درس برنامه نویسی چند هسته ای

آزمایش ششم – ضرب ماتریس

آرش حریرپوش
9731505

آرش حریرپوش
9731505

❖ گام اول

برای بررسی صحت عملکرد این کد می توان کد را به صورت زیر تغییر داد، تا خروجی حاصل از ضرب ماتریس ها را مشاهده کرد:

```
// System includes
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>
#include <device_launch_parameters.h>


/**
* Matrix multiplication (CUDA Kernel) on the device: C = A * B
*/
#define TILE_WIDTH 16
__global__ void
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int k;
	int row = threadIdx.y, col = threadIdx.x;
	float sum = 0.0f;
	for (k = 0; k < n; ++k) {
		sum += A[row * n + k] * B[k * n + col];
	}
	C [row * n + col] = sum;
}

void constantInit(float *data, int size, float val)
{
	for (int i = 0; i < size; ++i)
	{
		data[i] = val;
	}
}

// Prints a Matrices to the stdout.
void printMat(float * v, int n) {
	int i;
	printf("[-] Vector elements: \n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++)
			printf("%f    ", v[i*n + j]);
		printf("\n");
	}
	printf("\b\b \n");
}

/**
* Run a simple test of matrix multiplication using CUDA
*/
int matrixMultiply(int argc, char **argv, int n)
```

```
{
	// Allocate host memory for matrices A and B
	unsigned int size_A = n * n;
	unsigned int mem_size_A = sizeof(float)* size_A;
	float *h_A = (float *)malloc(mem_size_A);
	unsigned int size_B = n * n;
	unsigned int mem_size_B = sizeof(float)* size_B;
	float *h_B = (float *)malloc(mem_size_B);

	// Initialize host memory
	const float valB = 0.01f;
	constantInit(h_A, size_A, 1.0f);
	constantInit(h_B, size_B, valB);

	// Allocate device memory
	float *d_A, *d_B, *d_C;

	// Allocate host matrix C
	unsigned int mem_size_C = n * n * sizeof(float);
	float *h_C = (float *)malloc(mem_size_C);

	if (h_C == NULL)
	{
		fprintf(stderr, "Failed to allocate host matrix C!\n");
		exit(EXIT_FAILURE);
	}

	cudaError_t error;

	error = cudaMalloc((void **)&d_A, mem_size_A);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_B, mem_size_B);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_B returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_C, mem_size_C);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_C returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// copy host memory to device
	error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
```

```
	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_A,h_A) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_B,h_B) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// Setup execution parameters
	dim3 threads(32, 32,1);
	dim3 grid(1,1,1);

	// Create and start timer
	printf("Computing result using CUDA Kernel...\n");

	// Allocate CUDA events that we'll use for timing
	cudaEvent_t start;
	error = cudaEventCreate(&start);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	cudaEvent_t stop;
	error = cudaEventCreate(&stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the start event
	error = cudaEventRecord(start, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Execute the kernel
	matrixMulCUDA << < grid, threads >> > (d_C, d_A, d_B, n);
```

```
	error = cudaGetLastError();
	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to launch kernel!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the stop event
	error = cudaEventRecord(stop, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Wait for the stop event to complete
	error = cudaEventSynchronize(stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to synchronize on the stop event (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	float msecTotal = 0.0f;
	error = cudaEventElapsedTime(&msecTotal, start, stop);

	printf("Elapsed time in msec = %f\n", msecTotal);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to get time elapsed between events (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Copy result from device to host
	error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (h_C,d_C) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	printMat(h_A, n);
	printMat(h_B, n);
	printMat(h_C, n);

	// Clean up memory
	free(h_A);
	free(h_B);
	free(h_C);
	cudaFree(d_A);
```

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

	return EXIT_SUCCESS;

}


/**
* Program main
*/
int main(int argc, char **argv)
{
	printf("[Matrix Multiply Using CUDA] - Starting...\n");

	// By default, we use device 0
	int devID = 0;
	cudaSetDevice(devID);

	cudaError_t error;
	cudaDeviceProp deviceProp;
	error = cudaGetDevice(&devID);

	if (error != cudaSuccess)
	{
		printf("cudaGetDevice returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}

	error = cudaGetDeviceProperties(&deviceProp, devID);

	if (deviceProp.computeMode == cudaComputeModeProhibited)
	{
		fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no
threads can use ::cudaSetDevice().\n");
		exit(EXIT_SUCCESS);
	}

	if (error != cudaSuccess)
	{
		printf("cudaGetDeviceProperties returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}
	else
	{
		printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);
	}

	// Size of square matrices
	size_t n = 0;
	printf("[-] N = ");
	scanf("%u", &n);

	printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", n, n, n, n);

	int matrix_result = matrixMultiply(argc, argv, n);
```

```
        exit(matrix_result);
}
```

این کد حاصل ضرب یک ماتریس که تمام المان های آن یک می باشد با یک ماتریس که تمام المان های آن 0.01 می باشد است، بنابراین نتیجه این حاصل ضرب باید یک ماتریس باشد که تمامی المان های آن برابر n * 0.01 که n برابر ابعاد ماتریس می باشد است، بنابراین زمانی که ابعاد این ماتریس ها برابر 32 * 32 (یعنی n=32) باشد، ماتریس خروجی یک ماتریس 32 *32 می باشد که تمام المان های آن برابر 0.32 می باشد؛ با اجرای کد بالا به ازای n=32 خروجی زیر بدست می آید:

```
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000

0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000

0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000

0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000

0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000

0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.320000        0.32000
```

تصویر بالا شامل بخشی از خروجی می باشد، که به دلیل طولانی بودن خروجی بقیه آن ها قرار داده نشده است؛ که آن مقادیر نیز مشابه همین مقادیر می باشد، حال برای اطمینان بیشتر خروجی ماتریس با ابعاد 4 را به ازای سایز grid برابر 4 در 4 بررسی می کنیم:

```
[-] N = 4
MatrixA(4,4), MatrixB(4,4)
Computing result using CUDA Kernel...
Elapsed time in msec = 0.008704
[-] Vector elements:
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000

