Leibniz.cu

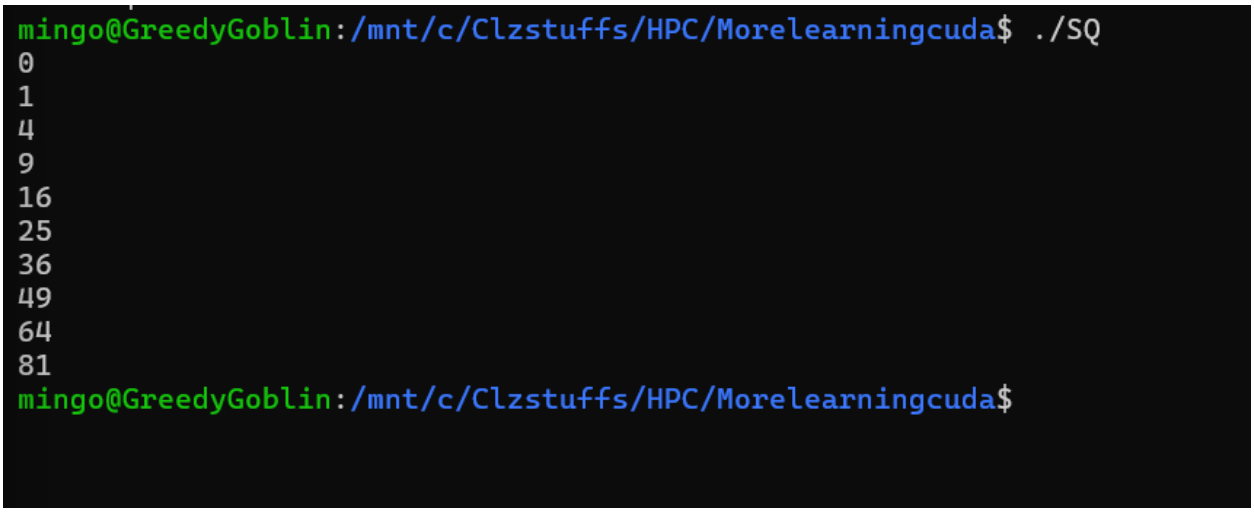
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ ./Leibniz
Pi = 3.141393
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ cat Leibniz.cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//pi/4 = sigma  ((-1)^k) / ( (2*k) + 1)

//k     0 -> 999

//[ , , , , , , , , , , , , ........ 999]

__global__ void Leibniz(double * leibnizOutput){

        int k = blockDim.x * blockIdx.x + threadIdx.x;
        //printf("%d\n", k);

        leibnizOutput[k] = (pow(-1,k)) / (2*k+1);
}

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

        //1000 threads
        int blocksX = 10;
        int threadsX = 500;

        //double because we need decimal places
        int memorySize = blocksX * threadsX * sizeof(double);

        double * gpuList;
        double * cpuList = (double*) malloc (memorySize);

        cudaMalloc( (void**) &gpuList, memorySize);

        Leibniz<<< blocksX , threadsX >>>(gpuList);
        cudaThreadSynchronize();

        cudaMemcpy(cpuList, gpuList, memorySize, cudaMemcpyDeviceToHost);

        double finalSum = 0;

        for (int i = 0; i < blocksX * threadsX; i++){
                finalSum = finalSum + cpuList[i];
        }

        printf("Pi = %lf\n", finalSum * 4);

