درس برنامه نویسی چند هسته ای

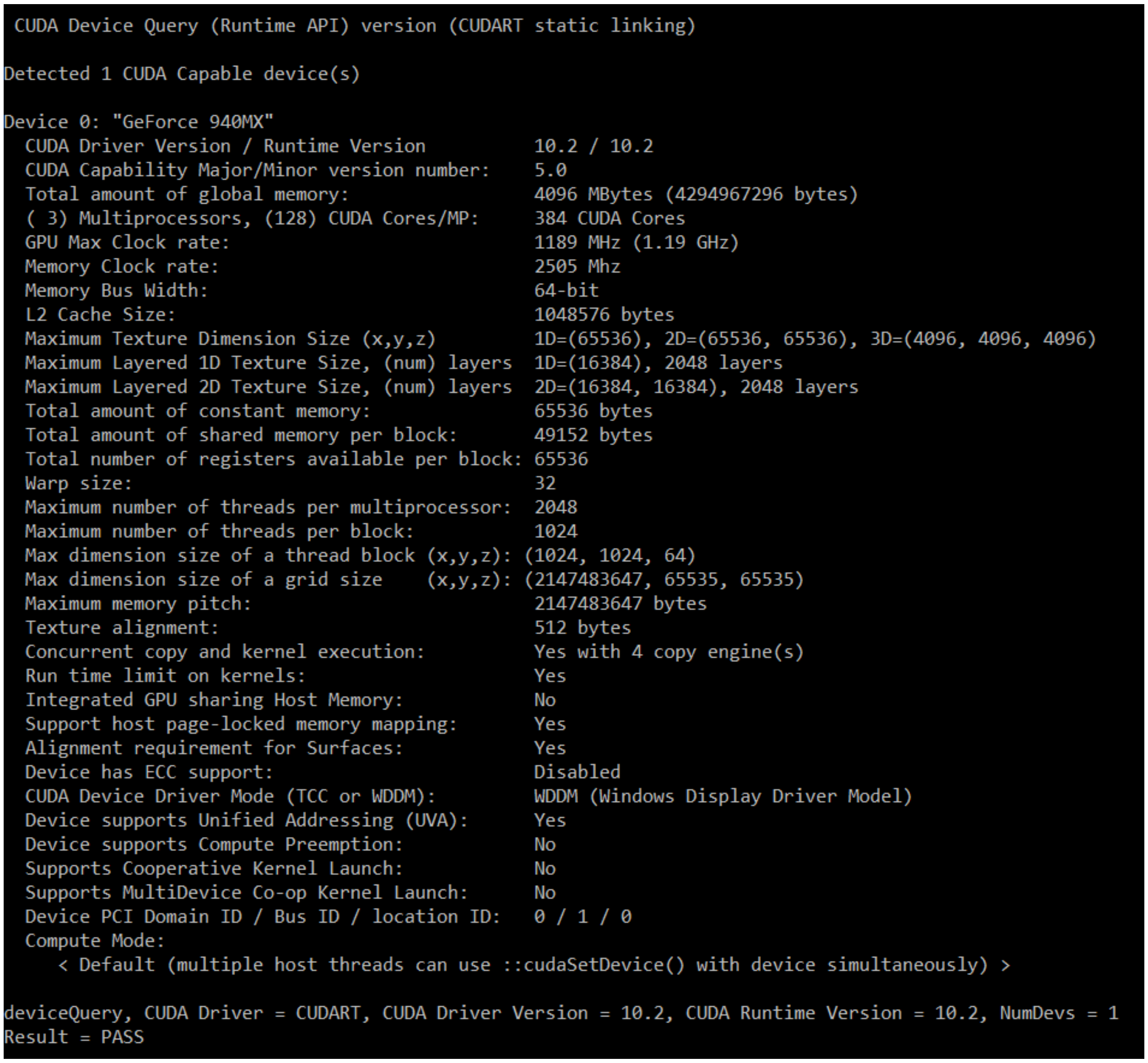
آزمایش پنجم

آرش حریرپوش
9731505

آرش حریرپوش
9731505

❖ گام اول

خروجی اجرای کد موجود در sample های CUDA خروجی زیر نمایش داده می شود:



```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce 940MX"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 4096 MBytes (4294967296 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1189 MHz (1.19 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 4 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version = 10.2, NumDevs = 1
Result = PASS
```

❖ گام دوم

مطابق دستور العمل های عنوان شده در دستورکار آزمایش کد داده شده را به صورت زیر تغییر می دهیم:

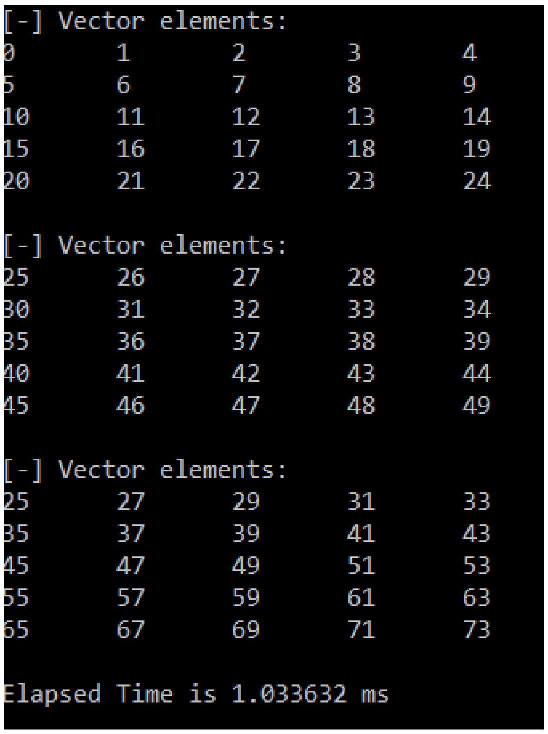
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include<stdlib.h>
#include <stdio.h>
```

آرش حریرپوش
9731505