[-] Vector elements:
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000

[-] Vector elements:
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
```

حال با توجه به نتایج بالا، این نتیجه استنباط می گردد که به خروجی این کد به ازای n=32 صحیح می باشد؛ حال زمان مورد نیاز برای محاسبه این خروجی نیز به صورت زیر می باشد:

```
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "GeForce 940MX" with compute capability 5.0

[-] N = 32
MatrixA(32,32), MatrixB(32,32)
Computing result using CUDA Kernel...
Elapsed time in msec = 0.012832
```

همچنین با توجه به اینکه یک بلوک نخ به ابعاد 32 در 32 موجود است و هر نخ در این بلوک یکی از خانه های خروجی این ضرب را انجام می دهد، به همین دلیل این روش برای ضرب آرایه 32 در 32 صحیح می باشد.

(همچنین کد این بخش در فایل matmul_step1.cu به پیوست قرار داده شده است.)

آرش حریرپوش
9731505

❖ گام دوم

راه حل اول:

برای بررسی صحت عملکرد این راه حل ابتدا این راه حل به صورت زیر پیاده سازی می گردد:

```
// System includes
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>
#include <device_launch_parameters.h>


/**
* Matrix multiplication (CUDA Kernel) on the device: C = A * B
*/
#define TILE_WIDTH 16
__global__ void
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int start_row = threadIdx.y * TILE_WIDTH;
	int end_row = start_row + TILE_WIDTH;
	int start_col = threadIdx.x * TILE_WIDTH;
	int end_col = start_col + TILE_WIDTH;
	for (int row = start_row; row < end_row; row++) {
		for (int col = start_col; col < end_col; col++) {
			float C_val = 0;
			for (int k = 0; k < n; ++k) {
				float A_elem = A[row * n + k];
				float B_elem = B[k * n + col];
				C_val += A_elem * B_elem;
			}
			C[row*n + col] = C_val;
		}
	}
}

void constantInit(float *data, int size, float val)
{
	for (int i = 0; i < size; ++i)
	{
		data[i] = val;
	}
}

// Prints a Matrices to the stdout.
void printMat(float * v, int n) {
	printf("[-] Vector elements: \n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++)
			printf("%f    ", v[i*n + j]);
		printf("\n");
	}
```

```
        printf("\b\b \n");
}

/**
* Run a simple test of matrix multiplication using CUDA
*/
int matrixMultiply(int argc, char **argv, int n)
{
        // Allocate host memory for matrices A and B
        unsigned int size_A = n * n;
        unsigned int mem_size_A = sizeof(float)* size_A;
        float *h_A = (float *)malloc(mem_size_A);
        unsigned int size_B = n * n;
        unsigned int mem_size_B = sizeof(float)* size_B;
        float *h_B = (float *)malloc(mem_size_B);

        // Initialize host memory
        const float valB = 0.01f;
        constantInit(h_A, size_A, 1.0f);
        constantInit(h_B, size_B, valB);

        // Allocate device memory
        float *d_A, *d_B, *d_C;

        // Allocate host matrix C
        unsigned int mem_size_C = n * n * sizeof(float);
        float *h_C = (float *)malloc(mem_size_C);

        if (h_C == NULL)
        {
                fprintf(stderr, "Failed to allocate host matrix C!\n");
                exit(EXIT_FAILURE);
        }

        cudaError_t error;

        error = cudaMalloc((void **)&d_A, mem_size_A);

        if (error != cudaSuccess)
        {
                printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
                exit(EXIT_FAILURE);
        }

        error = cudaMalloc((void **)&d_B, mem_size_B);

        if (error != cudaSuccess)
        {
                printf("cudaMalloc d_B returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
                exit(EXIT_FAILURE);
        }

        error = cudaMalloc((void **)&d_C, mem_size_C);

        if (error != cudaSuccess)
        {
```

```
		printf("cudaMalloc d_C returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// copy host memory to device
	error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_A,h_A) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_B,h_B) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// Setup execution parameters
	dim3 threads(32, 32,1);
	dim3 grid(1,1,1);

	// Create and start timer
	printf("Computing result using CUDA Kernel...\n");

	// Allocate CUDA events that we'll use for timing
	cudaEvent_t start;
	error = cudaEventCreate(&start);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	cudaEvent_t stop;
	error = cudaEventCreate(&stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the start event
	error = cudaEventRecord(start, NULL);

	if (error != cudaSuccess)
	{
```

```
		fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Execute the kernel
	matrixMulCUDA << < grid, threads >> > (d_C, d_A, d_B, n);

	error = cudaGetLastError();
	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to launch kernel!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the stop event
	error = cudaEventRecord(stop, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Wait for the stop event to complete
	error = cudaEventSynchronize(stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to synchronize on the stop event (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	float msecTotal = 0.0f;
	error = cudaEventElapsedTime(&msecTotal, start, stop);

	printf("Elapsed time in msec = %f\n", msecTotal);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to get time elapsed between events (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Copy result from device to host
	error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (h_C,d_C) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	/*printMat(h_A, n);
```

```
	printMat(h_B, n);
	printMat(h_C, n);*/

	// Clean up memory
	free(h_A);
	free(h_B);
	free(h_C);
	cudaFree(d_A);
	cudaFree(d_B);
	cudaFree(d_C);

	return EXIT_SUCCESS;

}


/**
* Program main
*/
int main(int argc, char **argv)
{
	printf("[Matrix Multiply Using CUDA] - Starting...\n");

	// By default, we use device 0
	int devID = 0;
	cudaSetDevice(devID);

	cudaError_t error;
	cudaDeviceProp deviceProp;
	error = cudaGetDevice(&devID);

	if (error != cudaSuccess)
	{
		printf("cudaGetDevice returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}

	error = cudaGetDeviceProperties(&deviceProp, devID);

	if (deviceProp.computeMode == cudaComputeModeProhibited)
	{
		fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no
threads can use ::cudaSetDevice().\n");
		exit(EXIT_SUCCESS);
	}

	if (error != cudaSuccess)
	{
		printf("cudaGetDeviceProperties returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}
	else
	{
		printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);
	}

