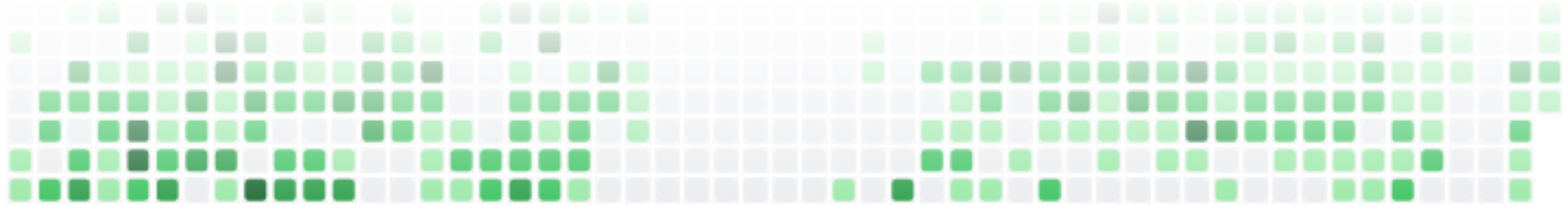
Introduction to Parallel Computing with CUDA

Professor: Alma Rodríguez Vázquez, Ph.D.
Student: Mariana Ávalos Arce

Universidad Panamericana, Faculty of Engineering
Fall 2021



Review: Pointers & Memory Allocation

1. Pointers

Let `int y;` then `&y` means the memory direction of y.

```
1 char c;
2 int* pc;
3 pc = &c; // error
```

```
1 char c;
2 char* pc;
3 pc = &c // no error
```

- **Declaration**

```
1 int x;
2 int* px; // declaration
```

- **Initialization**

Initialization of a pointer is when in its declaration, you also assign its value, which is a memory address.

```
1 int* px = &x;
```

- **Assignment**

After a pointer declaration, you can assign a memory address as the pointer value.

```
1 px = &x;
```

- **Indirection**

Indirection is accessing to the value of the variable the pointer is pointing at.

```
1 cout << *px;
```

Pointers To Pointers

```
1 float y = 2.5;
2 float *py = &y; // needs to be float because y is float //
     simple pointer
3 float **ppy = &py; // mem address of the pointer py, needs to
     be float because of y
4 float ***pppy = &ppy; // triple pointer, not so common
5 cout << ***pppy; // 2.5
```

Type of Pointers

- **void pointers**

Also called a generic pointer, they do not have a defined value of the variable they will point at, most of the times because the variable type they will point at is unknown. Void Pointers can point to whatever type of value.

```
char c;
int x;
void* pc;

pc = &c;
pc = &x; // all valid
```

- **null pointers**

A null pointer does not point to any direction, does not point to trash.

```
char c;
char* p = NULL;
```

- **constant pointers (int *const)**

```
int x = 4;
int* const px = &x; // pointer cannot change its memory
   address (but x can change), and must be initialized, else
   error
cout << *px; // 4 (print the value of x through a pointer)
*px = 8; // x or *px can change
cout << *px; // 8
// error px = &y;
```

- **pointers to constants (const int*)**

```
const int x_const = 12;
// error int* p3 = &x_const;
const int* p3 = &x_const; // no error, also this pointer can
   point to different const int variables // error if int * p3
    = &const_var
*p3 = 11; // error bc you're changing x_const or p3
const int y_const = 10;
p3 = &y_const; // p3 can change its target, but its variable
   type is const int
```

Pointers To Arrays

The name of an array is a pointer to its first element.

```
int arr[3] = {5,7,9};
int *p = arr; // same as p = &arr[0]
cout << *p; // 5
cout << p[0]; // 5
cout << *(p + 1); // 7, bc we are advancing one memory cell
cout << *(p++); // 7
```

Exercise 1

Code a function with no return value that receives three pointers to vectors (arrays) of integers.The function must add the vectors (v1[1] + v2[1]). For the implementation use:

- 2 arrays initialized with 5 elements each.
- 1 array to store the sum.
- 3 pointers that point to these vectors.

```
#include <iostream>
using namespace std;

void func(int* p1, int* p2, int* pr) {

    for (int i = 0; i < 5; i++) {
        pr[i] = p1[i] + p2[i];
    }

    return;
}

int main()
{
    int a1[5] = { 1,2,3,4,5 };
    int a2[5] = { 6,7,8,9,10 };
    int r[5] = { 0 };

    int* pr = r;
    int* p1 = a1;
    int* p2 = a2;

    func(p1, p2, pr);

    for (int i = 0; i < 5; i++) {
        cout << pr[i] << " "; // 7, 9, 11, 13, 15
    }

    for (int i = 0; i < 5; i++) {
        cout << r[i] << " "; // 7, 9, 11, 13, 15
    }
```

```
32 }
```