}
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$
```

This program finds the value of Pi by using the GPU to solve the Leibniz formula in parallel. It works by breaking the mathematical series into 5,000 individual fractions and assigning each one to a specific GPU thread. Instead of a CPU calculating these terms one by one, the GPU uses 10

blocks of 500 threads to calculate every fraction simultaneously and store them in a list. Once the GPU is finished, the program copies that list back to the CPU, which adds all the pieces together and multiplies the total by 4 to get the final result. By splitting the heavy math into thousands of tiny, simultaneous tasks, the code demonstrates how a GPU can accelerate scientific calculations.

SQ.cu

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ cat SQ.cu
//Program to square 100 numbers using CUDA

//Host is the CPU
//Device is the GPU

#include <stdio.h>

__global__ void square(int * device_input, int * device_output){

int threadId = threadIdx.x;
//printf("%d\n", threadId);
int  num = device_input[threadId];
device_output[threadId] = num * num;
//printf("%f\n", device_output[threadId]);
}

int main(int argc, char ** argv){
    int arraySize = 10; //size of list i want to create to square
    int arraySizeMemory = arraySize * sizeof(int); //memory amount

    //CPU - HOST Variables
    int hostNumbers[10];
    int hostSQNumbers[10];

    //creating input values on the CPU
    for (int i = 0; i<arraySize; i++){
        hostNumbers[i] =i;
    }

    //create GPU variables
    int * deviceNumbers;
    int * deviceSQNumbers;
    //hostNumbers = (float *) malloc(arraySizeMemory);
    //allocate memory on the GPU using cudaMallo
    //cudaMalloc ( variableName, memorySize)
    cudaMalloc( (void**) &deviceNumbers, arraySizeMemory);
    cudaMalloc( (void**) &deviceSQNumbers, arraySizeMemory);

    //transfer memory from CPU to GPU - cudaMemcpy
    //cudaMemcpy(destinationVar, originVar, memorySize, transferDirection)
    cudaMemcpy(deviceNumbers, hostNumbers, arraySizeMemory, cudaMemcpyHostToDevice);

    square<<<1 , 10>>>(deviceNumbers, deviceSQNumbers);
    cudaThreadSynchronize();

    cudaMemcpy(hostSQNumbers, deviceSQNumbers, arraySizeMemory, cudaMemcpyDeviceToHost);

    //free(hostNumbers);
        //free(hostSQNumbers);

        cudaFree(deviceNumbers);
        cudaFree(deviceSQNumbers);
        for(int i = 0; i<arraySize; i++){
       printf("%d\n", hostSQNumbers[i]);
    }

}
```



The process begins by creating a list of 10 numbers (0 through 9) on the **Host** Because the GPU has its own separate memory system, the code uses cudaMalloc to reserve space on the **Device** for both the starting numbers and the final results. It then copies the data from the CPU to the GPU using cudaMemcpy. The "magic" happens when the program launches a **Kernel** with 10 parallel threads; instead of a single loop running 10 times in a row, the GPU starts 10 tiny workers at the exact same time. Each worker identifies itself by a unique ID threadIdx.x, grabs its specific number, squares it, and saves it. Finally, the program copies the finished results back to the CPU to be printed and "cleans up" by freeing the used memory. In short, it transforms a sequential task into a parallel one, showcasing how GPUs can handle many identical calculations simultaneously.

