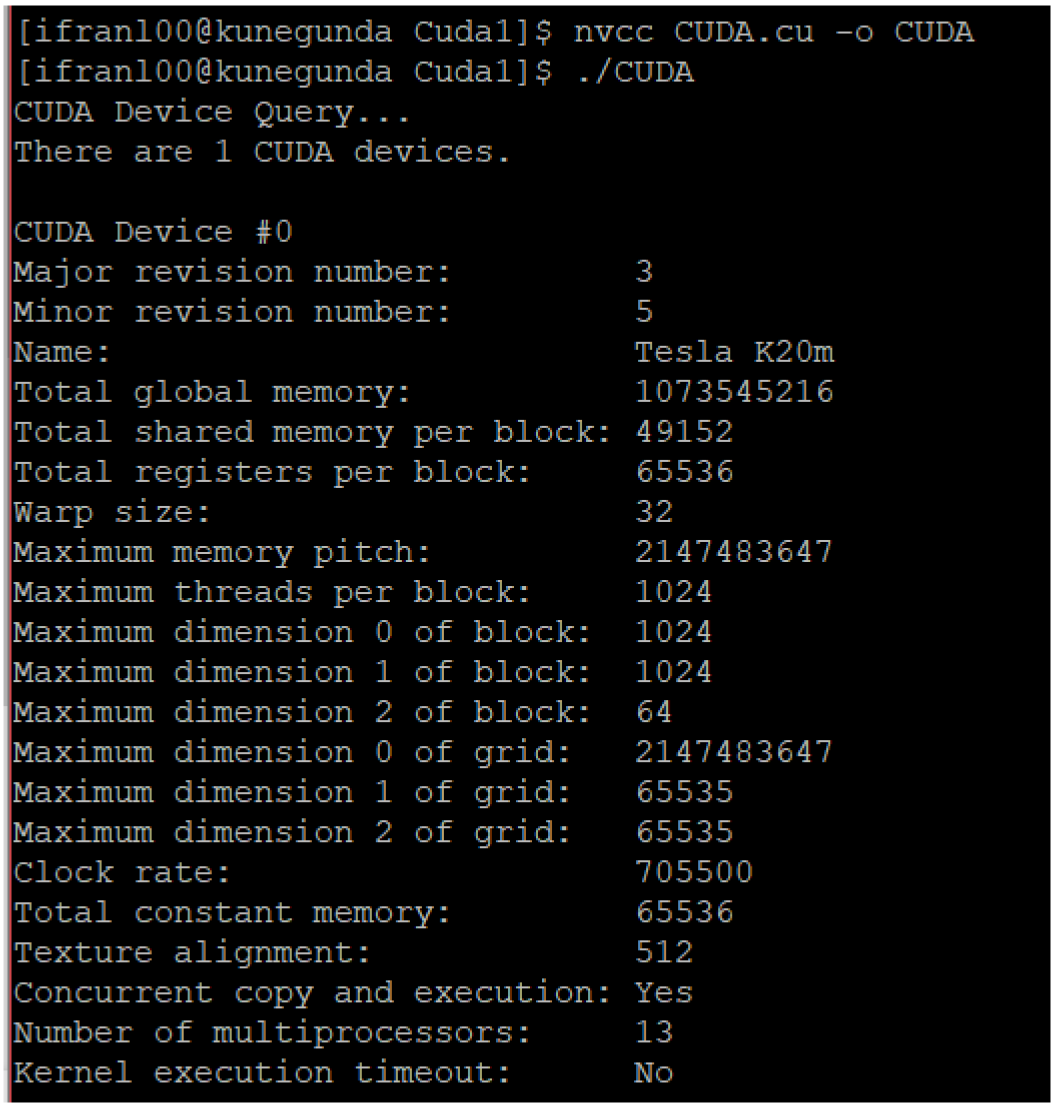
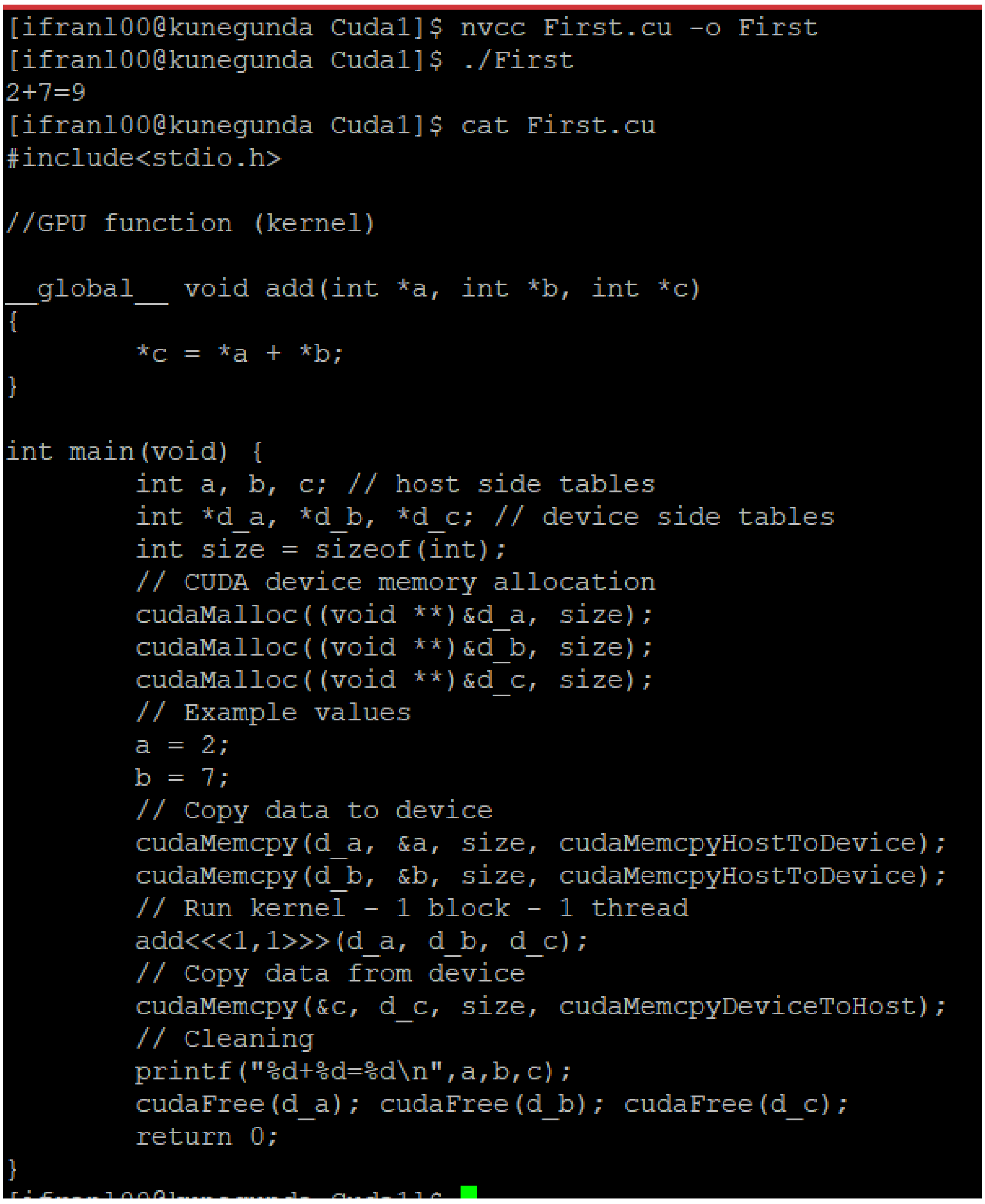
CUDA LABORATORY

1. ***Compile (nvcc) and run the program that shows information about the available CUDA device.***
Familiarize yourself with the device parameters.