Pointers To Arrays of Pointers

It is posible to create arrays of pointers, and pointers that point to these arrays.

```
1 int v_int = 12;
2 int v_int2 = 3;
3
4 int *pt_array[3]; // array of pointers
5 int  **p_pt_array = pt_array; // pointer to array of pointers
6
7 pt_array[0] = &v_int;
8 pt_array[1] = &v_int2; // same as p_pt_array[1] = &v_int2
```

2. Dynamic Memory

The **new** operator in c++ is used to reserve memory in execution time, while the operator **delete** frees this memory.

- Syntaxis

```
1 void* new dataType;
2 void delete void* block;
3 void delete [] void* block;
```

- Examples

```
1 int* p;
2 p = new int;
3 *p = 10;
4 cout << *p; // 10
5 delete p;
```

```
1 int* p;
2 p = new int [10];
3 for (int i = 0; i < 10; i++){
4     p[i] = i;
5     cout << p[i] << "-";
6 }
7 delete [] p;
```

C's function `malloc()` is used to reserve memory in execution time, while the function `free()` frees this memory.

- Syntaxis

```
1 void* malloc(size_t size)
2 void free(void* block)
```

- `malloc()` returns a void* pointer and so we cast it as `(int *)`.

```
1 int *p;
2 p = (int*)malloc(sizeof(int)); // create a space in dynamic
     memory (exec mem) that is referenced by p pointer
3 *p = 45
4 cout<<*p;
5 free(p); // param is the name of the pointer that references
     to mem you want to free
```

```
1 int *p;
2 p = (int*)malloc(sizeof(int) * 10);
3 for(int i = 0; i < 10; i++){
4     p[i] = i;
5     cout << p[i] << "-";
6 }
7 free(p);
```

Exercise 2

Code a function that has no return value, and that it receives two pointers to integers. The function must swap the values of the parameters that it receives. These values must be from user input and stored in dynamic memory. For the implementation, use:

- Dynamic memory
- Pointers

```
1  #include <iostream>
2  using namespace std;
3
4  void func(int *p1, int *p2) {
5
6      int aux;
7
8      aux = *p1;
9      *p1 = *p2;
10     *p2 = aux;
11
12     return;
13 }
14
15 int main()
16 {
17     int* v1 = (int*)malloc(sizeof(int));
```

```
18      int* v2 = (int*)malloc(sizeof(int));
19
20      cin >> *v1;
21      cin >> *v2;
22
23      func(v1, v2);
24      cout << "v1: " << *v1 << "\nv2: " << *v2; // swap values
            of v1 and v2
25  }
```

Some Findings

```
1  bool* ptr;
2  *ptr = false;
3  if (ptr) {
4      // will always be true without *
5  }
```

```
1  bool* ptr = &true; // error
```

```
1  int a = 5;
2  int* b = &a;
3  int c = *b // c is a separate location with value 5
```

```
1  int arr[10];
2  int* p6 = &arr[6];
3  int* p0 = &arr[0];
4
5  cout << (int)p6 << " " << (int)p0; // 162328 162304 (24
       difference, 6 times 4(int))
6  cout <<"diff: " << p6 - p0; // 6 (pointers work on a space
       defined by the data type they point to)
```

```
1  int arr[10] = {3,6,9,12,15,18,21,24,27,30};
2  int* p0 = arr;
3
4  for (int i = 0; i < 10; i++) {
5      cout << (arr + i) << endl; // 10 addresses that differ by
           4 (int)
6      cout << *(arr + i) << endl; // all array nums
7      cout << p0 << " = " << *p0 << endl; // 10 addresses = each
           num
8      p0++;
9  }
```

```
1  char word[] = "hello!";
2  char* p = word;
3  char* p0 = &word[0];
```

```
char* p3 = &word[3];

cout << p << endl; // hello! // char pointers are especially treated as strings, thats why
cout << p0 << endl; // hello! // these prints dont show addresses
cout << p3 << endl; // lo!
```

```
// dynamic memory: when you know the size of memory only at run time, not at compile time
int size;
int* ptr;

cout << "Enter size: ";
cin >> size;

ptr = (int*)malloc(sizeof(int) * size);

for (int i = 0; i < size; i++) {
    cout << "Value: ";
    cin >> ptr[i];
}

for (int i = 0; i < size; i++) {
    cout << ptr[i] << " ";
}
```

```
int* ptr1;
int* ptr2;

ptr1 = (int*)malloc(10 * sizeof(int));
*ptr1 = { 0 }; // only sets ptr[0] = 0
for (int i = 0; i < 10; i++) {
    cout << ptr1[i] << " ";
}
// 0 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451 -842150451
for (int i = 0; i < 10; i++) {
    ptr1[i] = i * 10;
}
cout << *(ptr1 + 5) << endl; // 50
```