PasswordCrack

```
a u 5 9 = ccbxrf345
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ cat PasswordCrack.cu
#include <stdio.h>
#include <stdlib.h>

//__global__ --> GPU function which can be launched by many blocks and threads
//__device__ --> GPU function or variables
//__host__ --> CPU function or variables

__device__ char* CudaCrypt(char* rawPassword){

        char * newPassword = (char *) malloc(sizeof(char) * 11);
        //ab12-->cdqwer35

        newPassword[0] = rawPassword[0] + 2;
        newPassword[1] = rawPassword[0] - 2;
        newPassword[2] = rawPassword[0] + 1;
        newPassword[3] = rawPassword[1] + 3;
        newPassword[4] = rawPassword[1] - 3;
        newPassword[5] = rawPassword[1] - 1;
        newPassword[6] = rawPassword[2] + 2;
        newPassword[7] = rawPassword[2] - 2;
        newPassword[8] = rawPassword[3] + 4;
        newPassword[9] = rawPassword[3] - 4;
        newPassword[10] = '\0';

//ab12
        for(int i =0; i<10; i++){
                if(i >= 0 && i < 6){ //checking all lower case letter limits
                        if(newPassword[i] > 122){
                                newPassword[i] = (newPassword[i] - 122) + 97;
                        }else if(newPassword[i] < 97){
                                newPassword[i] = (97 - newPassword[i]) + 97;
                        }
                }else{ //checking number section
                        if(newPassword[i] > 57){
                                newPassword[i] = (newPassword[i] - 57) + 48;
                        }else if(newPassword[i] < 48){
                                newPassword[i] = (48 - newPassword[i]) + 48;
                        }
                }
        }
        return newPassword;
}

__global__ void crack(char * alphabet, char * numbers){

char genRawPass[4];

genRawPass[0] = alphabet[blockIdx.x];
genRawPass[1] = alphabet[blockIdx.y];

genRawPass[2] = numbers[threadIdx.x];
genRawPass[3] = numbers[threadIdx.y];

//firstLetter - 'a' - 'z' (26 characters)
//secondLetter - 'a' - 'z' (26 characters)
//firstNum - '0' - '9' (10 characters)
//secondNum - '0' - '9' (10 characters)

//Idx --> gives current index of the block or thread

printf("%c %c %c %c = %s\n", genRawPass[0],genRawPass[1],genRawPass[2],genRawPass[3], CudaCrypt(genRawPass));

}

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

char cpuAlphabet[26] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
char cpuNumbers[10] = {'0','1','2','3','4','5','6','7','8','9'};

char * gpuAlphabet;
cudaMalloc( (void**) &gpuAlphabet, sizeof(char) * 26);
cudaMemcpy(gpuAlphabet, cpuAlphabet, sizeof(char) * 26, cudaMemcpyHostToDevice);

char * gpuNumbers;
cudaMalloc( (void**) &gpuNumbers, sizeof(char) * 26);
cudaMemcpy(gpuNumbers, cpuNumbers, sizeof(char) * 26, cudaMemcpyHostToDevice);

crack<<< dim3(26,26,1), dim3(10,10,1) >>>( gpuAlphabet, gpuNumbers );
cudaThreadSynchronize();
return 0;
}
```

```
a u 8 2 = ccbxrt1662
a u 9 2 = ccbxrt2762
a u 0 3 = ccbxrt2271
a u 1 3 = ccbxrt3171
a u 4 6 = ccbxrt6212
a u 5 6 = ccbxrt7312
a u 6 6 = ccbxrt8412
a u 7 6 = ccbxrt9512
a u 8 6 = ccbxrt1612
a u 9 6 = ccbxrt2712
a u 0 7 = ccbxrt2223
a u 1 7 = ccbxrt3123
a u 2 7 = ccbxrt4023
a u 3 7 = ccbxrt5123
a u 4 7 = ccbxrt6223
a u 5 7 = ccbxrt7323
a u 6 7 = ccbxrt8423
a u 7 7 = ccbxrt9523
a u 8 7 = ccbxrt1623
a u 9 7 = ccbxrt2723
a u 0 8 = ccbxrt2234
a u 1 8 = ccbxrt3134
a u 2 8 = ccbxrt4034
a u 3 8 = ccbxrt5134
a u 4 8 = ccbxrt6234
a u 5 8 = ccbxrt7334
a u 6 8 = ccbxrt8434
a u 7 8 = ccbxrt9534
a u 8 8 = ccbxrt1634
a u 9 8 = ccbxrt2734
a u 0 9 = ccbxrt2245
a u 1 9 = ccbxrt3145
a u 2 9 = ccbxrt4045
a u 3 9 = ccbxrt5145
a u 4 9 = ccbxrt6245
a u 5 9 = ccbxrt7345
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$
```

