**Code 1 : Adding two arrays**

**Key Function :**

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.

**Functioning :**

1. Creating 2 arrays and a result array will be initialised with zero.
2. All these arrays are then passed as parameter to a function , in that function first the input arrays are copies into the GPU memory.
3. We launched a kernel on the GPU with one thread for each element.
4. In the GPU the thread id is fetched and then using that id the corresponding location in the result array is modified.
5. Then we print the final array

**Code :**

#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 size);

\_\_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 profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

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), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

// Launch a kernel on the GPU with one thread for each element.

addKernel<<<size, 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", cudaGetErrorString(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 after launching addKernel!\n", cudaStatus);

goto Error;

}

// Copy output vector from GPU buffer to host memory.

cudaStatus = cudaMemcpy(c, dev\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

Error:

cudaFree(dev\_c);

cudaFree(dev\_a);

cudaFree(dev\_b);

return cudaStatus;

}

**Output:**

**Code 2: Adding two matrix**

**Key Function :**

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.
4. dim3 : this is the function which is used to generate threads in multi-dimension
   1. eg dim3 X(N,N)

**Functioning :**

1. Creating 2 matrix and a result matrix will be initialised with zero.
2. All these matrices will be copied into the GPU memory
3. We launched a kernel on the GPU with threads in two dimensions x and y.
4. In the GPU the thread id is fetched for x component and y component and then using that id the corresponding location in the result matrix is modified.
5. Then we print the final matrix

**Code:**

#include "cuda\_runtime.h"

//matrix Addition

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <cuda.h>

#include <cuda\_runtime.h>

//#include "kernel.h"

#define N 3

//int fin[N][N] = { 0 };

\_\_global\_\_ void MatAdd(int A[][N], int B[][N],int C[][N]) {

int i = threadIdx.x;

int j = threadIdx.y;

C[i][j] = A[i][j] + B[i][j];

/\*

#if \_\_CUDA\_ARCH\_\_ >= 200

atomicAdd(&C[i][j], A[i][j] + B[i][j]);

#endif

\*/

}

int main() {

int A[N][N] = { { 1,2,3 },{ 3,4,2 },{1,2,3 } };

int B[N][N] = { { 5,6,1 },{ 7,8,2 },{1,2,3} };

int C[N][N] = { { 0,0,0 },{ 0,0,0 },{0,0,0} };

int(\*pA)[N], (\*pB)[N], (\*pC)[N],(\*pf)[N];

cudaMalloc((void\*\*)&pA, (N\*N) \* sizeof(int));

cudaMalloc((void\*\*)&pB, (N\*N) \* sizeof(int));

cudaMalloc((void\*\*)&pC, (N\*N) \* sizeof(int));

cudaMemcpy(pA, A, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(pB, B, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(pC, C, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

//cudaMemcpy(pf, fin, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

int numBlocks = 1;

dim3 threadsPerBlock(N, N);

MatAdd << <numBlocks, threadsPerBlock >>>(pA, pB, pC);

cudaMemcpy(C, pC, (N\*N) \* sizeof(int), cudaMemcpyDeviceToHost);

int i, j; printf("C = \n");

for (i = 0; i<N; i++) {

for (j = 0; j<N; j++) {

printf("%d ", C[i][j]);

}

printf("\n");

}

cudaFree(pA);

cudaFree(pB);

cudaFree(pC);

printf("\n");

return 0;

}

**Output :**

**Code 3: Multiplying 2 matrix**

**Key Function :**

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.
4. dim3 : this is the function which is used to generate threads in multi-dimension
   1. eg dim3 X(N,N)

**Functioning :**

1. Creating 2 matrix and a result matrix will be initialised with zero.
2. All these matrices will be copied into the GPU memory
3. We launched a kernel on the GPU with threads in two dimensions x and y.
4. Here is a catch , we have followed a different mechanism of multiplying as at a time we will be able to get the id of a particular element in a matrix.
5. So to allow multiplication to work fine , what we did is :
   1. We summed the value of that block of the matrix to all the indexes in the final result matrix.
   2. We used atomic add for this so that add all the threads will update the matrix value one at a time.
6. Then we print the final matrix.