	// Size of square matrices
```

```
        size_t n = 0;
        printf("[-] N = ");
        scanf("%u", &n);

        printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", n, n, n, n);

        int matrix_result = matrixMultiply(argc, argv, n);

        exit(matrix_result);
}
```

حال بخش مربوط به پرینت کردن نتایج را از کامنت خارج کرده و به ازای n = 4 صحت عملکرد این روش بررسی می گردد:

```
[-] N = 4
MatrixA(4,4), MatrixB(4,4)
Computing result using CUDA Kernel...
Elapsed time in msec = 0.809568
[-] Vector elements:
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000

[-] Vector elements:
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000

[-] Vector elements:
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
```

با توجه به نتایج بدست آمده در شکل بالا صحت عملکرد کد این مرحله تایید می گردد، حال زمان اجرای این برنامه برای آرایه های با سایز های مختلف مورد بررسی قرار می گیرد:

n	TILE_WIDTH	Block size	Elapsed Time (MSec)
64	16	32×32	8.796512
128	16	32×32	17.574017
128	32	32×32	72.183777
512	16	32×32	77.731102
1024	32	32×32	1275.687622
2048	64	32×32	10355.607422

همانطور که مشاهده می کنید با افزایش اندازه TILE_WIDTH زمان اجرای برنامه افزایش می یابد چرا که هر نخ کار بیشتری باید انجام دهد و کار بین تعداد نخ های کمتری تقسیم می شود و افزایش TILE_WIDTH زمانی باید رخ دهد که دیگر نتوان با اندازه قبلی محاسبات را به درستی انجام داد، برای مثال در جدول بالا اگر مقدار TILE_WIDTH را برای n = 1024 برابر 16 قرار دهیم خروجی به درستی محاسبه نمی شود چرا که در این حالت هر نخ $16 \times 16 = 256$ المان را محاسبه می کند و 1024 نخ نیز در حال اجرای این برنامه می باشند که باعث می شود $1024 \times 256 = 262144$ المان را بتوان محاسبه کرد که بزرگ ترین ماتریسی که می توان ضرب کرد برابر n = 512 می باشد؛ بنابراین برای محاسبه مقدار TILE_WIDTH برای n = 1024 به صورت زیر عمل می کنیم:

$$\sqrt{\frac{1024 \times 1024}{1024}} = \sqrt{1024} = 32$$

در نتیجه برای محاسبه ضرب ماتریس با n = 1024 اندازه TILE_WIDTH باید حداقل برابر 32 باشد؛ همچنین در این روش Occupancy برابر 16.6% می باشد.

و همچنین در این روش ضرب این آرایه ها توسط یک بلوک نخ 32 در 32 انجام می شود که در آن هر نخ وظیفه محاسبه چندین خانه از آرایه خروجی را بر عهده دارد، بنابراین حاصل خروجی بدست آمده در این راه حل در صورت انتخاب درست مقدار TILE_WIDTH صحیح می باشد.

راه حل دوم:

برای بررسی صحت عملکرد این راه حل ابتدا این راه حل به صورت زیر پیاده سازی می گردد:

```
// System includes
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>
#include <device_launch_parameters.h>


/**
* Matrix multiplication (CUDA Kernel) on the device: C = A * B
*/
#define TILE_WIDTH 16
__global__ void
```

```
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;
	float C_val = 0;
	for (int k = 0; k < n; ++k) {
		float A_elem = A[row * n + k];
		float B_elem = B[k * n + col];
		C_val += A_elem * B_elem;
	} C
		[row*n + col] = C_val;
}

void constantInit(float *data, int size, float val)
{
	for (int i = 0; i < size; ++i)
	{
		data[i] = val;
	}
}

// Prints a Matrices to the stdout.
void printMat(float * v, int n) {
	printf("[-] Vector elements: \n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++)
			printf("%f    ", v[i*n + j]);
		printf("\n");
	}
	printf("\b\b \n");
}

/**
* Run a simple test of matrix multiplication using CUDA
*/
int matrixMultiply(int argc, char **argv, int n)
{
	// Allocate host memory for matrices A and B
	unsigned int size_A = n * n;
	unsigned int mem_size_A = sizeof(float)* size_A;
	float *h_A = (float *)malloc(mem_size_A);
	unsigned int size_B = n * n;
	unsigned int mem_size_B = sizeof(float)* size_B;
	float *h_B = (float *)malloc(mem_size_B);

	// Initialize host memory
	const float valB = 0.01f;
	constantInit(h_A, size_A, 1.0f);
	constantInit(h_B, size_B, valB);

	// Allocate device memory
	float *d_A, *d_B, *d_C;

	// Allocate host matrix C
	unsigned int mem_size_C = n * n * sizeof(float);
	float *h_C = (float *)malloc(mem_size_C);

	if (h_C == NULL)
```

```
	{
		fprintf(stderr, "Failed to allocate host matrix C!\n");
		exit(EXIT_FAILURE);
	}

	cudaError_t error;

	error = cudaMalloc((void **)&d_A, mem_size_A);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_B, mem_size_B);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_B returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_C, mem_size_C);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_C returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// copy host memory to device
	error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_A,h_A) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_B,h_B) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// Setup execution parameters
	dim3 threads(32, 32,1);
	dim3 grid(1,1,1);

	// Create and start timer
```

```
	printf("Computing result using CUDA Kernel...\n");

	// Allocate CUDA events that we'll use for timing
	cudaEvent_t start;
	error = cudaEventCreate(&start);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	cudaEvent_t stop;
	error = cudaEventCreate(&stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the start event
	error = cudaEventRecord(start, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Execute the kernel
	matrixMulCUDA << < grid, threads >> > (d_C, d_A, d_B, n);

	error = cudaGetLastError();
	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to launch kernel!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the stop event
	error = cudaEventRecord(stop, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Wait for the stop event to complete
	error = cudaEventSynchronize(stop);

	if (error != cudaSuccess)
	{
```

```
		fprintf(stderr, "Failed to synchronize on the stop event (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	float msecTotal = 0.0f;
	error = cudaEventElapsedTime(&msecTotal, start, stop);

	printf("Elapsed time in msec = %f\n", msecTotal);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to get time elapsed between events (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Copy result from device to host
	error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (h_C,d_C) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	/*printMat(h_A, n);
	printMat(h_B, n);
	printMat(h_C, n);*/

