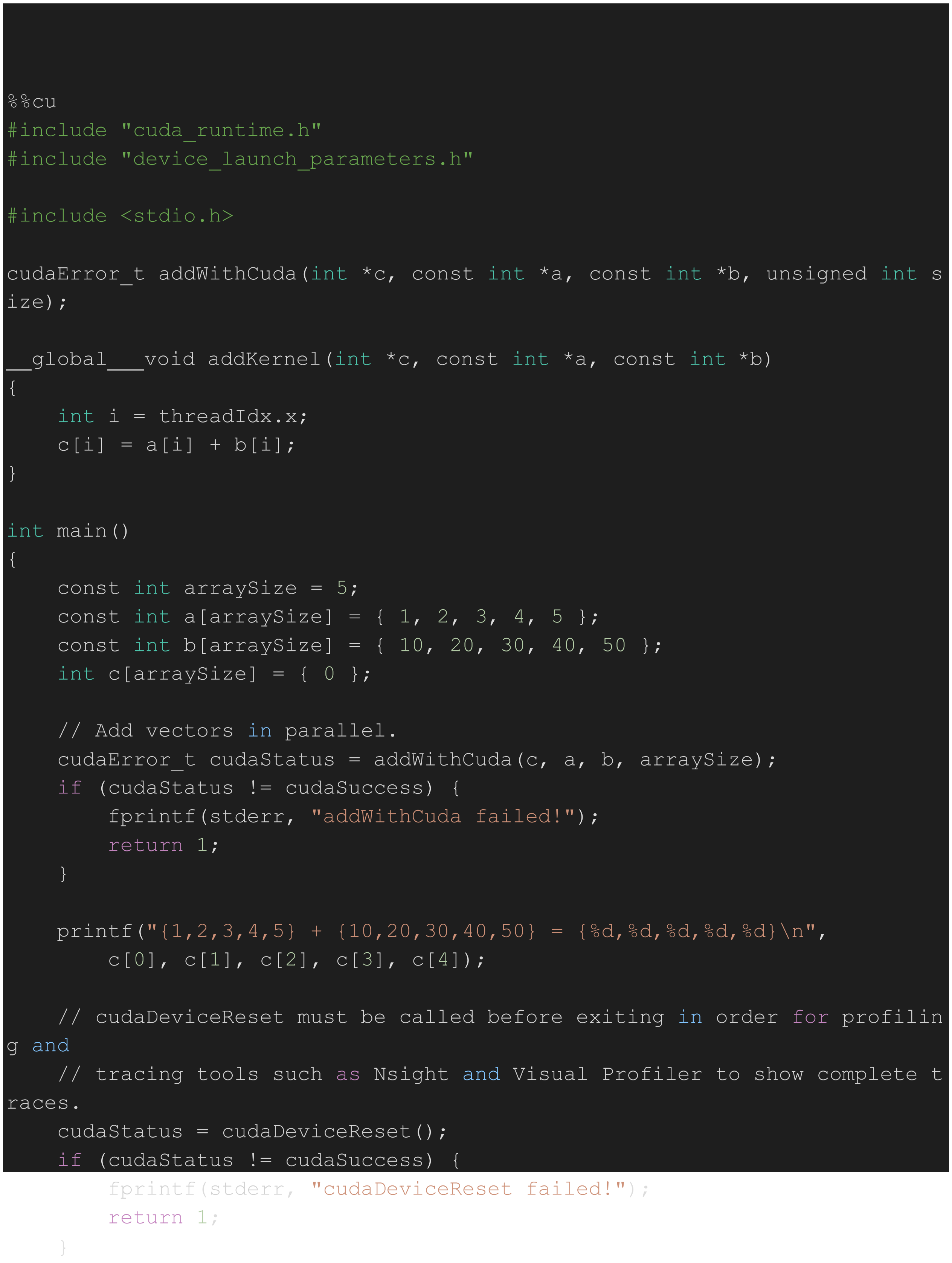
Name : Usman Manzoor

Section : 6B

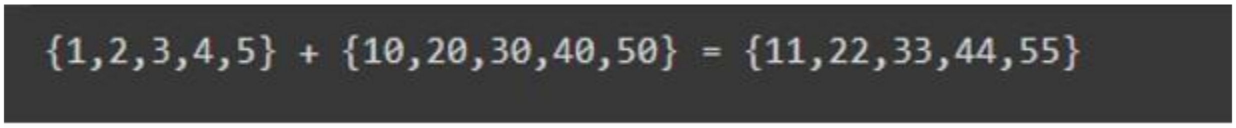
Roll No: p19-0068

Assignement: CUDA

Following the step in manual we get to here



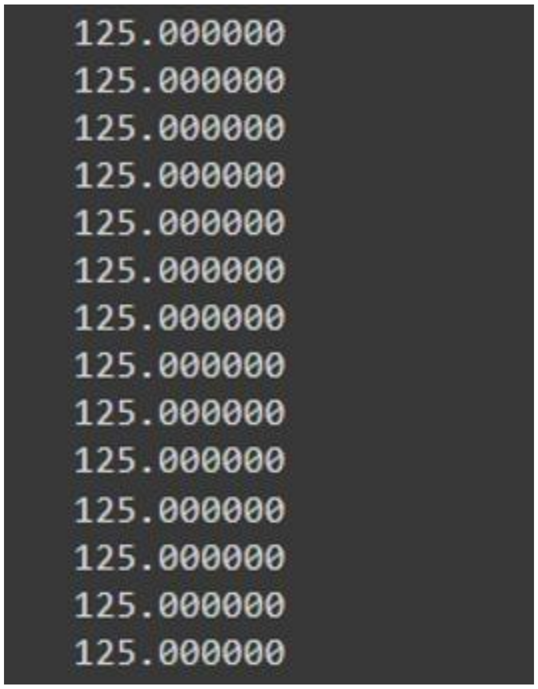
```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int s
ize);

__global___void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    const int arraySize = 5;
    const int a[arraySize] = { 1, 2, 3, 4, 5 };
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",
        c[0], c[1], c[2], c[3], c[4]);

    // cudaDeviceReset must be called before exiting in order for profilin
g and
    // tracing tools such as Nsight and Visual Profiler to show complete t
races.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }
    return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)    .
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    // Copy input vectors from host memory to GPU buffers.
    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostTo
Device);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, size>>>(dev_c, dev_a, dev_b);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorStrin
g(cudaStatus));
        goto Error;
    }

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d afte
r launching addKernel!\n", cudaStatus);
        goto Error;
    }

    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDevice
ToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}
```



```
{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
```

Matrix Multiplication

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

int main(void) {
    void MatrixMultiplication(float *, float *, float *, int);