**Code:**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <device\_functions.h>

#include <stdio.h>

#include <stdlib.h>

#include <cuda.h>

#include <cuda\_runtime.h>

#define N 3

\_\_global\_\_ void MatMul(int A[][N], int B[][N], int C[][N]) {

int i = threadIdx.x;

int j = threadIdx.y;

int temp;

for (int j1 = 0; j1 < N; j1++)

{

temp = A[i][j] \* B[j][j1];

#if \_\_CUDA\_ARCH\_\_ >= 200

atomicAdd(&C[i][j1], temp);

#endif

}

}

int main() {

int A[N][N] = { { 1,2,3 },{ 4,5,6 },{ 7,8,9 } };

int B[N][N] = { { 1,2,3 },{ 4,5,6 },{ 7,8,9 } };

int C[N][N] = { { 0,0,0 },{ 0,0,0 },{ 0,0,0 } };

int(\*pA)[N], (\*pB)[N], (\*pC)[N], (\*pf)[N];

cudaMalloc((void\*\*)&pA, (N\*N) \* sizeof(int));

cudaMalloc((void\*\*)&pB, (N\*N) \* sizeof(int));

cudaMalloc((void\*\*)&pC, (N\*N) \* sizeof(int));

cudaMemcpy(pA, A, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(pB, B, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(pC, C, (N\*N) \* sizeof(int), cudaMemcpyHostToDevice);

int numBlocks = 1;

dim3 threadsPerBlock(N,N);

MatMul << <numBlocks, threadsPerBlock >> >(pA, pB, pC);

cudaMemcpy(C, pC, (N\*N) \* sizeof(int), cudaMemcpyDeviceToHost);

int i, j; printf("\nC = \n");

for (i = 0; i<N; i++) {

for (j = 0; j<N; j++) {

printf("%d ", C[i][j]);

}

printf("\n");

}

cudaFree(pA);

cudaFree(pB);

cudaFree(pC);

printf("\n");

return 0;

}

**Output:**

**Code 4: hist**

**Key Function :**

1. cudaMemcpy : This function helps in copying the data from host to device and vice-versa
2. cudaMalloc : This function allocates memory in the GPU
3. cudaFree : This function will free the memory allocated.

**Functioning:**

1. Creating one arrays and a result array will be initialised with zero.
2. All these arrays are then passed as parameter to a function , in that function first the input arrays are copies into the GPU memory.
3. We launched a kernel on the GPU with one thread for each element.
4. In the GPU the thread id is fetched and then using that id the corresponding location in the result array is modified in the following way :
   * 1. We will update the index equal to the value of the array element by ‘1’.
     2. We will use atomic add for this purpose as we want one thread to update the value of the result array at a time so that the final array obtained is consistent.
5. Then we print the final array.

**Code:**

//histogram processing

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

cudaError\_t findFreqWithCuda(int \*c, unsigned int fsize, const int \*b, unsigned int size);

\_\_global\_\_ void addKernel(int \*c, const int \*b)

{

int i = threadIdx.x;

#if \_\_CUDA\_ARCH\_\_ >= 200

atomicAdd(&c[b[i]],1);

#endif

}

int main()

{

const int arraySize = 11;

const int fsize = 10005;//maximum value of the number in the array can be at max 10005

const int a[arraySize] = { 1, 2, 3, 4, 5,1,1,1,2,3,6 };

int maxx = INT\_MIN;

for (int i = 0; i < arraySize; i++)

{

//maxx = max(maxx, arr[i]);

if (maxx < a[i])

maxx=a[i];

}

int c[fsize] = { 0 };

// Add vectors in parallel.

cudaError\_t cudaStatus = findFreqWithCuda(c,fsize,a,arraySize);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "findFreqWithCuda failed!");

return 1;

}

for (int i = 0; i <= maxx; i++)

{

if (c[i] != 0)

{

printf("%d occurs %d times\n", i, c[i]);

}

}

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

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 findFreqWithCuda(int \*c, unsigned int fsize, 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, fsize \* 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\_b, b, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

// Launch a kernel on the GPU with one thread for each element.

//dim3 threadsPerBlock(11,11,11);

addKernel<<<1,size>>>(dev\_c,dev\_b);

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(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 after launching addKernel!\n", cudaStatus);

goto Error;

}

// Copy output vector from GPU buffer to host memory.

cudaStatus = cudaMemcpy(c, dev\_c, fsize \* sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

Error:

cudaFree(dev\_c);

cudaFree(dev\_b);

return cudaStatus;

}

**Output:**