	// Clean up memory
	free(h_A);
	free(h_B);
	free(h_C);
	cudaFree(d_A);
	cudaFree(d_B);
	cudaFree(d_C);

	return EXIT_SUCCESS;

}


/**
* Program main
*/
int main(int argc, char **argv)
{
	printf("[Matrix Multiply Using CUDA] - Starting...\n");

	// By default, we use device 0
	int devID = 0;
	cudaSetDevice(devID);

	cudaError_t error;
	cudaDeviceProp deviceProp;
	error = cudaGetDevice(&devID);
```

```
	if (error != cudaSuccess)
	{
		printf("cudaGetDevice returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}

	error = cudaGetDeviceProperties(&deviceProp, devID);

	if (deviceProp.computeMode == cudaComputeModeProhibited)
	{
		fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no
threads can use ::cudaSetDevice().\n");
		exit(EXIT_SUCCESS);
	}

	if (error != cudaSuccess)
	{
		printf("cudaGetDeviceProperties returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}
	else
	{
		printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);
	}

	// Size of square matrices
	size_t n = 0;
	printf("[-] N = ");
	scanf("%u", &n);

	printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", n, n, n, n);

	int matrix_result = matrixMultiply(argc, argv, n);

	exit(matrix_result);
}
```

حال بخش مربوط به پرینت کردن نتایج را از کامنت خارج کرده و به ازای n = 4 و سایز grid و بلوک برابر 2 در 2 صحت عملکرد این روش بررسی می گردد:

```
[-] N = 4
MatrixA(4,4), MatrixB(4,4)
Computing result using CUDA Kernel...
Elapsed time in msec = 0.009312
[-] Vector elements:
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000

[-] Vector elements:
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000

[-] Vector elements:
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
```

با توجه به نتایج بدست آمده در شکل بالا، صحت عملکرد کد این مرحله تایید می گردد، سپس برای محاسبه تسریع ابتدا کد سری ضرب ماتریس به صورت زیر نوشته می شود:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>


void constantInit(float *data, int size, float val);
void mulMat(float * a, float *b, float *c, int n);
void printMat(float * v, int n);

int main()
{
	const int n = 1024;
	const float valB = 0.01f;
	unsigned int arr_size = n * n;
	float * a;
	float * b;
	float * c;
	double elapsedtime, starttime;
	a = (float*)malloc(sizeof(float)*arr_size);
	b = (float*)malloc(sizeof(float)*arr_size);
	c = (float*)malloc(sizeof(float)*arr_size);

	constantInit(a, arr_size, 1.0f);
	constantInit(b, arr_size, valB);

	starttime = omp_get_wtime();

	mulMat(a, b, c, n);
```

```
	elapsedtime = omp_get_wtime() - starttime;


	/*printMat(a, n);
	printMat(b, n);
	printMat(c, n);*/

	// report elapsed time
	printf("Time Elapsed %f ms\n", elapsedtime * 1000);

	return EXIT_SUCCESS;
}

// Fills a Matrice with data
void constantInit(float *data, int size, float val)
{
	for (int i = 0; i < size; ++i)
	{
		data[i] = val;
	}
}

// Multiplys two Matrices
void mulMat(float * a, float *b, float *c, int n) {
	for (int row = 0; row < n; row++) {
		for (int col = 0; col < n; col++) {
			float C_val = 0;
			for (int k = 0; k < n; ++k) {
				float A_elem = a[row * n + k];
				float B_elem = b[k * n + col];
				C_val += A_elem * B_elem;
			}
			c[row*n + col] = C_val;
		}
	}

}

// Prints a Matrices to the stdout.
void printMat(float * v, int n) {
	printf("[-] Vector elements: \n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++)
			printf("%f    ", v[i*n + j]);
		printf("\n");
	}
	printf("\b\b \n");
}
```

حال بخش مربوط به پرینت کردن نتایج را از کامنت خارج کرده و به ازای n = 4 صحت عملکرد این روش بررسی می گردد:

```
[-] Vector elements:
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000

[-] Vector elements:
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000

[-] Vector elements:
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000

Time Elapsed 0.000300 ms
```

با توجه به نتایج بدست آمده در شکل بالا صحت عملکرد کد این مرحله تایید می گردد، حال در جدول زیر زمان اجرای برنامه سری برای آرایه های با سایز های مختلف اندازه گیری می گردد:

n	Elapsed Time (MSec)
256	28.382
1024	6572.6447
2048	70106.3447
4096	719906.7739

سپس مقادیر جدول زیر تکمیل می گردد:

	Block size	Grid size	Block size	Grid size	Block size	Grid size
	4×4	64×64×1	8×8	32×32×1	32×32	128×128×1
Elapsed time (MSec)	1.680128		0.993024		2783.899170	
Speed up	$\frac{28.382}{1.680128} \simeq 16.89$		$\frac{28.382}{0.993024} \simeq 28.58$		$\frac{719906.7739}{2783.899170} \simeq 258.59$	
Occupancy (Achieved)	49.66%		95.02%		90.58%	
Occupancy (Theoretical)	50.00%		100.00%		100.00%	

به دلیل وجود قابلیت TDR کرنل هایی که بیش از دو ثانیه زمان برای اجرا داشته باشند متوقف می شوند به همین دلیل با اجرای برنامه برای این اندازه grid خروجی زیر بدست می آید:

```
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "GeForce 940MX" with compute capability 5.0

[-] N = 4096
MatrixA(4096,4096), MatrixB(4096,4096)
Computing result using CUDA Kernel...
Failed to synchronize on the stop event (error code unspecified launch failure)!
```

برای برطرف کردن این مشکل برنامه Nsight Monitor را به صورت Administor اجرا کرده و از طریق تنظیمات Options این قابلیت غیر فعال گردیده و دستگاه reboot می شود.