```
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX,
unsigned int matSizeY);
void fillMat(int * v, int matSizeX, int matSizeY);
void printMat(int * v, int matSizeX, int matSizeY);
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
       int j = threadIdx.y;

    c[i*blockDim.y+j] = a[i*blockDim.y + j] + b[i*blockDim.y + j];
}

int main()
{
       const int matSizeX = 32;
       const int matSizeY = 32;
       int * a;
       int * b;
       int * c;
       cudaEvent_t start;
       cudaEventCreate(&start);
       cudaEvent_t stop;

       cudaEventCreate(&stop);
       a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
       b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
       c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);

       fillMat(a, matSizeX, matSizeY);
       fillMat(b, matSizeX, matSizeY);


       cudaEventRecord(start, NULL);

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

       cudaEventRecord(stop, NULL);

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

       printMat(a, matSizeX, matSizeY);
       printMat(b, matSizeX, matSizeY);
       printMat(c, matSizeX, matSizeY);


       fprintf(stderr, "Elapsed Time is %f ms \n", msecTotal);

    // 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;
}
void fillMat(int * v, int matSizeX, int matSizeY) {
       static int L = 0;
       for (int i = 0; i < matSizeX; i++) {
              for (int j = 0; j < matSizeY; j++)
                     v[i*matSizeY + j] = L++;


       }
}
void printMat(int * v, int matSizeX, int matSizeY) {
       int i;
       printf("[-] Vector elements: \n");
       for (int i = 0; i < matSizeX; i++) {
              for (int j = 0; j < matSizeY; j++)
                     printf("%d    ", v[i*matSizeY + j]);
              printf("\n");

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

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX,
unsigned int matSizeY)
{
    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, matSizeX*matSizeY * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_a, matSizeX*matSizeY * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_b, matSizeX*matSizeY * 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, matSizeX*matSizeY * sizeof(int),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_b, b, matSizeX*matSizeY * sizeof(int),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
       dim3 block_size = dim3(matSizeX, matSizeY, 1);
    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, block_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, matSizeX*matSizeY * sizeof(int),
cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

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

    return cudaStatus;
}
```

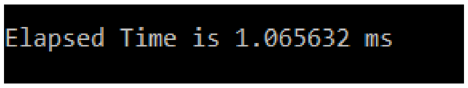
(همچنین کد این بخش در فایل MattAdd_step2.cu و کد سریال برنامه در فایل MatAdd.cpp به پیوست قرار داده شده است.)

خروجی کد بالا عبارتست از:



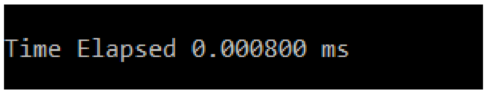
```
[-] Vector elements:
0       1       2       3       4
5       6       7       8       9
10      11      12      13      14
15      16      17      18      19
20      21      22      23      24

[-] Vector elements:
25      26      27      28      29
30      31      32      33      34
35      36      37      38      39
40      41      42      43      44
45      46      47      48      49

[-] Vector elements:
25      27      29      31      33
35      37      39      41      43
45      47      49      51      53
55      57      59      61      63
65      67      69      71      73

Elapsed Time is 1.033632 ms
```

زمان اجرای برنامه موازی برای آرایه های 4 در 4



```
Elapsed Time is 1.065632 ms
```

زمان اجرای برنامه موازی برای آرایه های 32 در 32



```
Time Elapsed 0.000800 ms
```

زمان اجرای برنامه سری برای آرایه های 32 در 32

با توجه به زمان اندازه گیری شده که در شکل بالا قابل مشاهده است، به علت این که جابجایی این المان ها بین host و device زمان بر می باشد، زمان اجرای این برنامه که توسط GPU اجرا می شود بسیار بیشتر از برنامه سریال می باشد.

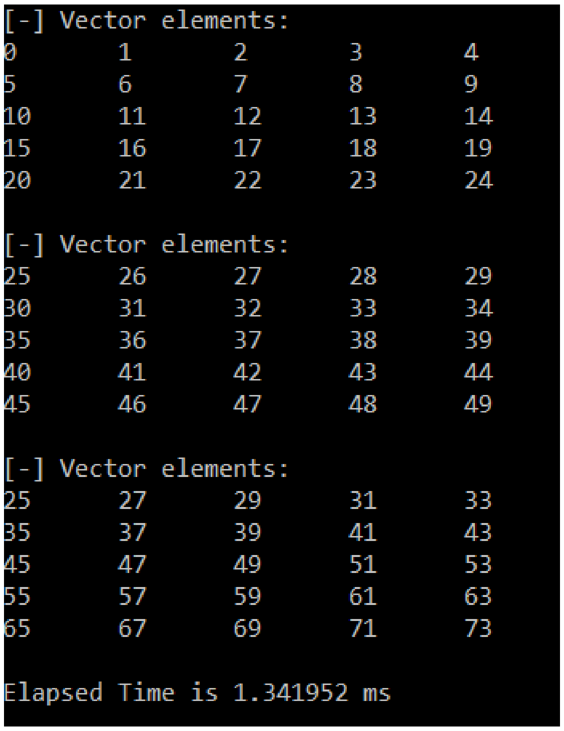
❖ گام سوم

روش اول: در این روش برای محاسبه جمع آرایه ها یک نخ n جمع انجام دهد، در این روش اندازه بلوک را برابر 32 در 32 در نظر می گیریم که برابر 1024 نخ بیشترین نخی که یک بلوک می تواند پشتیبانی کند قرار می دهیم؛ کد این روش به صورت زیر می باشد:

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

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX,
unsigned int matSizeY);
void fillMat(int * v, int matSizeX, int matSizeY);
void printMat(int * v, int matSizeX, int matSizeY);
__global__ void addKernel(int *c, const int *a, const int *b, int size)
{
	int i = threadIdx.x;
	int j = threadIdx.y;

	int n = int((size-1) / (blockDim.x * blockDim.y)) + 1;

	int index = (i * blockDim.y + j) * n;

	for (int k = 0; k < n; k++) {
		if (index + k < size) {
			int ni = index + k;
			c[ni] = a[ni] + b[ni];
		}
	}
}

int main()
{
	const int matSizeX = 1024;
	const int matSizeY = 1024;
	int * a;
	int * b;
	int * c;
	cudaEvent_t start;
	cudaEventCreate(&start);
	cudaEvent_t stop;

	cudaEventCreate(&stop);
	a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
	b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
	c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);

	fillMat(a, matSizeX, matSizeY);
	fillMat(b, matSizeX, matSizeY);


	cudaEventRecord(start, NULL);

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

	cudaEventRecord(stop, NULL);
	cudaStatus = cudaEventSynchronize(stop);
	float msecTotal = 0.0f;
	cudaStatus = cudaEventElapsedTime(&msecTotal, start, stop);

	/*printMat(a, matSizeX, matSizeY);
	printMat(b, matSizeX, matSizeY);
	printMat(c, matSizeX, matSizeY);*/

	fprintf(stderr, "Elapsed Time is %f ms \n", msecTotal);

	// 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;
}
void fillMat(int * v, int matSizeX, int matSizeY) {
	static int L = 0;
	for (int i = 0; i < matSizeX; i++) {
		for (int j = 0; j < matSizeY; j++)
			v[i*matSizeY + j] = L++;


	}
}
void printMat(int * v, int matSizeX, int matSizeY) {
	int i;
	printf("[-] Vector elements: \n");
	for (int i = 0; i < matSizeX; i++) {
		for (int j = 0; j < matSizeY; j++)
			printf("%d    ", v[i*matSizeY + j]);
		printf("\n");

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

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX,
unsigned int matSizeY)
{
	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, matSizeX*matSizeY * sizeof(int));
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMalloc failed!");
		goto Error;
	}
	cudaStatus = cudaMalloc((void**)&dev_a, matSizeX*matSizeY * sizeof(int));
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMalloc failed!");
		goto Error;
	}
	cudaStatus = cudaMalloc((void**)&dev_b, matSizeX*matSizeY * 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, matSizeX*matSizeY * sizeof(int), 
cudaMemcpyHostToDevice);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}

	cudaStatus = cudaMemcpy(dev_b, b, matSizeX*matSizeY * sizeof(int), 
cudaMemcpyHostToDevice);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}
	int arrSize = matSizeX * matSizeY;
	dim3 block_size = dim3(32, 32, 1);
	// Launch a kernel on the GPU with one thread for each element.
	addKernel << <1, block_size >> > (dev_c, dev_a, dev_b, arrSize);

	// 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, matSizeX*matSizeY * sizeof(int), 
cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
```