MatrixAdd

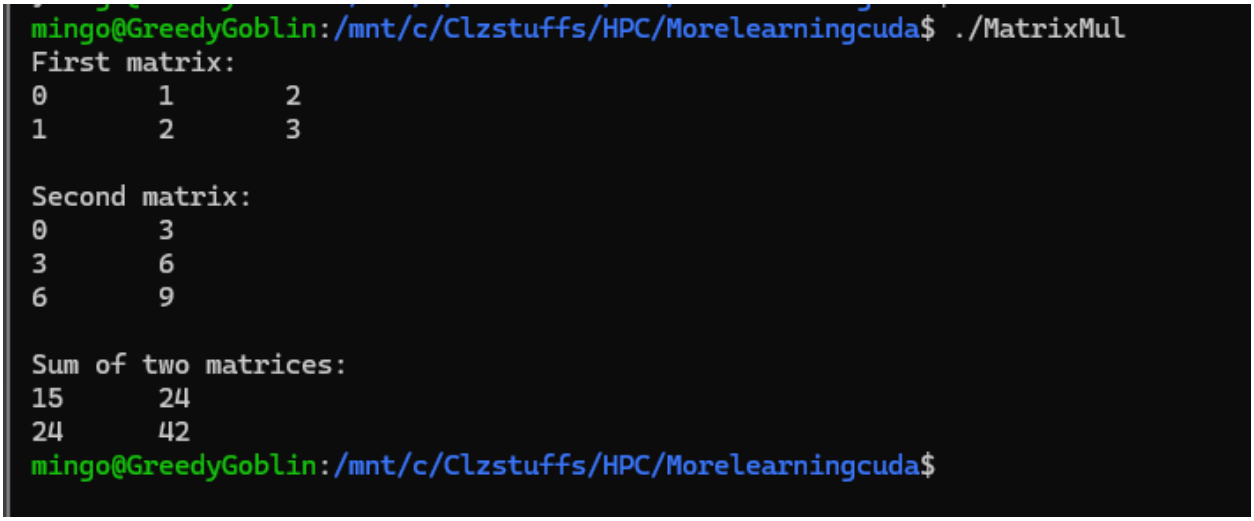
```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ cat MatrixAdd.cu
#include<stdio.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<cuda_runtime_api.h>
#define i 4
#define j 3
//kernel function decleatation
__global__ void matadd(int *x,int *y, int *z)
{
//expression for matrix addition using gridDim and blockIndex
    int index=gridDim.x * blockIdx.y +blockIdx.x;
    z[index]=x[index]+y[index];
}
int main()
{
    int h_a[i][j];
    int h_b[i][j];
    int h_c[i][j];

    int *d_a,*d_b,*d_c;
    int r,c;  //r=row and c=column
//first matrix
printf("\nFirst matrix:\n");
    for(r=0;r<i; r++)
    {
        for(c=0;c<j;c++)
            {
        h_a[r][c]=r+c;
        printf("%d\t", h_a[r][c]);
            }
        printf("\n");
    }
printf("\nSecond matrix:\n");
     for(r=0;r<i; r++)
    {
        for(c=0;c<j;c++)
            {
        h_b[r][c]=i*j+r;
        printf("%d\t", h_b[r][c]);
            }
        printf("\n");
    }
   //memory allocation on device
    cudaMalloc(&d_a,i*j*sizeof(int));
    cudaMalloc(&d_b,i*j*sizeof(int));
    cudaMalloc(&d_c,i*j*sizeof(int));
//copy matrices from host to device
 cudaMemcpy(d_a,h_a,i*j*sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(d_b,h_b,i*j*sizeof(int),cudaMemcpyHostToDevice);
//kernel launch
 matadd<<<dim3(j,i),1>>>(d_a,d_b,d_c);
//copy result matrix from defice to host
 cudaMemcpy(h_c,d_c,i*j*sizeof(int),cudaMemcpyDeviceToHost);
//display result
    printf("\nSum of two matrices:\n");
    for(r=0;r<i;r++)
    {
        for(c=0;c<j;c++)
        {
              printf("%d\t",h_c[r][c]);
        }
        printf("\n");
    }
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ nvcc MatrixAdd.cu -o MatrixAdd
```

This program performs matrix addition by treating each cell of the grid as an independent task for the GPU to solve. After filling two 4x3 matrices with numbers on the CPU, the code allocates memory on the GPU and copies the data over. To add them, it launches a grid where each thread block represents a single position in the matrix. Inside the matadd kernel, each thread calculates a unique index based on its coordinates in the grid, identifying exactly which pair of numbers it is responsible for. This allows the GPU to calculate all 12 additions at the exact same moment rather than looping through rows and columns one by one. Finally, the resulting matrix is copied back to the CPU and printed to the screen, showing the element-wise sum of the two original grids.