(برای مشاهده جزئیات بیشتر منابع: https://stackoverflow.com/questions/16317505/cublas-failed-to-synchronize-stop-event و http://developer.download.nvidia.com/NsightVisualStudio/2.2/Documentation/UserGuide/HTML/Content/Timeout_Detection_Recovery.htm مراجعه شود)

همچنین برنامه به ازای اندازه grid را برابر 64×64×1 و اندازه بلوک 32 در 32 نیز در نظر گرفته می شود که در این صورت مقادیر زیر برابر است با:

Elapsed time = $341.709381\ MSec$

$$Speed\ Up = \frac{70106.3447}{341.709381} \simeq 205.16$$

Occupancy (Achieved) = 90.29%

Occupancy (Theoretical) = 100.00%

همچنین این مقادیر به ازای اندازه grid و بلوک برابر 32×32×1 نیز مورد بررسی قرار می گیرد:

Elapsed time = $43.433762\ MSec$

$$Speed\ Up = \frac{6572.6447}{43.433762} \simeq 151.32$$

Occupancy (Achieved) = 90.35%

Occupancy (Theoretical) = 100.00%

با توجه به سایز آرایه هایی که توسط هر یک از اندازه بلوک و grid قابل محاسبه می باشد، مقدار speed up و Occupancy محاسبه شده است که برای دو ستون اول آرایه ورودی 256 در 256 در نظر گرفته شده و برای مقدار بالا آرایه 2048 در 2048 در نظر گرفته شده است؛ همچنین مقدار Occupancy توسط Nsight اندازه گیری شده است.

برای محاسبه بیشترین اندازه n برای هر یک از مقادیر جدول بالا به صورت زیر عمل می گردد:

$$\sqrt{4 \times 4 \times 64 \times 64} = \sqrt{2^{16}} = 2^8 = 256$$

$$\sqrt{8 \times 8 \times 32 \times 32} = \sqrt{2^{16}} = 2^8 = 256$$

$$\sqrt{32 \times 32 \times 128 \times 128} = \sqrt{2^{24}} = 2^{12} = 4096$$

$$\sqrt{64 \times 64 \times 32 \times 32} = \sqrt{2^{22}} = 2^{11} = 2048$$

$$\sqrt{32 \times 32 \times 32 \times 32} = \sqrt{2^{20}} = 2^{10} = 1024$$

دلیل افزایش تسریع به ازای grid برابر 64×64 و بلوک 32×32 افزایش اندازه آرایه های ضرب شده در یک دیگر می باشد، چرا که با بزرگ تر شده آرایه ها زمان انجام محاسبات به صورت سری بر روی آن ها به شدت افزایش می یابد، برای مثال در جدول بالا با چهار برابر کردن اندازه آرایه با تغییر ابعاد آرایه از 1024 در 1024 به 2048 در 2048 زمان اجرای آن تقریبا 10.5 برابر شده است.

همانطور که مشاهده می کنید، آرایه هایی که در این روش می توان محاسبه کرد می تواند بسیار بزرگ تر از راه حل قبل و همچنین بسیار سریع تر باشد، همانطور که مشاهده می کنید در راه حل قبل یک آرایه 128 در 128 در بهترین حالت 17.574017 میلی ثانیه زمان برای اجرا نیاز دارد این درحالی است که در راه حل دوم یک آرایه 256 در 256 در بهترین حالت 0.993024 میلی ثانیه زمان برای اجرا نیاز دارد؛ به دلیل اینکه در این روش چندین بلوک می توانند به صورت همزمان با یک دیگر فعال باشند و محاسبات را انجام دهند و همچنین تعداد بلوک ها زیاد می باشد در نتیجه تاثیر تاخیر حافظه کاهش می یابد چراکه زمانی که یک بلوک نیاز به حافظه دارد از اجرا خارج شده و بلوک دیگری اجرای خود را ادامه می دهد تا این بلوک داده مورد نیاز خود از حافظه را دریافت کند.

راه حل سوم:

برای بررسی صحت عملکرد این راه حل ابتدا این راه حل به صورت زیر پیاده سازی می گردد:

```
// System includes
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>
#include <device_launch_parameters.h>


/**
* Matrix multiplication (CUDA Kernel) on the device: C = A * B
*/
#define TILE_WIDTH 16
__global__ void
matrixMulCUDA(float *C, float *A, float *B, int n)
{
	int start_row = blockDim.y * blockIdx.y * TILE_WIDTH + threadIdx.y * TILE_WIDTH;
	int end_row = start_row + TILE_WIDTH;
	int start_col = blockDim.x * blockIdx.x * TILE_WIDTH + threadIdx.x * TILE_WIDTH;
	int end_col = start_col + TILE_WIDTH;
	for (int row = start_row; row < end_row; row++) {
		for (int col = start_col; col < end_col; col++) {
			float C_val = 0;
			for (int k = 0; k < n; ++k) {
				float A_elem = A[row * n + k];
				float B_elem = B[k * n + col];
				C_val += A_elem * B_elem;
			}
			C[row*n + col] = C_val;
		}
	}
}

void constantInit(float *data, int size, float val)
{
	for (int i = 0; i < size; ++i)
	{
		data[i] = val;
	}
}

// Prints a Matrices to the stdout.
void printMat(float * v, int n) {
	printf("[-] Vector elements: \n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++)
			printf("%f    ", v[i*n + j]);
		printf("\n");
	}
	printf("\b\b \n");
}
```

```
/**
* Run a simple test of matrix multiplication using CUDA
*/
int matrixMultiply(int argc, char **argv, int n)
{
	// Allocate host memory for matrices A and B
	unsigned int size_A = n * n;
	unsigned int mem_size_A = sizeof(float)* size_A;
	float *h_A = (float *)malloc(mem_size_A);
	unsigned int size_B = n * n;
	unsigned int mem_size_B = sizeof(float)* size_B;
	float *h_B = (float *)malloc(mem_size_B);

	// Initialize host memory
	const float valB = 0.01f;
	constantInit(h_A, size_A, 1.0f);
	constantInit(h_B, size_B, valB);

	// Allocate device memory
	float *d_A, *d_B, *d_C;

	// Allocate host matrix C
	unsigned int mem_size_C = n * n * sizeof(float);
	float *h_C = (float *)malloc(mem_size_C);

	if (h_C == NULL)
	{
		fprintf(stderr, "Failed to allocate host matrix C!\n");
		exit(EXIT_FAILURE);
	}

	cudaError_t error;

	error = cudaMalloc((void **)&d_A, mem_size_A);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_B, mem_size_B);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_B returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_C, mem_size_C);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_C returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
```

```
		exit(EXIT_FAILURE);
	}

	// copy host memory to device
	error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_A,h_A) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_B,h_B) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// Setup execution parameters
	dim3 threads(32, 32,1);
	dim3 grid(1,1,1);

	// Create and start timer
	printf("Computing result using CUDA Kernel...\n");

	// Allocate CUDA events that we'll use for timing
	cudaEvent_t start;
	error = cudaEventCreate(&start);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	cudaEvent_t stop;
	error = cudaEventCreate(&stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the start event
	error = cudaEventRecord(start, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
```

```
	}

	// Execute the kernel
	matrixMulCUDA << < grid, threads >> > (d_C, d_A, d_B, n);

	error = cudaGetLastError();
	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to launch kernel!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Record the stop event
	error = cudaEventRecord(stop, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Wait for the stop event to complete
	error = cudaEventSynchronize(stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to synchronize on the stop event (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	float msecTotal = 0.0f;
	error = cudaEventElapsedTime(&msecTotal, start, stop);

	printf("Elapsed time in msec = %f\n", msecTotal);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to get time elapsed between events (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Copy result from device to host
	error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (h_C,d_C) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	/*printMat(h_A, n);
	printMat(h_B, n);
	printMat(h_C, n);*/
```

```
	// Clean up memory
	free(h_A);
	free(h_B);
	free(h_C);
	cudaFree(d_A);
	cudaFree(d_B);
	cudaFree(d_C);

	return EXIT_SUCCESS;

}


/**
* Program main
*/
int main(int argc, char **argv)
{
	printf("[Matrix Multiply Using CUDA] - Starting...\n");

	// By default, we use device 0
	int devID = 0;
	cudaSetDevice(devID);

	cudaError_t error;
	cudaDeviceProp deviceProp;
	error = cudaGetDevice(&devID);

	if (error != cudaSuccess)
	{
		printf("cudaGetDevice returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}

	error = cudaGetDeviceProperties(&deviceProp, devID);

	if (deviceProp.computeMode == cudaComputeModeProhibited)
	{
		fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no
threads can use ::cudaSetDevice().\n");
		exit(EXIT_SUCCESS);
	}

	if (error != cudaSuccess)
	{
		printf("cudaGetDeviceProperties returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}
	else
	{
		printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);
	}

	// Size of square matrices
	size_t n = 0;
	printf("[-] N = ");
	scanf("%u", &n);
```

```
        printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", n, n, n, n);

        int matrix_result = matrixMultiply(argc, argv, n);

        exit(matrix_result);
}
```

حال بخش مربوط به پرینت کردن نتایج را از کامنت خارج کرده و به ازای n = 4 صحت عملکرد این روش بررسی می گردد:

```
[-] N = 4
MatrixA(4,4), MatrixB(4,4)
Computing result using CUDA Kernel...
Elapsed time in msec = 0.810848
[-] Vector elements:
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000

[-] Vector elements:
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000

[-] Vector elements:
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
```

با توجه به نتایج بدست آمده در شکل بالا، صحت عملکرد کد این مرحله تایید می گردد، حال زمان اجرای این برنامه برای آرایه های با سایز های مختلف مورد بررسی قرار می گیرد:

n	TILE_WIDTH	Block size	Grid Size	Elapsed Time (MSec)	Occupancy (Achieved)
256	16	4×4	4×4	3.097984	8.26%
256	8	4×4	8×8	2.261728	32.11%
512	16	32×32	1×1	76.666016	16.65%
512	16	16×16	2×2	35.271103	16.66%
1024	8	32×32	4×4	650.800232	90.79%
1024	4	8×8	32×32	98.014397	96.04%
2048	16	8×8	16×16	1975.368164	91.01%
2048	8	8×8	32×32	1842.232666	95.88%

به کمک نتایج جدول بالا می توان نتیجه گرفت، در این راه حل با افزایش سایز grid و کاهش سایز بلوک و TILE_WIDTH مقدار Occupancy افزایش و زمان اجرای برنامه کاهش می یابد. به این دلیل که هر چه اندازه TILE_WIDTH را کاهش دهیم میزان کار نخ ها کاهش پیدا کرده و کار بین تعداد نخ های بیشتری تقسیم می شود و هر چه سایز grid را افزایش دهیم بلوک های برنامه افزایش می یابد و تعداد بلوک های موازی که با هم اجرا می شوند افزایش می یابد و همچنین تاثیر تاخیر حافظه نیز کاهش می یابد (البته کاهش و افزایش گفته شده دارای یک حد آستانه ای می باشد که از آن حد آستانه به بعد دیگر بهبود رخ نخواهد داد).

به کمک جدول زیر زمان اجرای این سه راه حل با یک دیگر مقایسه می گردد:

Solution_Number	N	**Elapsed Time (MSec)**
1	1024	1275.687622
2	1024	43.433762
3	1024	98.014397

با توجه به جدول بالا، سریع ترین زمان اجرا متعلق به راه حل دوم می باشد و پس از آن راه حل سوم و در نهایت راه حل اول از هر سه راه حل کند تر می باشد؛ که به این دلیل می باشد که در راه حل اول تنها یک بلوک همواره در حال اجرا شدن می باشد که به همین دلیل به دلیل تاخیر های حافظه زمان اجرا به شدت افزایش پیدا کرده است و در روش سوم نیز به دلیل اینکه هر نخ چندین خانه خروجی را محاسبه می کند، و به دلیل اینکه این اطلاعات باید از حافظه آورده شود دسترسی به حافظه نیز زمان بر می باشد کند تر از روش دوم می باشد اما چون چندین بلوک همزمان فعال هستند و تعداد بلوک ها نسبت به روش قبل بیشتر است بنابراین از روش اول سریع تر است و در نهایت روش دوم به دلیل اینکه دارای بلوک های زیاد می باشد و هر نخ مسئول محاسبه یک خانه خروجی می باشد و همچنین به دلیل اینکه استفاده از cache ها به صورت بهینه تری صورت می پذیرد، این راه حل بسیار سریع تر دو روش دیگر می باشد.