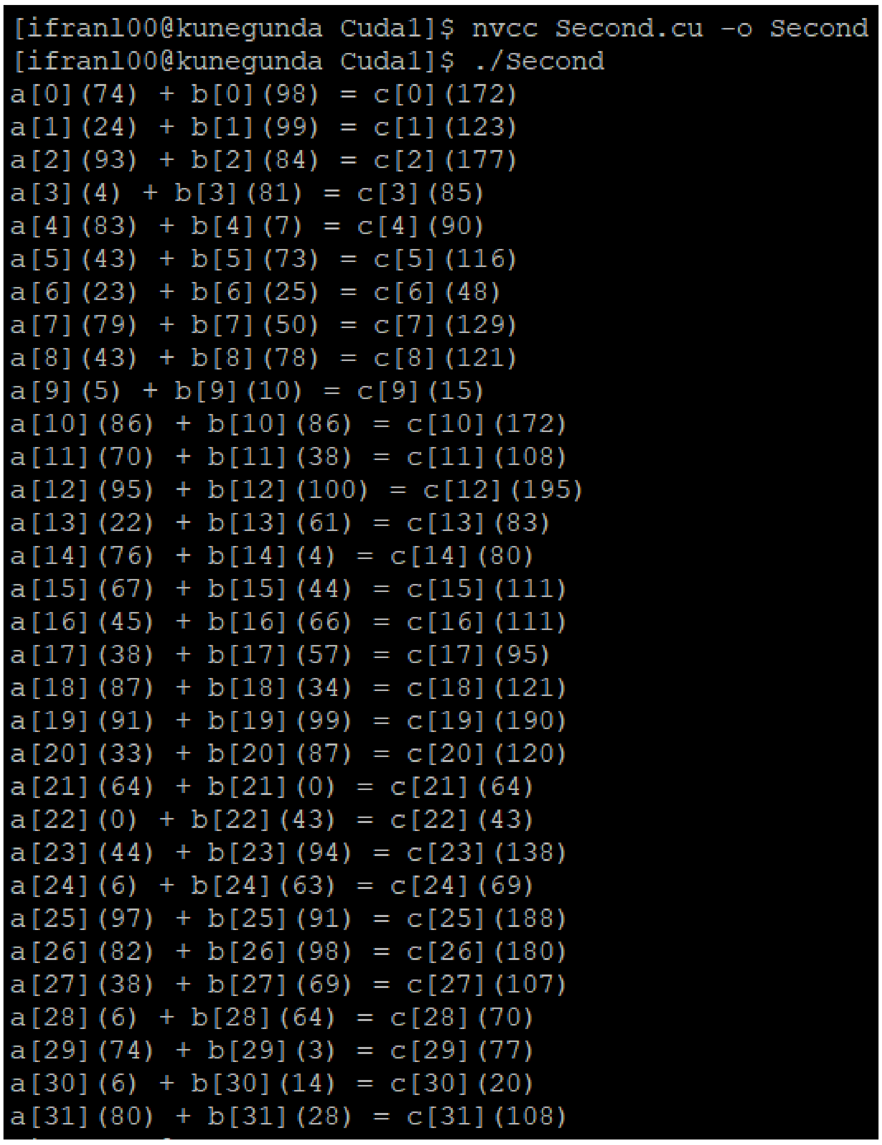
```
[ifranl00@kunegunda Cuda1]$ nvcc CUDA.cu -o CUDA
[ifranl00@kunegunda Cuda1]$ ./CUDA
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         3
Minor revision number:         5
Name:                          Tesla K20m
Total global memory:           1073545216
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    705500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     13
Kernel execution timeout:      No
```

2.. *Familiarize yourself with the sample source code that runs on the graphics card.*