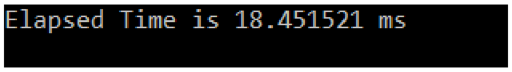
```
			goto Error;
		}

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

		return cudaStatus;
}
```

برای بررسی صحت درستی کد بالا، آن را به ازای کرنل 1 در 1 و سایز آرایه 5 در 5 تست می کنیم، که نتیجه آن برابر است با:



```
[-] Vector elements:
0       1       2       3       4
5       6       7       8       9
10      11      12      13      14
15      16      17      18      19
20      21      22      23      24

[-] Vector elements:
25      26      27      28      29
30      31      32      33      34
35      36      37      38      39
40      41      42      43      44
45      46      47      48      49

[-] Vector elements:
25      27      29      31      33
35      37      39      41      43
45      47      49      51      53
55      57      59      61      63
65      67      69      71      73

Elapsed Time is 1.341952 ms
```

حال که درستی اجرای کد اثبات گردیده است، زمان مورد نیاز برای جمع دو آرایه به اندازه های 1024 در 1024 و اندازه بلوک 32 در 32 اندازه گیری می گردد:



```
Elapsed Time is 18.451521 ms
```

روش دوم: در این روش برای محاسبه جمع آرایه ها n بلوک که هر یک دارای 1024 نخ می باشد اجرا می گردد، و هر نخ مسئول محاسبه جمع یک المان از آرایه خروجی می باشد، در این روش اندازه بلوک را برابر 32 در 32 در نظر می گیریم که برابر 1024 نخ بیشترین نخی که یک بلوک می تواند پشتیبانی کند قرار می دهیم؛ کد این روش به صورت زیر می باشد:

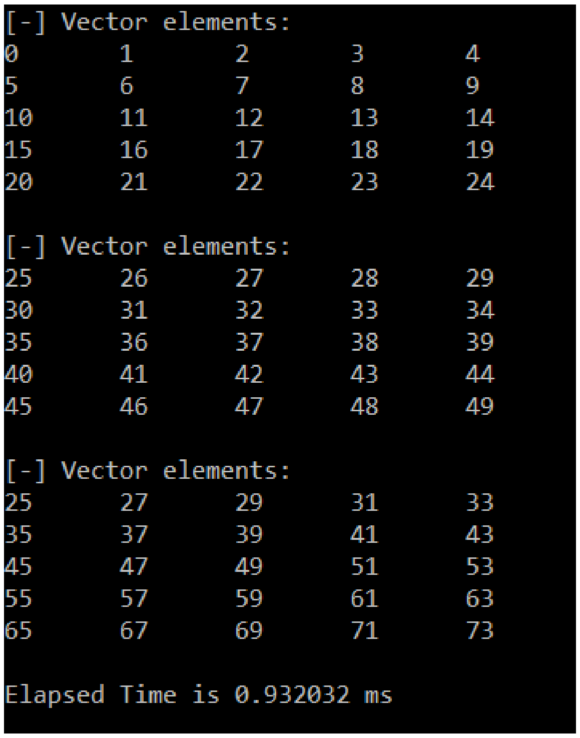
آرش حریرپوش
9731505

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

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX,
unsigned int matSizeY);
void fillMat(int * v, int matSizeX, int matSizeY);
void printMat(int * v, int matSizeX, int matSizeY);
__global__ void addKernel(int *c, const int *a, const int *b, int size)
{
	int index = threadIdx.x + blockIdx.x * blockDim.x;
	if (index < size) {
		c[index] = a[index] + b[index];
	}
}

int main()
{
	const int matSizeX = 1024;
	const int matSizeY = 1024;
	int * a;
	int * b;
	int * c;
	cudaEvent_t start;
	cudaEventCreate(&start);
	cudaEvent_t stop;

	cudaEventCreate(&stop);
	a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
	b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
	c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);

	fillMat(a, matSizeX, matSizeY);
	fillMat(b, matSizeX, matSizeY);


	cudaEventRecord(start, NULL);