(همچنین کد مرحله اول این بخش در فایل matmul_step2_1.cu و کد مرحله دوم این بخش در فایل matmul_step2_2.cu و کد مرحله سوم این بخش در فایل matmul_step2_3.cu و کد سریال ضرب ماتریس در فایل matMul_serial.cpp به پیوست قرار داده شده است.)

❖ گام سوم

ابتدا برنامه ای به کمک cuBLAS برای پیاده سازی ضرب ماتریس ها به صورت زیر پیاده سازی می گردد:

```
// System includes
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// CUDA runtime
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cublas_v2.h>

/**
* Matrix multiplication (CUDA Kernel) on the device: C = A * B
*/

void
matrixMulCUDA(cublasHandle_t &handle, float *C, float *A, float *B, int n)
{
   int lda = n, ldb = n, ldc = n;

   // Scalaing factors
   const float alf = 1.0f;
   const float bet = 0.0f;

   const float *alpha = &alf;
   const float *beta = &bet;

   // Do the actual multiplication
       // Calculate: c = (alpha*a) * b + (beta*c)
       // MxN = MxK * KxN
       // Signature: handle, operation, operation, M, N, K, alpha, A, lda, B, ldb,
       // beta, C, ldc
   cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, n, n, alpha, A, lda, B, ldb, beta, C,
ldc);

   // Destroy the handle
   cublasDestroy(handle);

}

void constantInit(float *data, int size, float val)
{
       for (int i = 0; i < size; ++i)
       {
              data[i] = val;
       }
}

// Prints a Matrices to the stdout.
void printMat(float * v, int n) {
       int i;
```

```
	printf("[-] Vector elements: \n");
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++)
			printf("%f    ", v[i*n + j]);
		printf("\n");
	}
	printf("\b\b \n");
}

/**
* Run a simple test of matrix multiplication using CUDA
*/
int matrixMultiply(int argc, char **argv, int n)
{
	// cuBLAS handle
	cublasHandle_t handle;
	cublasCreate(&handle);

	// Allocate host memory for matrices A and B
	unsigned int size_A = n * n;
	unsigned int mem_size_A = sizeof(float)* size_A;
	float *h_A = (float *)malloc(mem_size_A);
	unsigned int size_B = n * n;
	unsigned int mem_size_B = sizeof(float)* size_B;
	float *h_B = (float *)malloc(mem_size_B);

	// Initialize host memory
	const float valB = 0.01f;
	constantInit(h_A, size_A, 1.0f);
	constantInit(h_B, size_B, valB);

	// Allocate device memory
	float *d_A, *d_B, *d_C;

	// Allocate host matrix C
	unsigned int mem_size_C = n * n * sizeof(float);
	float *h_C = (float *)malloc(mem_size_C);

	if (h_C == NULL)
	{
		fprintf(stderr, "Failed to allocate host matrix C!\n");
		exit(EXIT_FAILURE);
	}

	cudaError_t error;

	error = cudaMalloc((void **)&d_A, mem_size_A);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_B, mem_size_B);

	if (error != cudaSuccess)
```

```
	{
		printf("cudaMalloc d_B returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMalloc((void **)&d_C, mem_size_C);

	if (error != cudaSuccess)
	{
		printf("cudaMalloc d_C returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// copy host memory to device
	error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_A,h_A) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	error = cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (d_B,h_B) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	// Create and start timer
	printf("Computing result using CUDA Kernel...\n");

	// Allocate CUDA events that we'll use for timing
	cudaEvent_t start;
	error = cudaEventCreate(&start);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	cudaEvent_t stop;
	error = cudaEventCreate(&stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to create stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}
```

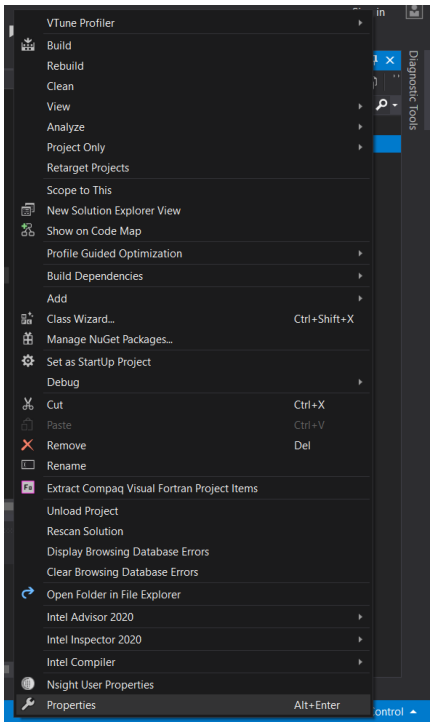
```
	// Record the start event
	error = cudaEventRecord(start, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record start event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Multiply d_A and d_B on GPU
	matrixMulCUDA (handle, d_C, d_A, d_B, n);

	// Record the stop event
	error = cudaEventRecord(stop, NULL);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to record stop event (error code %s)!\n",
cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Wait for the stop event to complete
	error = cudaEventSynchronize(stop);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to synchronize on the stop event (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	float msecTotal = 0.0f;
	error = cudaEventElapsedTime(&msecTotal, start, stop);

	printf("Elapsed time in msec = %f\n", msecTotal);

	if (error != cudaSuccess)
	{
		fprintf(stderr, "Failed to get time elapsed between events (error code
%s)!\n", cudaGetErrorString(error));
		exit(EXIT_FAILURE);
	}

	// Copy result from device to host
	error = cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

	if (error != cudaSuccess)
	{
		printf("cudaMemcpy (h_C,d_C) returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
		exit(EXIT_FAILURE);
	}

	/*printMat(h_A, n);
	printMat(h_B, n);
	printMat(h_C, n);*/
```