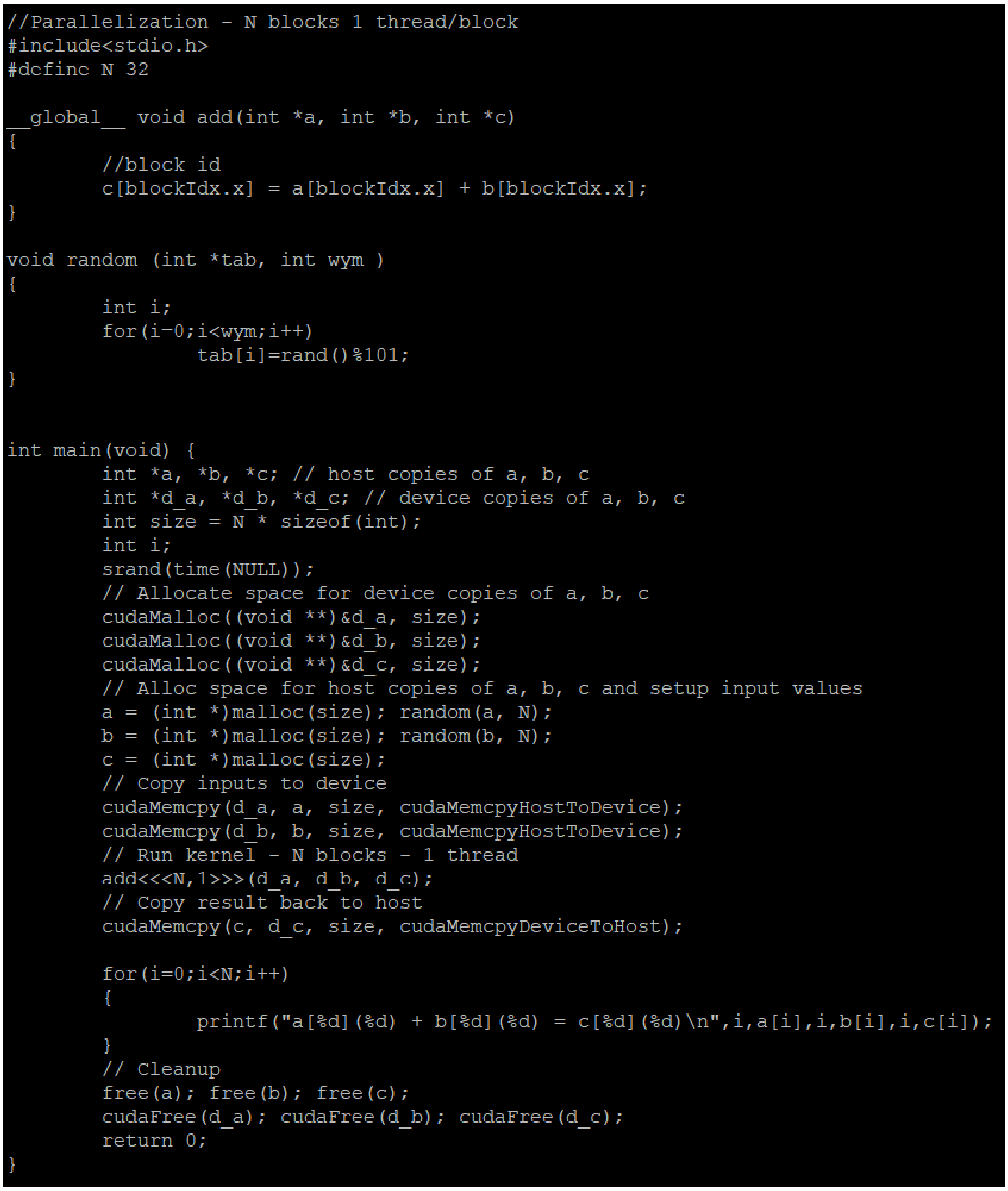


```
[ifranl00@kunegunda Cuda1]$ nvcc First.cu -o First
[ifranl00@kunegunda Cuda1]$ ./First
2+7=9
[ifranl00@kunegunda Cuda1]$ cat First.cu
#include<stdio.h>

//GPU function (kernel)

__global__ void add(int *a, int *b, int *c)
{
        *c = *a + *b;
}

int main(void) {
        int a, b, c; // host side tables
        int *d_a, *d_b, *d_c; // device side tables
        int size = sizeof(int);
        // CUDA device memory allocation
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        // Example values
        a = 2;
        b = 7;
        // Copy data to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
        // Run kernel - 1 block - 1 thread
        add<<<1,1>>>(d_a, d_b, d_c);
        // Copy data from device
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
        // Cleaning
        printf("%d+%d=%d\n",a,b,c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

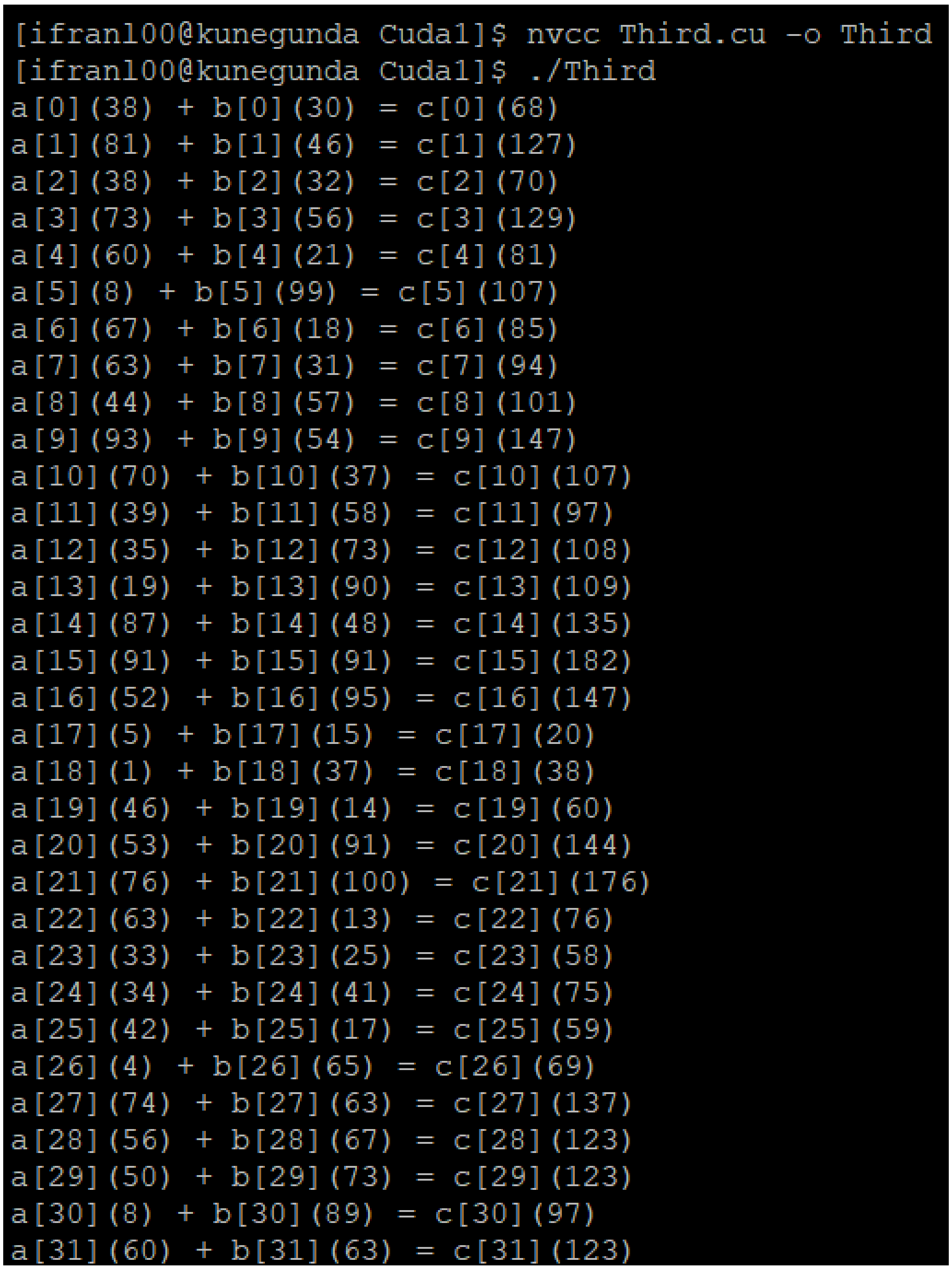
3.*. Familiarize yourself with two levels of parallelization - through blocks and through threads.*



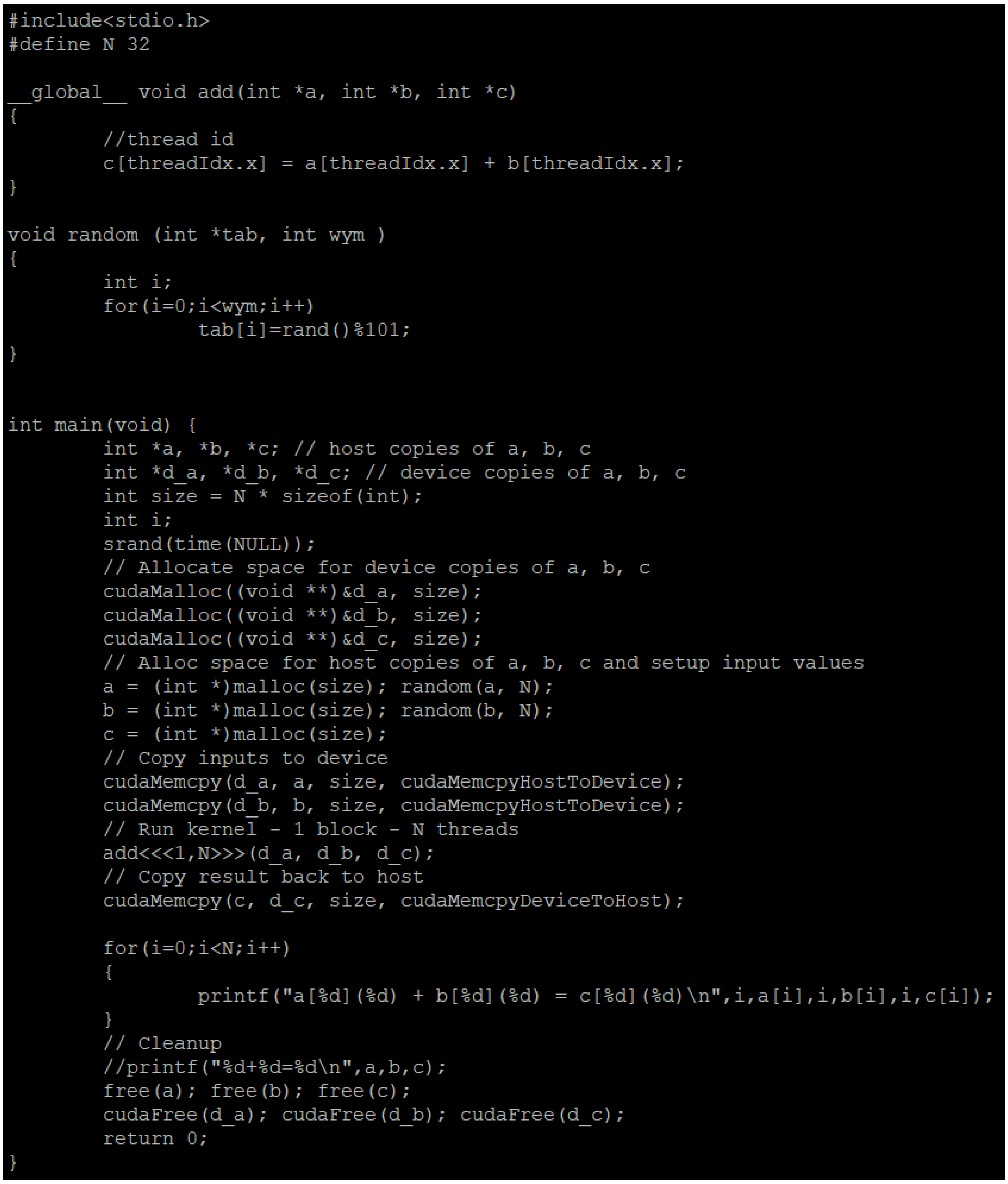
```