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

	cudaEventRecord(stop, NULL);

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

	/*printMat(a, matSizeX, matSizeY);
	printMat(b, matSizeX, matSizeY);
	printMat(c, matSizeX, matSizeY);*/

	fprintf(stderr, "Elapsed Time is %f ms \n", msecTotal);
```

```
        // 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;
}
void fillMat(int * v, int matSizeX, int matSizeY) {
        static int L = 0;
        for (int i = 0; i < matSizeX; i++) {
               for (int j = 0; j < matSizeY; j++)
                      v[i*matSizeY + j] = L++;


        }
}
void printMat(int * v, int matSizeX, int matSizeY) {
        int i;
        printf("[-] Vector elements: \n");
        for (int i = 0; i < matSizeX; i++) {
               for (int j = 0; j < matSizeY; j++)
                      printf("%d    ", v[i*matSizeY + j]);
               printf("\n");

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

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int matSizeX,
unsigned int matSizeY)
{
        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, matSizeX*matSizeY * sizeof(int));
        if (cudaStatus != cudaSuccess) {
               fprintf(stderr, "cudaMalloc failed!");
               goto Error;
        }
        cudaStatus = cudaMalloc((void**)&dev_a, matSizeX*matSizeY * sizeof(int));
        if (cudaStatus != cudaSuccess) {
               fprintf(stderr, "cudaMalloc failed!");
               goto Error;
```

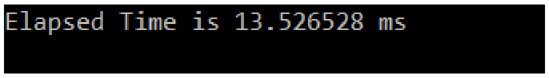
```
	}
	cudaStatus = cudaMalloc((void**)&dev_b, matSizeX*matSizeY * sizeof(int));
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMalloc failed!");
		goto Error;
	}

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

	cudaStatus = cudaMemcpy(dev_b, b, matSizeX*matSizeY * sizeof(int),
cudaMemcpyHostToDevice);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}
	int arrSize = matSizeX * matSizeY;
	int nb = int((arrSize - 1) / 1024) + 1;

	dim3 block_size = dim3(32, 32, 1);
	dim3 numOfBlock = dim3(nb, 1, 1);
	// Launch a kernel on the GPU with one thread for each element.
	addKernel << <numOfBlock, block_size >> > (dev_c, dev_a, dev_b, arrSize);

	// 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, matSizeX*matSizeY * sizeof(int),
cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}

Error:
	cudaFree(dev_c);
	cudaFree(dev_a);
	cudaFree(dev_b);
	return cudaStatus;
}
```

برای بررسی صحت درستی کد بالا، تعدادی کرنل 1 در 1 برای آرایه ورودی 5 در 5 اجرا می گردد، که نتیجه آن برابر است با:



```
[-] Vector elements:
0       1       2       3       4
5       6       7       8       9
10      11      12      13      14
15      16      17      18      19
20      21      22      23      24

[-] Vector elements:
25      26      27      28      29
30      31      32      33      34
35      36      37      38      39
40      41      42      43      44
45      46      47      48      49

[-] Vector elements:
25      27      29      31      33
35      37      39      41      43
45      47      49      51      53
55      57      59      61      63
65      67      69      71      73

Elapsed Time is 0.932032 ms
```

حال که درستی اجرای کد اثبات گردیده است، زمان مورد نیاز برای جمع دو آرایه به اندازه های 1024 در 1024 و اندازه بلوک 32 در 32 اندازه گیری می گردد:



```
Elapsed Time is 13.526528 ms
```

با مقایسه زمان های اندازه گیری شده برای روش های یک و دوم مشخص می گردد که روش دوم سریع تر از روش اول می باشد، به این دلیل که زمانی که یک برنامه بلوک های نخ بیشتری داشته باشد، زمانی که یک بلوک به دلیل نیاز به حافظه متوقف می شود schedular انتخاب های جایگزین بیشتری دارد تا بدون ایجاد تاخیر بلوک دیگری را اجرا کند و زمانی که داده های آن بلوک آورده شد توسط schedular در زمان مناسب دوباره انتخاب و در یکی از SM ها دوباره اجرا می شود، همچنین با افزایش تعداد بلوک ها می توان همزمان به اندازه تعداد SM ها کار موازی انجام داد، در حالی که در روش اول تنها یک SM فعال است به دلیل اینکه بلوک ها به SM ها

Assign می شوند و در این روش کرنل تنها شامل یک بلوک می شود و در روش دوم چندین SM همزمان فعال می شوند و کار برنامه را پیش می برند به همین علت روش دوم سریع تر از روش اول می باشد.

(همچنین کد روش اول این بخش در فایل MattAdd_step3_1.cu و کد روش دوم این بخش در فایل MattAdd_step3_2.cu به پیوست قرار داده شده است.)

❖ گام چهارم

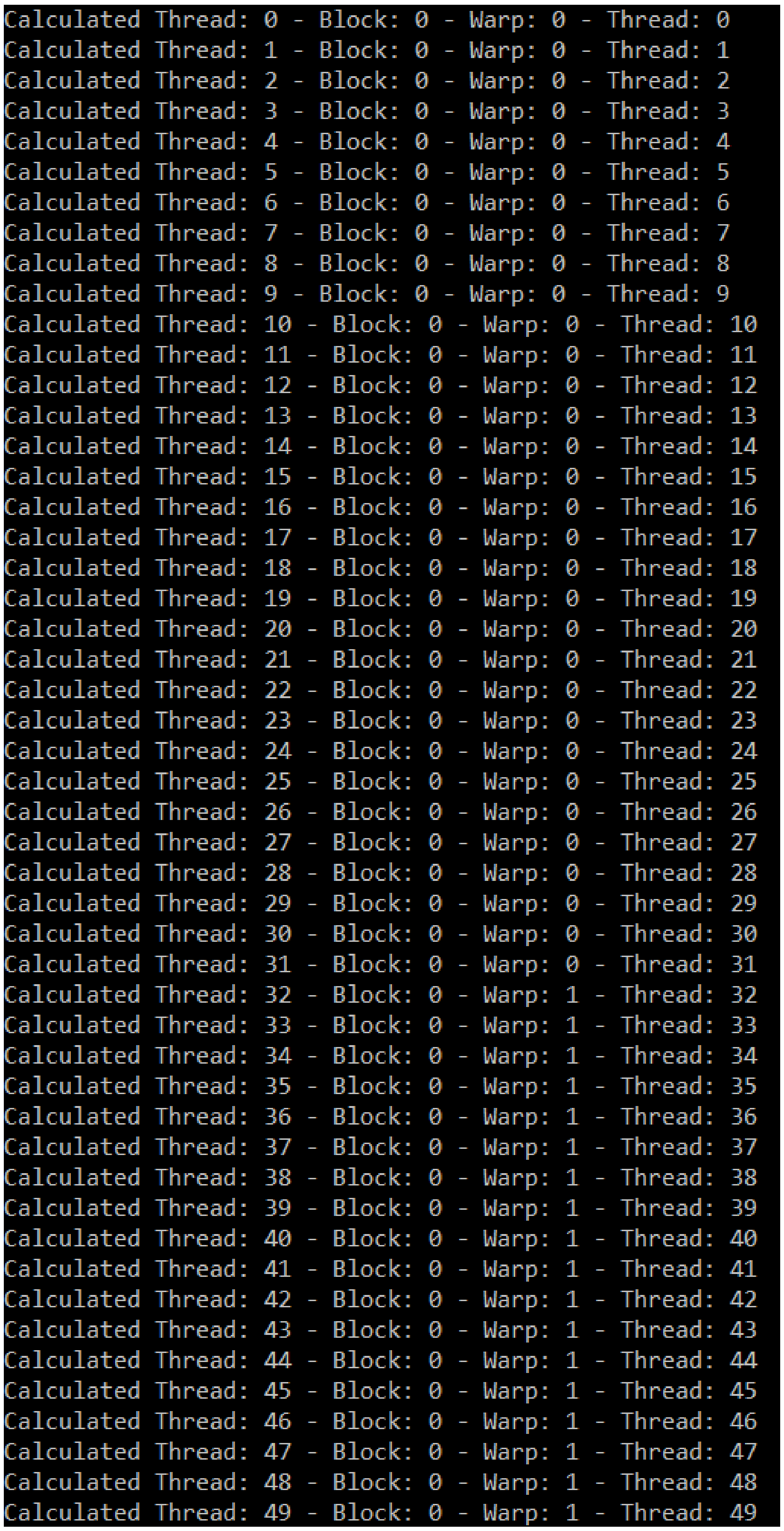
برای نوشتن این برنامه، ابتدا یک struct به نام threadInfo تعریف می گردد که دارای متغیر هایی برای ذخیره اندیس سراسری نخ (globalThread)، شماره بلوک (block)، شماره Warp (warp) و اندیس محلی آن نخ (thread) می باشد؛ سپس در تابع printWithCudaآرایه ای به اندازه تعداد نخ ها در device ایجاد می شود و پس از اینکه مقادیر این آرایه توسط تابع printKernel کرنل تکمیل گردید، مقادیر این آرایه در آرایه موجود در host کپی می شود، و در نهایت مقادیر کپی شده در آرایه موجود در host توسط تابع printThreadInfo به عنوان خروجی پرینت می شوند؛ کد این برنامه عبارت است از:

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

struct threadInfo
{
	int globalThread;
	int block;
	int warp;
	int thread;
};

cudaError_t printWithCuda(threadInfo *t, unsigned int numOfBlock, unsigned int
block_size);
void printThreadInfo(threadInfo * t, int size);

__global__ void printKernel(threadInfo *t)
{
	int globalThread = threadIdx.x + blockIdx.x * blockDim.x;
	int warp = threadIdx.x / warpSize;
	t[globalThread] = { globalThread, blockIdx.x, warp, threadIdx.x };

}

int main()
{
	const int numOfBlock = 2;
	const int block_size = 64;
	int size = numOfBlock * block_size;
	cudaEvent_t start;
	cudaEventCreate(&start);
	cudaEvent_t stop;
	cudaEventCreate(&stop);

	threadInfo *t = (threadInfo*)malloc(sizeof(threadInfo) * size);

	cudaEventRecord(start, NULL);

	// Print vectors in parallel.
	cudaError_t cudaStatus = printWithCuda(t, numOfBlock, block_size);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "addWithCuda failed!");
		return 1;
	}

	// Print Threads info
	printThreadInfo(t, size);

	cudaEventRecord(stop, NULL);

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

	fprintf(stderr, "Elapsed Time is %f ms \n", msecTotal);

	// 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;
}

void printThreadInfo(threadInfo * t, int size) {
	for (int i = 0; i < size; i++)
	{
		printf("Calculated Thread: %d - Block: %d - Warp: %d - Thread: %d \n",
t[i].globalThread, t[i].block, t[i].warp, t[i].thread);
	}

}

// Helper function for using CUDA to add vectors in parallel.
```