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ nvcc MatrixAdd.cu -o MatrixAdd
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ ./MatrixAdd

First matrix:
0       1       2
1       2       3
2       3       4
3       4       5

Second matrix:
12      12      12
13      13      13
14      14      14
15      15      15

Sum of two matrices:
12      13      14
14      15      16
16      17      18
18      19      20
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$
```

MatrixMul

```
mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$ cat MatrixMul.cu
#include<stdio.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<cuda_runtime_api.h>
#define i 2
#define k 3
#define j 2
//kernel function declearation
__global__ void MatrixMulFunc(int *x,int *y, int *z)
{
    int m=blockIdx.x;
    int n=blockIdx.y;
//matrix multiplication expression
   z[j*n+m]=0;
    for(int p=0;p<k;p++){
         z[j*n+m]+=x[k*n+p]*y[j*p+m];
        }
   }
int main()
{
    int h_a[i][k];
    int h_b[k][j];
    int h_c[i][j];

    int *d_a,*d_b,*d_c;
    int r,c;   //r=row and c=column
printf("First matrix:\n");
    for(r=0;r<i; r++)
    {
        for(c=0;c<k;c++)
            {
        h_a[r][c]=r+c;
        printf("%d\t", h_a[r][c]);
            }
        printf("\n");
    }
printf("\nSecond matrix:\n");
      for(r=0;r<k; r++)
    {
        for(c=0;c<j;c++)
            {
        h_b[r][c]=(r+c)*k;
        printf("%d\t", h_b[r][c]);
            }
        printf("\n");
    }
//allocate memory to store matrices h_a[i][k],h_b[k][j],h_c[i][j] in device
    cudaMalloc(&d_a,i*k*sizeof(int));
    cudaMalloc(&d_b,k*j*sizeof(int));
    cudaMalloc(&d_c,i*j*sizeof(int));
//copy matrices h_a[i][k],h_b[k][j] from host to device
 cudaMemcpy(d_a,h_a,i*k*sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(d_b,h_b,k*j*sizeof(int),cudaMemcpyHostToDevice);
//kernel function invocation usinf 2D block having one thread
    MatrixMulFunc<<<dim3(j,k),1>>>(d_a,d_b,d_c);
//copy result matrix from device to host
 cudaMemcpy(h_c,d_c,i*j*sizeof(int),cudaMemcpyDeviceToHost);
//to display result matirx
    printf("\nSum of two matrices:\n");
    for(r=0;r<i;r++)
    {
        for(c=0;c<j;c++)
        {
             printf("%d\t",h_c[r][c]);
        }
        printf("\n");
    }
//de-allocate allocated memory of device
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}mingo@GreedyGoblin:/mnt/c/Clzstuffs/HPC/Morelearningcuda$
```



This program performs matrix multiplication by assigning the calculation of each individual element in the resulting matrix to a specific GPU thread. After initializing two matrices on the CPU one size 2x3 and another 3x2 the code transfers them to the GPU's memory. In matrix multiplication, every spot in the answer is found by multiplying a row from the first matrix by a column from the second. The GPU accelerates this by launching a grid of threads where each thread focuses on just one cell of the final 2x2 result. Inside the MatrixMulFunc kernel, each thread uses its coordinates (blockIdx.x and blockIdx.y) to identify which row and column it needs to process. It then runs a short loop to calculate the "dot product" for its assigned cell. Once all threads finish their specific math simultaneously, the completed result matrix is copied back to the CPU to be printed.