[ifranl00@kunegunda Cuda1]$ nvcc Second.cu -o Second
[ifranl00@kunegunda Cuda1]$ ./Second
a[0](74) + b[0](98) = c[0](172)
a[1](24) + b[1](99) = c[1](123)
a[2](93) + b[2](84) = c[2](177)
a[3](4) + b[3](81) = c[3](85)
a[4](83) + b[4](7) = c[4](90)
a[5](43) + b[5](73) = c[5](116)
a[6](23) + b[6](25) = c[6](48)
a[7](79) + b[7](50) = c[7](129)
a[8](43) + b[8](78) = c[8](121)
a[9](5) + b[9](10) = c[9](15)
a[10](86) + b[10](86) = c[10](172)
a[11](70) + b[11](38) = c[11](108)
a[12](95) + b[12](100) = c[12](195)
a[13](22) + b[13](61) = c[13](83)
a[14](76) + b[14](4) = c[14](80)
a[15](67) + b[15](44) = c[15](111)
a[16](45) + b[16](66) = c[16](111)
a[17](38) + b[17](57) = c[17](95)
a[18](87) + b[18](34) = c[18](121)
a[19](91) + b[19](99) = c[19](190)
a[20](33) + b[20](87) = c[20](120)
a[21](64) + b[21](0) = c[21](64)
a[22](0) + b[22](43) = c[22](43)
a[23](44) + b[23](94) = c[23](138)
a[24](6) + b[24](63) = c[24](69)
a[25](97) + b[25](91) = c[25](188)
a[26](82) + b[26](98) = c[26](180)
a[27](38) + b[27](69) = c[27](107)
a[28](6) + b[28](64) = c[28](70)
a[29](74) + b[29](3) = c[29](77)
a[30](6) + b[30](14) = c[30](20)
a[31](80) + b[31](28) = c[31](108)
```