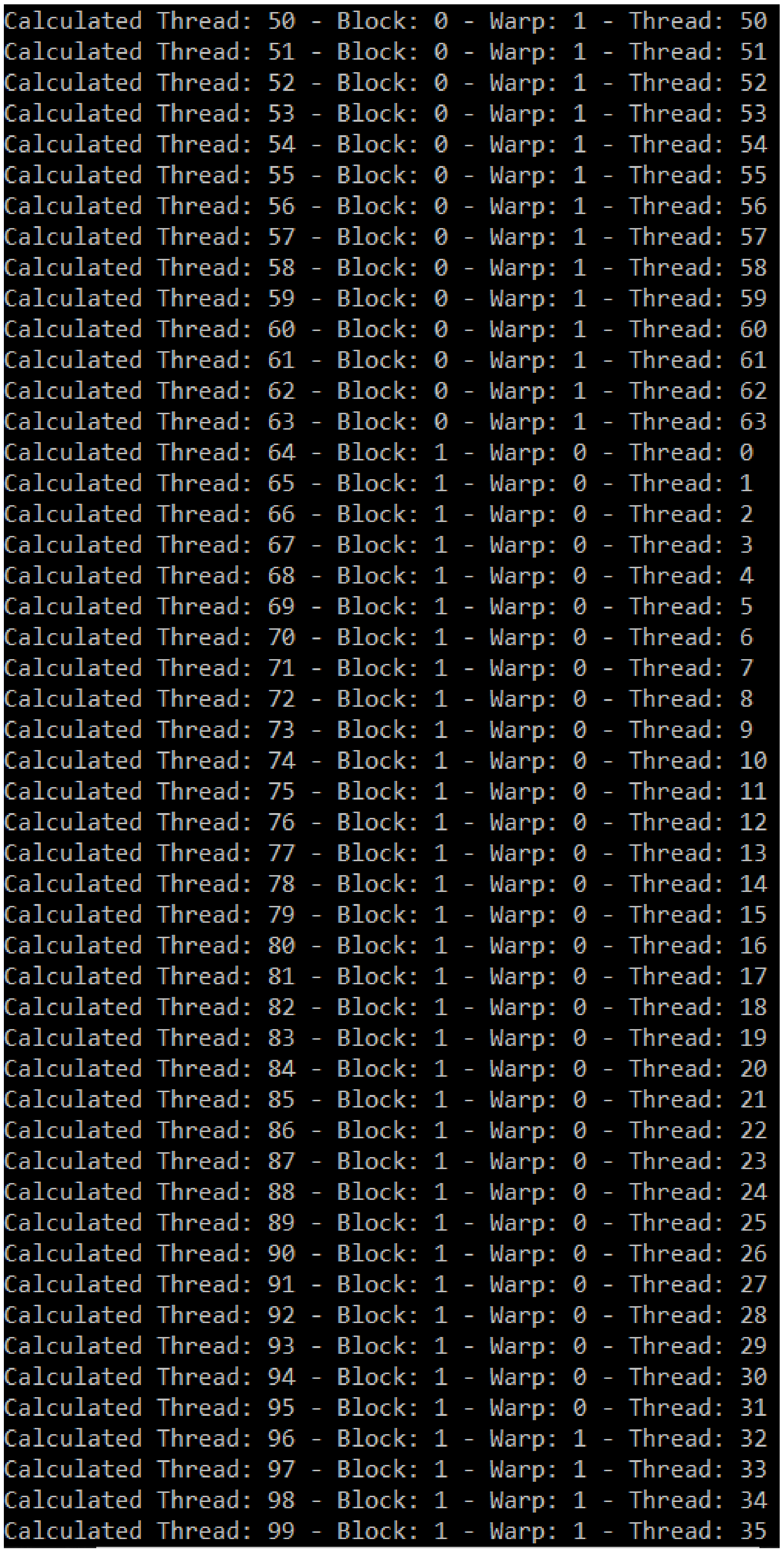
```
cudaError_t printWithCuda(threadInfo *t, unsigned int numOfBlock, unsigned int 
block_size)
{
	threadInfo *dev_t;
	cudaError_t cudaStatus;
	const int size = numOfBlock * block_size;
	// Choose which GPU to run on, change this on a multi-GPU system.
	cudaStatus = cudaSetDevice(0);
	if (cudaStatus != 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_t, size * sizeof(threadInfo));
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMalloc failed!");
		goto Error;
	}

	// Launch a kernel on the GPU with one thread for each element.
	printKernel << < numOfBlock, block_size >> > (dev_t);

	// 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(t, dev_t, size * sizeof(threadInfo), 
cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}

Error:
	cudaFree(dev_t);

	return cudaStatus;
}
```

نمونه خروجی کد بالا برابر است با:

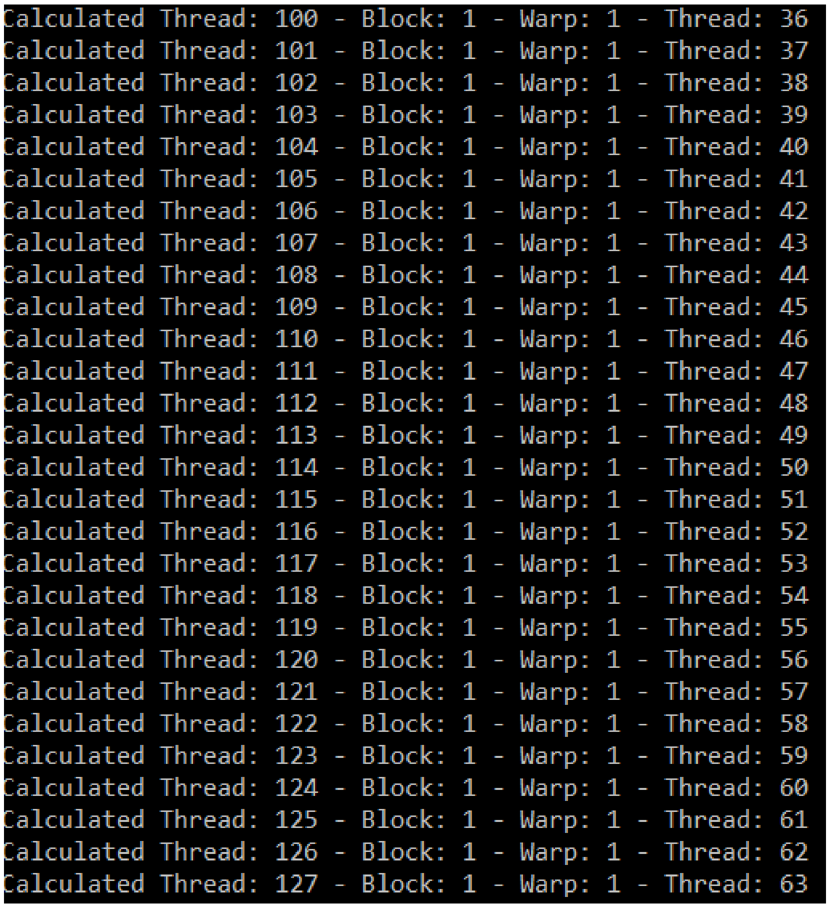
آرش حریرپوش
9731505



```
Calculated Thread: 0 - Block: 0 - Warp: 0 - Thread: 0
Calculated Thread: 1 - Block: 0 - Warp: 0 - Thread: 1
Calculated Thread: 2 - Block: 0 - Warp: 0 - Thread: 2
Calculated Thread: 3 - Block: 0 - Warp: 0 - Thread: 3
Calculated Thread: 4 - Block: 0 - Warp: 0 - Thread: 4
Calculated Thread: 5 - Block: 0 - Warp: 0 - Thread: 5
Calculated Thread: 6 - Block: 0 - Warp: 0 - Thread: 6
Calculated Thread: 7 - Block: 0 - Warp: 0 - Thread: 7
Calculated Thread: 8 - Block: 0 - Warp: 0 - Thread: 8
Calculated Thread: 9 - Block: 0 - Warp: 0 - Thread: 9
Calculated Thread: 10 - Block: 0 - Warp: 0 - Thread: 10
Calculated Thread: 11 - Block: 0 - Warp: 0 - Thread: 11
Calculated Thread: 12 - Block: 0 - Warp: 0 - Thread: 12
Calculated Thread: 13 - Block: 0 - Warp: 0 - Thread: 13
Calculated Thread: 14 - Block: 0 - Warp: 0 - Thread: 14
Calculated Thread: 15 - Block: 0 - Warp: 0 - Thread: 15
Calculated Thread: 16 - Block: 0 - Warp: 0 - Thread: 16
Calculated Thread: 17 - Block: 0 - Warp: 0 - Thread: 17
Calculated Thread: 18 - Block: 0 - Warp: 0 - Thread: 18
Calculated Thread: 19 - Block: 0 - Warp: 0 - Thread: 19
Calculated Thread: 20 - Block: 0 - Warp: 0 - Thread: 20
Calculated Thread: 21 - Block: 0 - Warp: 0 - Thread: 21
Calculated Thread: 22 - Block: 0 - Warp: 0 - Thread: 22
Calculated Thread: 23 - Block: 0 - Warp: 0 - Thread: 23
Calculated Thread: 24 - Block: 0 - Warp: 0 - Thread: 24
Calculated Thread: 25 - Block: 0 - Warp: 0 - Thread: 25
Calculated Thread: 26 - Block: 0 - Warp: 0 - Thread: 26
Calculated Thread: 27 - Block: 0 - Warp: 0 - Thread: 27
Calculated Thread: 28 - Block: 0 - Warp: 0 - Thread: 28
Calculated Thread: 29 - Block: 0 - Warp: 0 - Thread: 29
Calculated Thread: 30 - Block: 0 - Warp: 0 - Thread: 30
Calculated Thread: 31 - Block: 0 - Warp: 0 - Thread: 31
Calculated Thread: 32 - Block: 0 - Warp: 1 - Thread: 32
Calculated Thread: 33 - Block: 0 - Warp: 1 - Thread: 33
Calculated Thread: 34 - Block: 0 - Warp: 1 - Thread: 34
Calculated Thread: 35 - Block: 0 - Warp: 1 - Thread: 35
Calculated Thread: 36 - Block: 0 - Warp: 1 - Thread: 36
Calculated Thread: 37 - Block: 0 - Warp: 1 - Thread: 37
Calculated Thread: 38 - Block: 0 - Warp: 1 - Thread: 38
Calculated Thread: 39 - Block: 0 - Warp: 1 - Thread: 39
Calculated Thread: 40 - Block: 0 - Warp: 1 - Thread: 40
Calculated Thread: 41 - Block: 0 - Warp: 1 - Thread: 41
Calculated Thread: 42 - Block: 0 - Warp: 1 - Thread: 42
Calculated Thread: 43 - Block: 0 - Warp: 1 - Thread: 43
Calculated Thread: 44 - Block: 0 - Warp: 1 - Thread: 44
Calculated Thread: 45 - Block: 0 - Warp: 1 - Thread: 45
Calculated Thread: 46 - Block: 0 - Warp: 1 - Thread: 46
Calculated Thread: 47 - Block: 0 - Warp: 1 - Thread: 47
Calculated Thread: 48 - Block: 0 - Warp: 1 - Thread: 48
Calculated Thread: 49 - Block: 0 - Warp: 1 - Thread: 49
```



```
Calculated Thread: 50 - Block: 0 - Warp: 1 - Thread: 50
Calculated Thread: 51 - Block: 0 - Warp: 1 - Thread: 51
Calculated Thread: 52 - Block: 0 - Warp: 1 - Thread: 52
Calculated Thread: 53 - Block: 0 - Warp: 1 - Thread: 53
Calculated Thread: 54 - Block: 0 - Warp: 1 - Thread: 54
Calculated Thread: 55 - Block: 0 - Warp: 1 - Thread: 55
Calculated Thread: 56 - Block: 0 - Warp: 1 - Thread: 56
Calculated Thread: 57 - Block: 0 - Warp: 1 - Thread: 57
Calculated Thread: 58 - Block: 0 - Warp: 1 - Thread: 58
Calculated Thread: 59 - Block: 0 - Warp: 1 - Thread: 59
Calculated Thread: 60 - Block: 0 - Warp: 1 - Thread: 60
Calculated Thread: 61 - Block: 0 - Warp: 1 - Thread: 61
Calculated Thread: 62 - Block: 0 - Warp: 1 - Thread: 62
Calculated Thread: 63 - Block: 0 - Warp: 1 - Thread: 63
