
       {
           int temp=A[id];
           A[id]=A[id+1];
           A[id+1]=temp;
       }
   }
}

__global__ void evenOdd(int *A, int N)
{
    int id=blockDim.x*blockIdx.x+threadIdx.x;

     if(id>N)
      return ;

   if(id%2==0 &&id+1<N)
   {
       if(A[id]>A[id+1])
      {
           int temp=A[id];
           A[id]=A[id+1];
           A[id+1]=temp;
      }
   }
}

int main()
{
    int N=51;
```

```
    int N=51;
    int size=sizeof(int)*N;

    int A[N];

    for(int i=0;i<N;i++)
    {
        A[i]=N-i;
    }

    printf("Array before sorting:\n");
    for(int i=0;i<N;i++)
    {
        printf("%d ",A[i]);
    }
    printf("\n");



    int *d_A;

    cudaMalloc((void**)&d_A,size);

    cudaMemcpy(d_A,A,size,cudaMemcpyHostToDevice);

    for(int i=0;i<=ceil(N/2);i++)
    {
        evenOdd<<<ceil(N/7.0),7>>>(d_A,N);
        oddEven<<<ceil(N/7.0),7>>>(d_A,N);
    }

    cudaMemcpy(A,d_A,size,cudaMemcpyDeviceToHost);

    printf("Array after sorting:\n");

    for(int i=0;i<N;i++)
    {
        printf("%d ",A[i]);
    }

    return 0;
}
```

```
 Array before sorting:
 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26
 Array after sorting:
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
```

Ends

✓ 2s completed at 23:14