```
//Parallelization - N blocks 1 thread/block
#include<stdio.h>
#define N 32

__global__ void add(int *a, int *b, int *c)
{
        //block id
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

void random (int *tab, int wym )
{
        int i;
        for(i=0;i<wym;i++)
                tab[i]=rand()%101;
}


int main(void) {
        int *a, *b, *c; // host copies of a, b, c
        int *d_a, *d_b, *d_c; // device copies of a, b, c
        int size = N * sizeof(int);
        int i;
        srand(time(NULL));
        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random(a, N);
        b = (int *)malloc(size); random(b, N);
        c = (int *)malloc(size);
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        // Run kernel - N blocks - 1 thread
        add<<<N,1>>>(d_a, d_b, d_c);
        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        for(i=0;i<N;i++)
        {
                printf("a[%d](%d) + b[%d](%d) = c[%d](%d)\n",i,a[i],i,b[i],i,c[i]);
        }
        // Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```



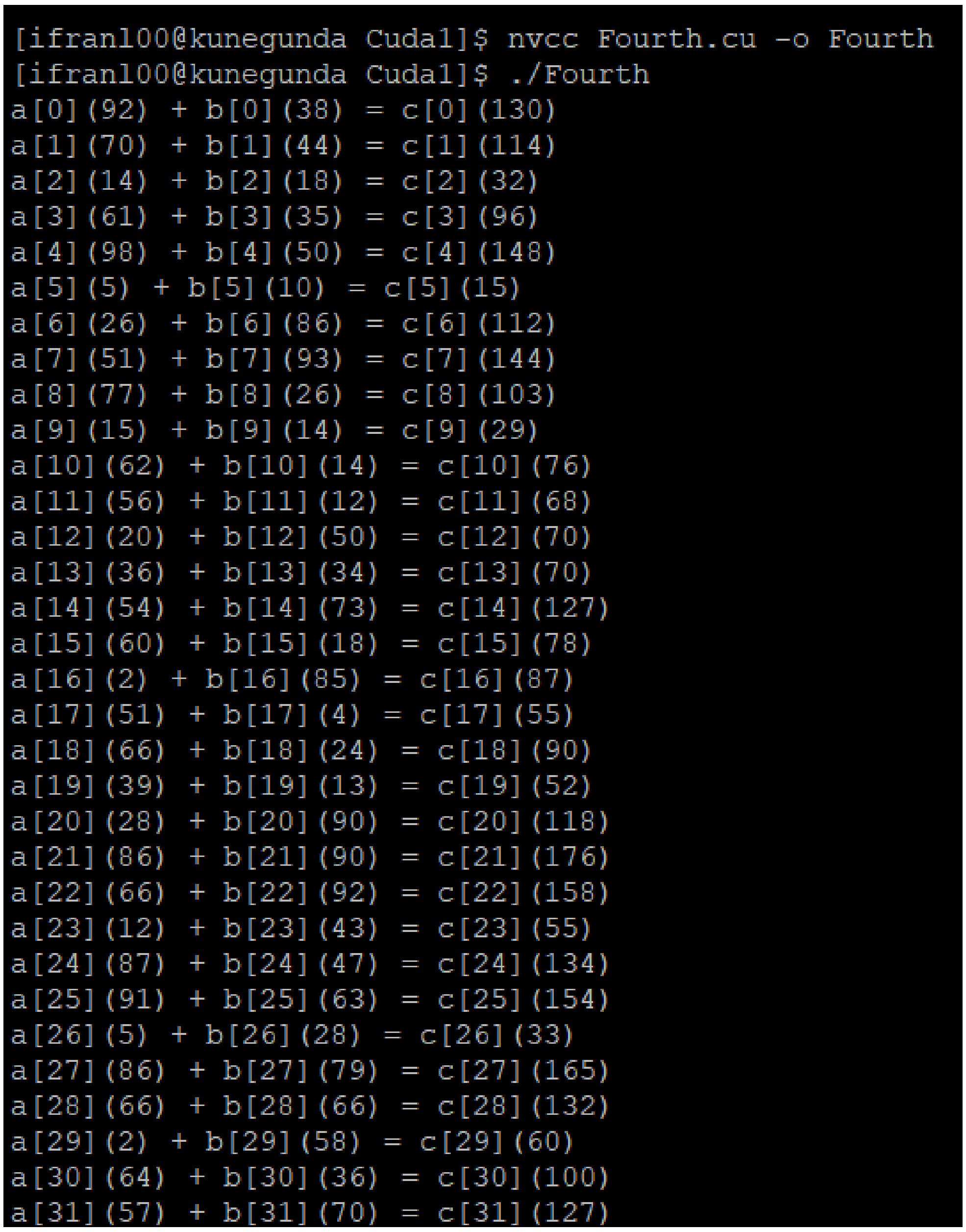
```
[ifranl00@kunegunda Cuda1]$ nvcc Third.cu -o Third
[ifranl00@kunegunda Cuda1]$ ./Third
a[0](38) + b[0](30) = c[0](68)
a[1](81) + b[1](46) = c[1](127)
a[2](38) + b[2](32) = c[2](70)
a[3](73) + b[3](56) = c[3](129)
a[4](60) + b[4](21) = c[4](81)
a[5](8) + b[5](99) = c[5](107)
a[6](67) + b[6](18) = c[6](85)
a[7](63) + b[7](31) = c[7](94)
a[8](44) + b[8](57) = c[8](101)
a[9](93) + b[9](54) = c[9](147)
a[10](70) + b[10](37) = c[10](107)
a[11](39) + b[11](58) = c[11](97)
a[12](35) + b[12](73) = c[12](108)
a[13](19) + b[13](90) = c[13](109)
a[14](87) + b[14](48) = c[14](135)
a[15](91) + b[15](91) = c[15](182)
a[16](52) + b[16](95) = c[16](147)
a[17](5) + b[17](15) = c[17](20)
a[18](1) + b[18](37) = c[18](38)
a[19](46) + b[19](14) = c[19](60)
a[20](53) + b[20](91) = c[20](144)
a[21](76) + b[21](100) = c[21](176)
a[22](63) + b[22](13) = c[22](76)
a[23](33) + b[23](25) = c[23](58)
a[24](34) + b[24](41) = c[24](75)
a[25](42) + b[25](17) = c[25](59)
a[26](4) + b[26](65) = c[26](69)
a[27](74) + b[27](63) = c[27](137)
a[28](56) + b[28](67) = c[28](123)
a[29](50) + b[29](73) = c[29](123)
a[30](8) + b[30](89) = c[30](97)
a[31](60) + b[31](63) = c[31](123)
```