Calculated Thread: 64 - Block: 1 - Warp: 0 - Thread: 0
Calculated Thread: 65 - Block: 1 - Warp: 0 - Thread: 1
Calculated Thread: 66 - Block: 1 - Warp: 0 - Thread: 2
Calculated Thread: 67 - Block: 1 - Warp: 0 - Thread: 3
Calculated Thread: 68 - Block: 1 - Warp: 0 - Thread: 4
Calculated Thread: 69 - Block: 1 - Warp: 0 - Thread: 5
Calculated Thread: 70 - Block: 1 - Warp: 0 - Thread: 6
Calculated Thread: 71 - Block: 1 - Warp: 0 - Thread: 7
Calculated Thread: 72 - Block: 1 - Warp: 0 - Thread: 8
Calculated Thread: 73 - Block: 1 - Warp: 0 - Thread: 9
Calculated Thread: 74 - Block: 1 - Warp: 0 - Thread: 10
Calculated Thread: 75 - Block: 1 - Warp: 0 - Thread: 11
Calculated Thread: 76 - Block: 1 - Warp: 0 - Thread: 12
Calculated Thread: 77 - Block: 1 - Warp: 0 - Thread: 13
Calculated Thread: 78 - Block: 1 - Warp: 0 - Thread: 14
Calculated Thread: 79 - Block: 1 - Warp: 0 - Thread: 15
Calculated Thread: 80 - Block: 1 - Warp: 0 - Thread: 16
Calculated Thread: 81 - Block: 1 - Warp: 0 - Thread: 17
Calculated Thread: 82 - Block: 1 - Warp: 0 - Thread: 18
Calculated Thread: 83 - Block: 1 - Warp: 0 - Thread: 19
Calculated Thread: 84 - Block: 1 - Warp: 0 - Thread: 20
Calculated Thread: 85 - Block: 1 - Warp: 0 - Thread: 21
Calculated Thread: 86 - Block: 1 - Warp: 0 - Thread: 22
Calculated Thread: 87 - Block: 1 - Warp: 0 - Thread: 23
Calculated Thread: 88 - Block: 1 - Warp: 0 - Thread: 24
Calculated Thread: 89 - Block: 1 - Warp: 0 - Thread: 25
Calculated Thread: 90 - Block: 1 - Warp: 0 - Thread: 26
Calculated Thread: 91 - Block: 1 - Warp: 0 - Thread: 27
Calculated Thread: 92 - Block: 1 - Warp: 0 - Thread: 28
Calculated Thread: 93 - Block: 1 - Warp: 0 - Thread: 29
Calculated Thread: 94 - Block: 1 - Warp: 0 - Thread: 30
Calculated Thread: 95 - Block: 1 - Warp: 0 - Thread: 31
Calculated Thread: 96 - Block: 1 - Warp: 1 - Thread: 32
Calculated Thread: 97 - Block: 1 - Warp: 1 - Thread: 33
Calculated Thread: 98 - Block: 1 - Warp: 1 - Thread: 34
Calculated Thread: 99 - Block: 1 - Warp: 1 - Thread: 35
```



```
Calculated Thread: 100 - Block: 1 - Warp: 1 - Thread: 36
Calculated Thread: 101 - Block: 1 - Warp: 1 - Thread: 37
Calculated Thread: 102 - Block: 1 - Warp: 1 - Thread: 38
Calculated Thread: 103 - Block: 1 - Warp: 1 - Thread: 39
Calculated Thread: 104 - Block: 1 - Warp: 1 - Thread: 40
Calculated Thread: 105 - Block: 1 - Warp: 1 - Thread: 41
Calculated Thread: 106 - Block: 1 - Warp: 1 - Thread: 42
Calculated Thread: 107 - Block: 1 - Warp: 1 - Thread: 43
Calculated Thread: 108 - Block: 1 - Warp: 1 - Thread: 44
Calculated Thread: 109 - Block: 1 - Warp: 1 - Thread: 45
Calculated Thread: 110 - Block: 1 - Warp: 1 - Thread: 46
Calculated Thread: 111 - Block: 1 - Warp: 1 - Thread: 47
Calculated Thread: 112 - Block: 1 - Warp: 1 - Thread: 48
Calculated Thread: 113 - Block: 1 - Warp: 1 - Thread: 49
Calculated Thread: 114 - Block: 1 - Warp: 1 - Thread: 50
Calculated Thread: 115 - Block: 1 - Warp: 1 - Thread: 51
Calculated Thread: 116 - Block: 1 - Warp: 1 - Thread: 52
Calculated Thread: 117 - Block: 1 - Warp: 1 - Thread: 53
Calculated Thread: 118 - Block: 1 - Warp: 1 - Thread: 54
Calculated Thread: 119 - Block: 1 - Warp: 1 - Thread: 55
Calculated Thread: 120 - Block: 1 - Warp: 1 - Thread: 56
Calculated Thread: 121 - Block: 1 - Warp: 1 - Thread: 57
Calculated Thread: 122 - Block: 1 - Warp: 1 - Thread: 58
Calculated Thread: 123 - Block: 1 - Warp: 1 - Thread: 59
Calculated Thread: 124 - Block: 1 - Warp: 1 - Thread: 60
Calculated Thread: 125 - Block: 1 - Warp: 1 - Thread: 61
Calculated Thread: 126 - Block: 1 - Warp: 1 - Thread: 62
Calculated Thread: 127 - Block: 1 - Warp: 1 - Thread: 63
```

(همچنین کد این بخش در فایل MattAdd_step4.cu به پیوست قرار داده شده است.)