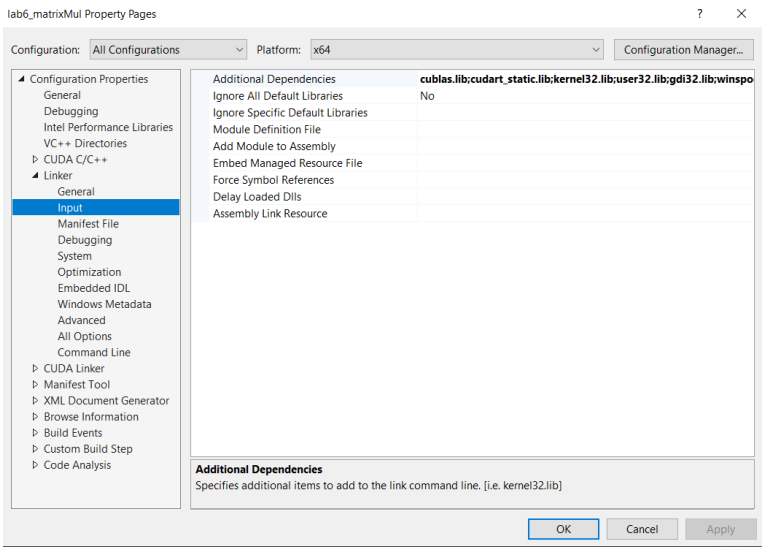
```
	// Clean up memory
	free(h_A);
	free(h_B);
	free(h_C);
	cudaFree(d_A);
	cudaFree(d_B);
	cudaFree(d_C);

	return EXIT_SUCCESS;

}


/**
* Program main
*/
int main(int argc, char **argv)
{
	printf("[Matrix Multiply Using CUDA] - Starting...\n");

	// By default, we use device 0
	int devID = 0;
	cudaSetDevice(devID);

	cudaError_t error;
	cudaDeviceProp deviceProp;
	error = cudaGetDevice(&devID);

	if (error != cudaSuccess)
	{
		printf("cudaGetDevice returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}

	error = cudaGetDeviceProperties(&deviceProp, devID);

	if (deviceProp.computeMode == cudaComputeModeProhibited)
	{
		fprintf(stderr, "Error: device is running in <Compute Mode Prohibited>, no
threads can use ::cudaSetDevice().\n");
		exit(EXIT_SUCCESS);
	}

	if (error != cudaSuccess)
	{
		printf("cudaGetDeviceProperties returned error %s (code %d), line(%d)\n",
cudaGetErrorString(error), error, __LINE__);
	}
	else
	{
		printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
deviceProp.name, deviceProp.major, deviceProp.minor);
	}

	// Size of square matrices
	size_t n = 0;
	printf("[-] N = ");
```

```
    scanf("%u", &n);
    printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", n, n, n, n);
    int matrix_result = matrixMultiply(argc, argv, n);
    exit(matrix_result);
}
```

سپس برای اینکه این برنامه بتواند اجرا شود، ابتدا در بخش Properties سپس Linker و پس از آن Input کتاب خانه cuBLAS را به بخش Additional Dependencies اضافه می گردد؛

مراحل انجام این کار به صورت زیر می باشد:





حال بخش مربوط به پرینت کردن نتایج را از کامنت خارج کرده و به ازای n = 4 صحت عملکرد کد بررسی می گردد:

```
[-] N = 4
MatrixA(4,4), MatrixB(4,4)
Computing result using CUDA Kernel...
Elapsed time in msec = 0.210976
[-] Vector elements:
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000
1.000000        1.000000        1.000000        1.000000

[-] Vector elements:
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000
0.010000        0.010000        0.010000        0.010000

[-] Vector elements:
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
0.040000        0.040000        0.040000        0.040000
```

با توجه به نتایج بدست آمده در شکل بالا صحت عملکرد کد این مرحله تایید می گردد، سپس برای محاسبه سرعت پیاده سازی در گام دوم از راه حل دوم ارائه شده که از بقیه راه حل ها بهتر می باشد استفاده می گردد، همچنین برای هر اندازه از آرایه ورودی مقدار اندازه بلوک و grid متفاوتی استفاده می گردد؛ که به صورت زیر می باشد:

ArraySize	BlockSize	GridSize
32 × 32	8 × 8	4 × 4
128 × 128	8 × 8	16 × 16
256 × 256	16 × 16	16 × 16
512 × 512	16 × 16	32 × 32

همچنین زمان اجرای برنامه سری برای آرایه های با سایز های مختلف اندازه گیری می گردد:

n	Elapsed Time (MSec)
32	0.051800
128	3.502100
256	28.382
512	524.956800

سپس مقادیر جدول زیر تکمیل می گردد:

	32 × 32	128 × 128	256 × 256	512 × 512
پیاده سازی گام دوم	$\frac{0.051800}{0.011040} \simeq 4.69$	$\frac{3.502100}{0.134720} \simeq 25.99$	$\frac{28.382}{0.666752} \simeq 42.56$	$\frac{524.956800}{5.496448} \simeq 95.50$
cuBLAS	$\frac{0.051800}{0.219968} \simeq 0.23$	$\frac{3.502100}{0.376352} \simeq 9.30$	$\frac{28.382}{0.327136} \simeq 86.75$	$\frac{524.956800}{1.687712} \simeq 311.04$
افزایش سرعت	-4.46	-16.69	44.19	215.54

همانطور که از جدول بالا مشاهده می شود، استفاده از کتاب خانه cuBLAS در ماتریس های با اندازه زیاد (بزرگ تر از 256×256) بسیار موثر عمل می کند و میزان تسریع برنامه به شدت افزایش می یابد، این در حالی است که برای ماتریس های کوچک تر از این سایز این روش موثر نمی باشد و احتمالا میزان سربار هایی که ایجاد می کند بیشتر از بهینه سازی باشد که انجام می دهد، به همین دلیل برای ماتریس های با اندازه های کوچک استفاده از راه حل دوم گام قبل با استفاده از اندازه بلوک و grid متناسب با آن برنامه موثر تر از استفاده از کتاب خانه cuBLAS می باشد؛ بنابراین استفاده از cuBLAS برای ماتریس های به اندازه 256 ×256 و 512 ×512 موثر و استفاده از مرحله دوم گام دوم برای ماتریس های به اندازه 32 ×32 و 128 × 128 موثر می باشد.

(همچنین کد این بخش در فایل matmul_step3.cu به پیوست قرار داده شده است.)