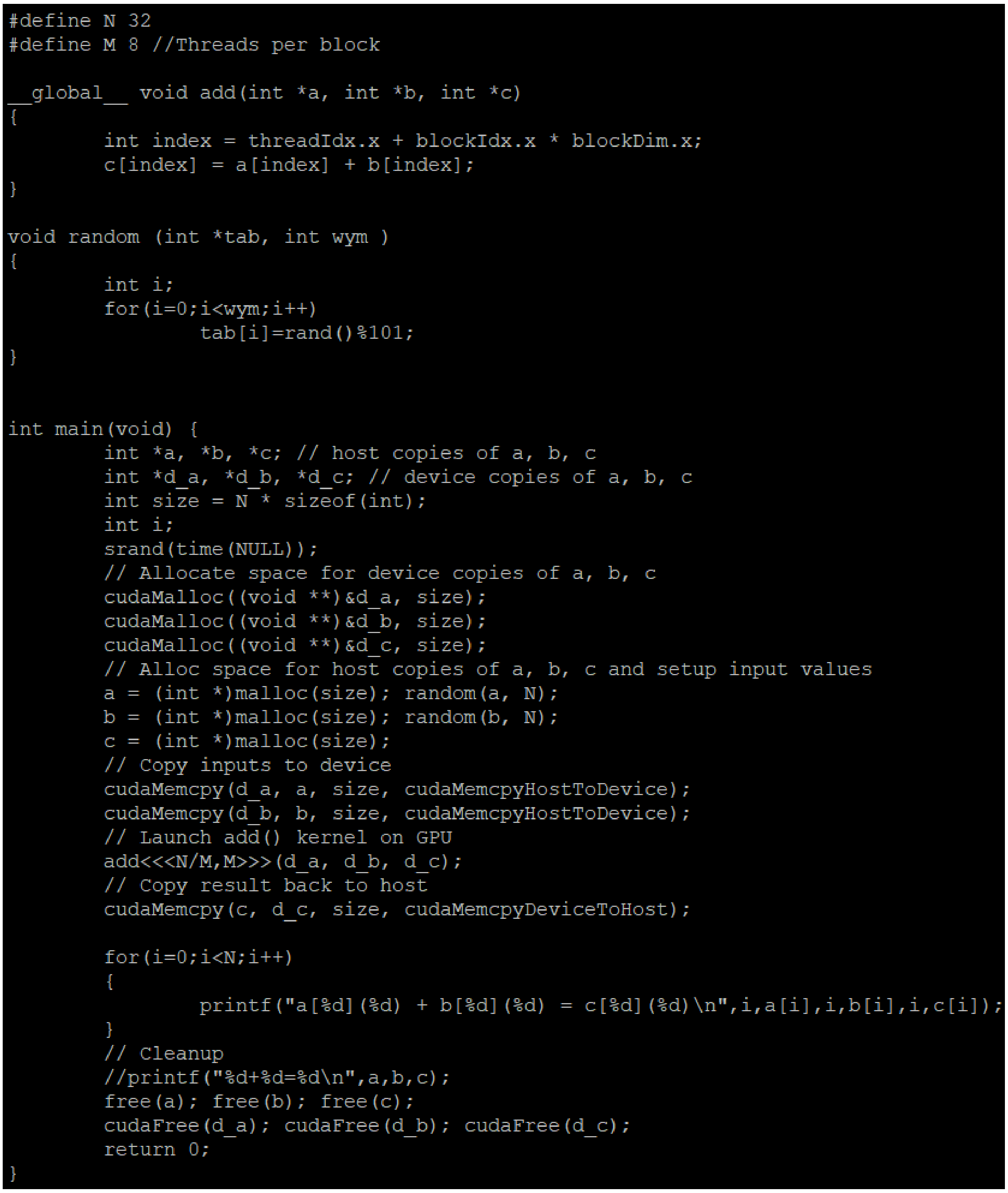
```
#include<stdio.h>
#define N 32

__global__ void add(int *a, int *b, int *c)
{
        //thread id
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

void random (int *tab, int wym )
{
        int i;
        for(i=0;i<wym;i++)
                tab[i]=rand()%101;
}


int main(void) {
        int *a, *b, *c; // host copies of a, b, c
        int *d_a, *d_b, *d_c; // device copies of a, b, c
        int size = N * sizeof(int);
        int i;
        srand(time(NULL));
        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random(a, N);
        b = (int *)malloc(size); random(b, N);
        c = (int *)malloc(size);
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        // Run kernel - 1 block - N threads
        add<<<1,N>>>(d_a, d_b, d_c);
        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        for(i=0;i<N;i++)
        {
                printf("a[%d](%d) + b[%d](%d) = c[%d](%d)\n",i,a[i],i,b[i],i,c[i]);
        }
        // Cleanup
        //printf("%d+%d=%d\n",a,b,c);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

4.. *Compare and analyze the division of work in two programs - with the vector size divided by the number of blocks and with the size indivisible by the number of blocks.*

In the first case, the vector size is 32 and the number of threads is 8 so the division of work by blocks is easily made.

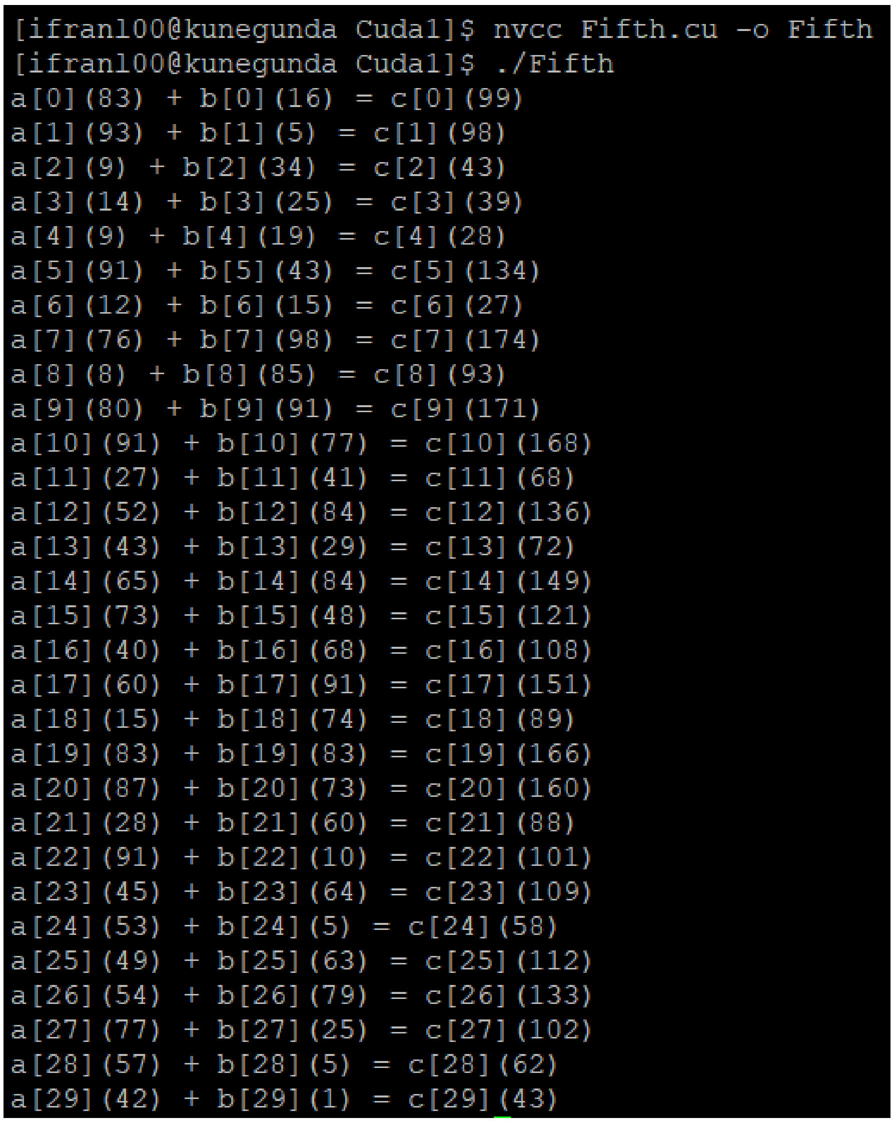