    const int Width = 5;
    float M[Width*Width], N[Width*Width], P[Width*Width];
    for(int i = 0; i < (Width*Width) ; i++) {
        M[i] = 5;
        N[i] = 5;
        P[i] = 0;
    }
    MatrixMultiplication(M, N, P, Width);
    for(int i = 0; i < (Width*Width) ; i++) {
        printf("%f \n", P[i]);
    }
    int quit;
    scanf("%d",&quit);
    return 0;
}

//Matrix multiplication kernel - thread specification
__global___void MatrixMulKernel(float *Md, float *Nd, float *Pd, int Width
) {
    //2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    //Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for(int k = 0; k < Width ; ++k) {
        float Mdelement = Md[ty*Width + k];
        float Ndelement = Nd[k*Width + tx];
        Pvalue += (Mdelement*Ndelement);
    }

    Pd[ty*Width + tx] = Pvalue;
}

void MatrixMultiplication(float *M, float *N, float *P, int Width) {
    int size = Width*Width*sizeof(float);
    float *Md, *Nd, *Pd;

    //Transfer M and N to device memory
    cudaMalloc((void**)&Md, size);
    cudaMemcpy(Md,M,size,cudaMemcpyHostToDevice);
    cudaMalloc((void**)&Nd, size);
    cudaMemcpy(Nd,N,size,cudaMemcpyHostToDevice);

    //Allocate P on the device
    cudaMalloc((void**)&Pd,size);

    //Setup the execution configuration
    dim3 dimBlock(Width,Width);
    dim3 dimGrid(1,1);

    //Launch the device computation threads!
    MatrixMulKernel<<<dimGrid,dimBlock>>>(Md,Nd,Pd,Width);

    //Transfer P from device to host
    cudaMemcpy(P,Pd,size,cudaMemcpyDeviceToHost);

    //Free device matrices
    cudaFree(Md);
    cudaFree(Nd);
    cudaFree(Pd);
}
```


```
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
125.000000
```