```
[ifranl00@kunegunda Cuda1]$ nvcc Fourth.cu -o Fourth
[ifranl00@kunegunda Cuda1]$ ./Fourth
a[0](92) + b[0](38) = c[0](130)
a[1](70) + b[1](44) = c[1](114)
a[2](14) + b[2](18) = c[2](32)
a[3](61) + b[3](35) = c[3](96)
a[4](98) + b[4](50) = c[4](148)
a[5](5) + b[5](10) = c[5](15)
a[6](26) + b[6](86) = c[6](112)
a[7](51) + b[7](93) = c[7](144)
a[8](77) + b[8](26) = c[8](103)
a[9](15) + b[9](14) = c[9](29)
a[10](62) + b[10](14) = c[10](76)
a[11](56) + b[11](12) = c[11](68)
a[12](20) + b[12](50) = c[12](70)
a[13](36) + b[13](34) = c[13](70)
a[14](54) + b[14](73) = c[14](127)
a[15](60) + b[15](18) = c[15](78)
a[16](2) + b[16](85) = c[16](87)
a[17](51) + b[17](4) = c[17](55)
a[18](66) + b[18](24) = c[18](90)
a[19](39) + b[19](13) = c[19](52)
a[20](28) + b[20](90) = c[20](118)
a[21](86) + b[21](90) = c[21](176)
a[22](66) + b[22](92) = c[22](158)
a[23](12) + b[23](43) = c[23](55)
a[24](87) + b[24](47) = c[24](134)
a[25](91) + b[25](63) = c[25](154)
a[26](5) + b[26](28) = c[26](33)
a[27](86) + b[27](79) = c[27](165)
a[28](66) + b[28](66) = c[28](132)
a[29](2) + b[29](58) = c[29](60)
a[30](64) + b[30](36) = c[30](100)
a[31](57) + b[31](70) = c[31](127)
```

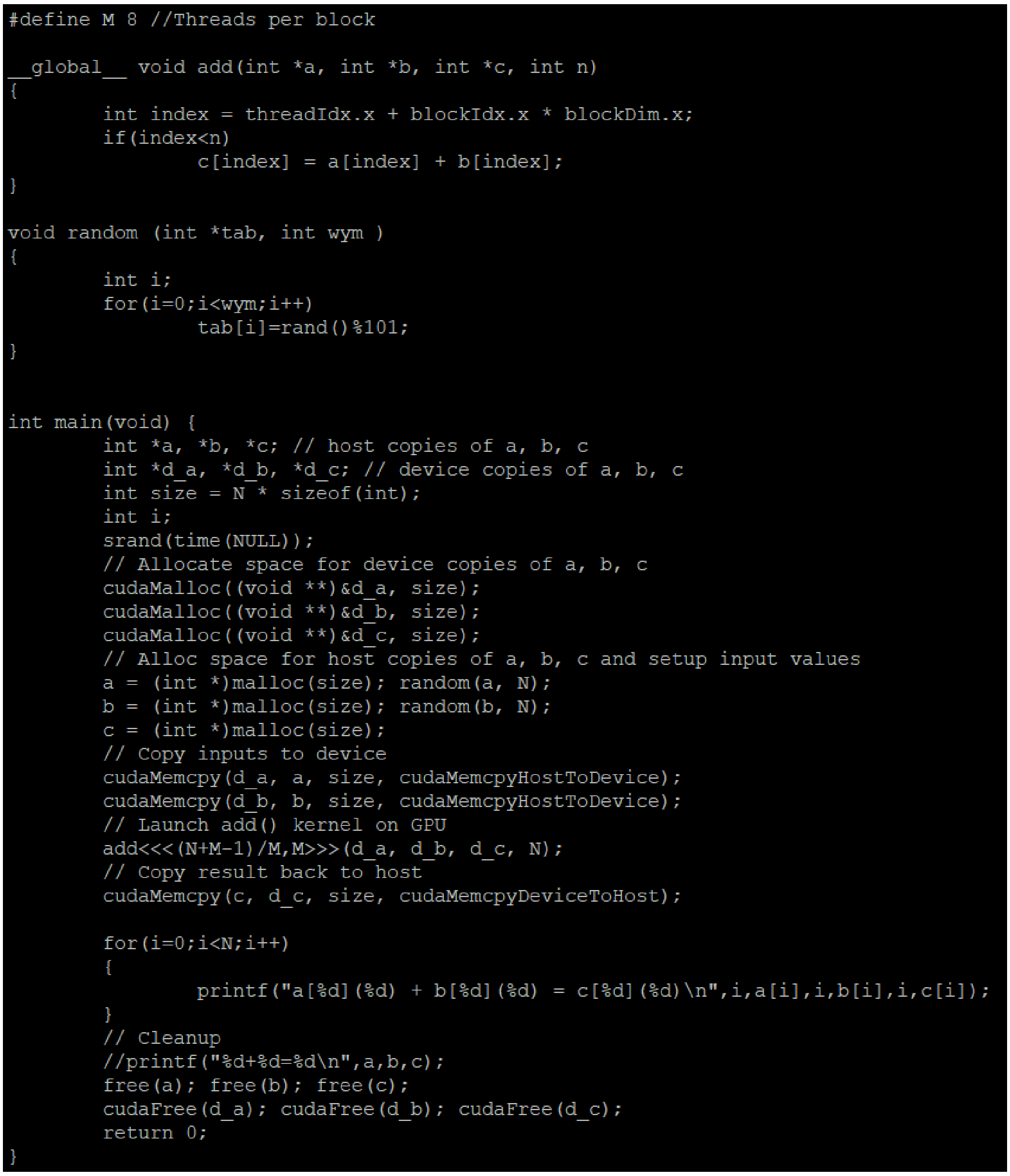


```
#define N 32
#define M 8 //Threads per block

__global__ void add(int *a, int *b, int *c)
{
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}

void random (int *tab, int wym )
{
        int i;
        for(i=0;i<wym;i++)
                tab[i]=rand()%101;
}


int main(void) {
        int *a, *b, *c; // host copies of a, b, c
        int *d_a, *d_b, *d_c; // device copies of a, b, c
        int size = N * sizeof(int);
        int i;
        srand(time(NULL));
        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random(a, N);
        b = (int *)malloc(size); random(b, N);
        c = (int *)malloc(size);
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        // Launch add() kernel on GPU
        add<<<N/M,M>>>(d_a, d_b, d_c);
        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        for(i=0;i<N;i++)
        {
                printf("a[%d](%d) + b[%d](%d) = c[%d](%d)\n",i,a[i],i,b[i],i,c[i]);
        }
        // Cleanup
        //printf("%d+%d=%d\n",a,b,c);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

In the second case, the vector size is 30 and the number of threads per block is 8, because 30 is not divisible by 8 this program implements a balance.



```
[ifranl00@kunegunda Cuda1]$ nvcc Fifth.cu -o Fifth
[ifranl00@kunegunda Cuda1]$ ./Fifth
a[0](83) + b[0](16) = c[0](99)
a[1](93) + b[1](5) = c[1](98)
a[2](9) + b[2](34) = c[2](43)
a[3](14) + b[3](25) = c[3](39)
a[4](9) + b[4](19) = c[4](28)
a[5](91) + b[5](43) = c[5](134)
a[6](12) + b[6](15) = c[6](27)
a[7](76) + b[7](98) = c[7](174)
a[8](8) + b[8](85) = c[8](93)
a[9](80) + b[9](91) = c[9](171)
a[10](91) + b[10](77) = c[10](168)
a[11](27) + b[11](41) = c[11](68)
a[12](52) + b[12](84) = c[12](136)
a[13](43) + b[13](29) = c[13](72)
a[14](65) + b[14](84) = c[14](149)
a[15](73) + b[15](48) = c[15](121)
a[16](40) + b[16](68) = c[16](108)
a[17](60) + b[17](91) = c[17](151)
a[18](15) + b[18](74) = c[18](89)
a[19](83) + b[19](83) = c[19](166)
a[20](87) + b[20](73) = c[20](160)
a[21](28) + b[21](60) = c[21](88)
a[22](91) + b[22](10) = c[22](101)
a[23](45) + b[23](64) = c[23](109)
a[24](53) + b[24](5) = c[24](58)
a[25](49) + b[25](63) = c[25](112)
a[26](54) + b[26](79) = c[26](133)
a[27](77) + b[27](25) = c[27](102)
a[28](57) + b[28](5) = c[28](62)
a[29](42) + b[29](1) = c[29](43)
```



```
#define M 8 //Threads per block

__global__ void add(int *a, int *b, int *c, int n)
{
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        if(index<n)
                c[index] = a[index] + b[index];
}

void random (int *tab, int wym )
{
        int i;
        for(i=0;i<wym;i++)
                tab[i]=rand()%101;
}


int main(void) {
        int *a, *b, *c; // host copies of a, b, c
        int *d_a, *d_b, *d_c; // device copies of a, b, c
        int size = N * sizeof(int);
        int i;
        srand(time(NULL));
        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); random(a, N);
        b = (int *)malloc(size); random(b, N);
        c = (int *)malloc(size);
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        // Launch add() kernel on GPU
        add<<<(N+M-1)/M,M>>>(d_a, d_b, d_c, N);
        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        for(i=0;i<N;i++)
        {
                printf("a[%d](%d) + b[%d](%d) = c[%d](%d)\n",i,a[i],i,b[i],i,c[i]);
        }
        // Cleanup
        //printf("%d+%d=%d\n",a,b,c);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Let’s compare both programs:

The differences between them are the kernel code and how the kernel is launched.

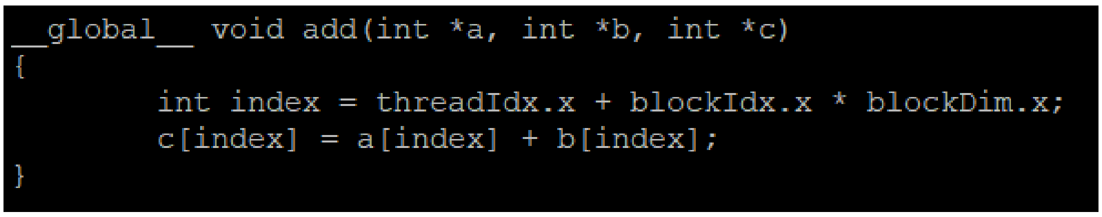
1. Device code or kernel code.

In the first case where there is no need to do balance so only three pointers are passed.

Both of them will use predefined variables of CUDA: blockIndx and blockDim and threadIndx where:

- blockIdx contains the thread block within the grid.
- blockDim is the second parameter passed in the kernel launch and it contains the dimension of each thread block.
- threadIdx contains the index of the thread in its thread block.

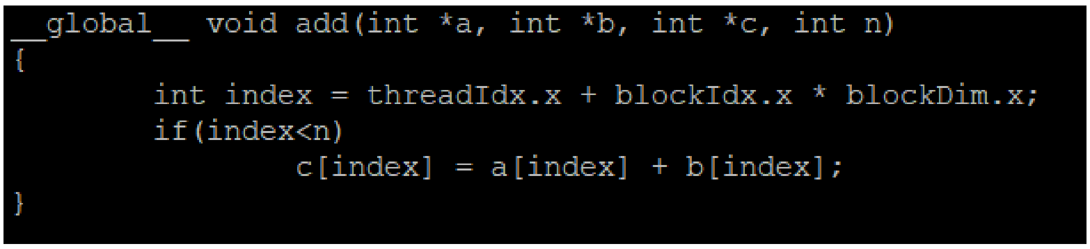
Also, both of programs will save in int index the global index to access the data and do the addition.



```
__global__ void add(int *a, int *b, int *c)
{
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}
```

In the other program a lot of lines are the same but more will be needed for do the balance, in this case another variable is passed name n.

The n variable is used to make a condition to not make an access to something that it is outside the bounds because in this case the number of elements of the vector is not divisible by the number of threads per block.



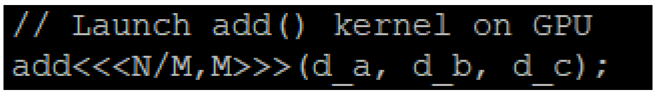
```
__global__ void add(int *a, int *b, int *c, int n)
{
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        if(index<n)
                c[index] = a[index] + b[index];
}
```

2. Kernel launch.

In both programs, kernel lauch is specified and the first argument in the execution configuration specifies the number of thread blocks in the grid, and the second specifies the number of threads in a thread block.

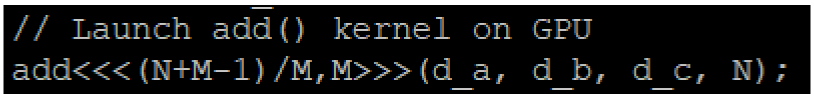
So for both cases, the second parameter does not need to be different.

In Fourth.cu the number of thread block in the grid will be the simple division of N/M because is a exact division with 0 as reminder.



```
// Launch add() kernel on GPU
add<<<N/M,M>>>(d_a, d_b, d_c);
```

In Fifth.cu N+M-1 is passed because the reminder is not 0 and this way we can do the balance without errors. Also, N needs to be passed to control access out of bounds.

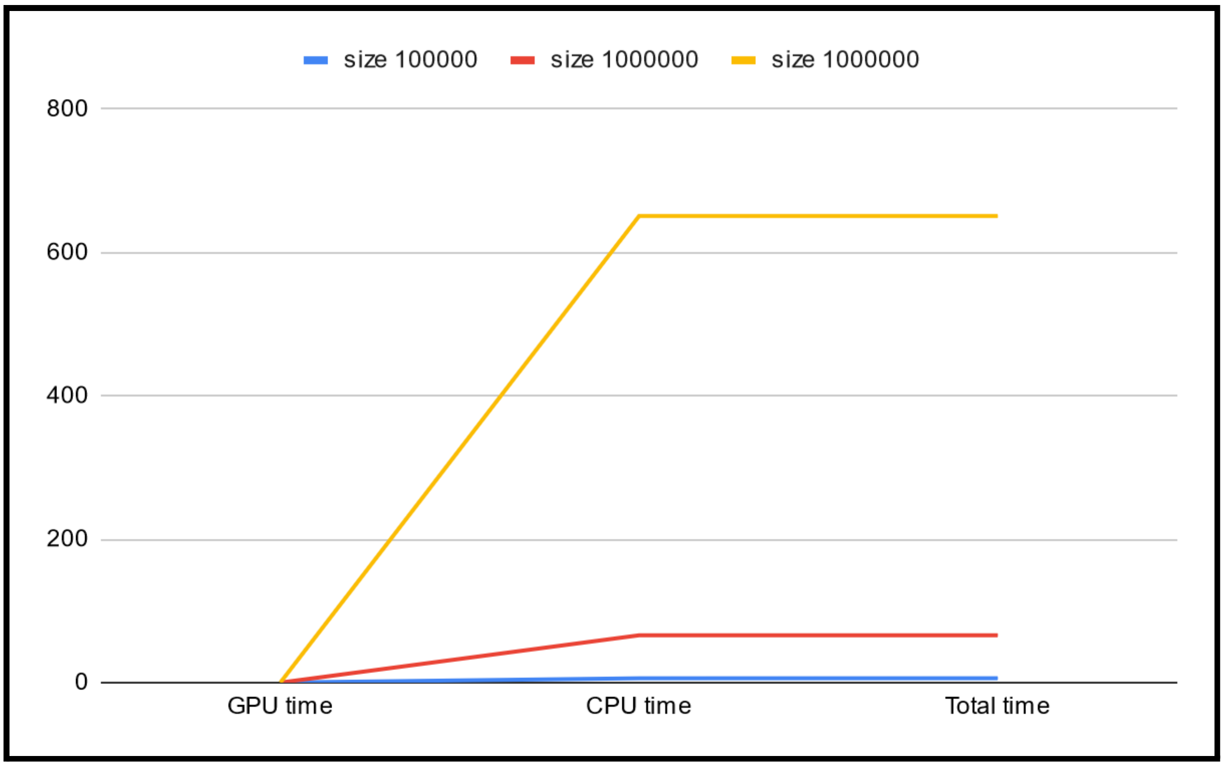


```
// Launch add() kernel on GPU
add<<<(N+M-1)/M,M>>>(d_a, d_b, d_c, N);
```

Of course, the first program of this two is going to be faster because there are less conditions to check but the latest is the best if the size of the vector and the number of threads per block where not defined and for exemple, entered by console because a not even division can occur.

5.*. Modify the last of the programs with the procedure of adding vectors on the CPU, add time measurement procedures and measure the time for the largest possible size of the vectors. Three times should be measured - the first as the CPU computing time, the second as the time of the GPU calculations without copying the data from memory, and the third as the duration of the entire operation on the graphics card. The results should be presented on a comparative graph for different sizes of vectors*

	size 100000	size 1000000	size 1000000
GPU time (ms)	0.0554	0.0019	0.0018
CPU time (ms)	6	66	651
Total time (ms)	6.0553	66.0019	651.0018



Source code modified can be found in this compressed file named exercise6.cu.