Passing in Funtions

There are 3 ways in which you can pass arguments (values) to functions:

- **Pass by Value**

This method copies the actual **value** of an argument into the parameter of the function,

where this parameter is only **local** to the function. Changes made to the parameter inside the function have no effect on the argument. By default, c++ uses this method of passing in functions. You create a new memory location with the value of the passed argument.

```
void passByValue(int a, int b){
    a++;
    b++;
}
int main(){
    int a = 2, b = 3;
    passByValue(a, b);
    cout << a << " " << b; // a = 2, b = 3
}
```

- **Pass by Reference**

This method copies the **reference** of an argument into the formal parameter. Inside the function, the reference is used to access the **actual argument** used in the call. Therefore, changes made to the parameter affect the passed argument. In this case, x is a new name given to a, and y is the new name for b. X is a new name for the same memory location named a.

```
void passByReference(int &x, int &y){
    x++;
    y++;
}
int main(){
    int a = 2, b = 3;
    passByReference(a, b);
    cout << a << " " << b; // a = 3, b = 4
}
```

- **Pass by Address**

Also named Pass by Pointer, this method copies the **address** of an argument into the formal parameter of a function. Inside the function, the **actual address** is used to access the actual argument used in the call. Therefore, changes made to the parameter affect the passed argument. To do this, argument pointers are passed to the function. You are passing the address of a to the inside of pointer x in the function argument.

```
void passByAddress(int* x, int* y){
    *x += 1;
    *y += 1;
}
int main(){
    int a = 2, b = 3;
    int* pa = &a, * pb = &b;
    passByAddress(&a, &b); // a = 3, b = 4
    passByAddress(pa, pb);
```

```
10      cout << a << " " << b; // a = 4, b = 5
11  }
```

Introduction: CUDA

CUDA is a hardware design and framework that allows the programming of the GPU nuclei and resource management.

CUDA Architecture: Compute Unified Device Architecture)

- Architecture introduced in 2006.
- Allows the scientific community the access to the GPU resources.
- Brings support for high level programming languages, such as C/C++.
- Unifies the use of independent processors, which come from the classic architecture.
- Uses an Heterogeneous Model: this is formed with two hardware elements, Device and Host.

The Hardware (GPC) design has the following:

- SM Unit (all green squares and blues, etc), also called MultiProcessors. Each multiProcessor has a max limit of blocks that can be processed in this SM. These have:
  - SP Unit (each green square), also called CUDA cores or nucleus. Search for this in CUDA website.
- A cluster (GPC) is the group of SM's or Streaming Multiprocessors.
- L2 Cache Shared Memory

Figure 1: Image

Heterogeneous Model

- **Host**: CPU. Less cores or nuclei.
- **Device**: GPU or Graphics Card.

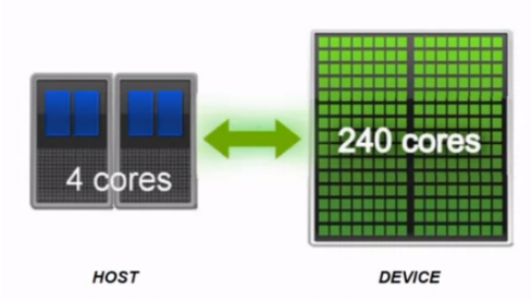


Figure 2: Image

Processing Stream

Starts with the Host (Sequential) and goes then to Device (Parallel) and then Host, etc...

Kernel, Threads, blocks and Grids

- **Kernel**: gives the instructions to all the cores or organizes the cores. The code snippet that you want to execute in parallel.
- **Blocks**: cores are organized or grouped in blocks. The yellow squares. A block groups threads.
- **Grid**: a group of one or more blocks (Green big square). Each GPU has only one Grid.
- **Warp**: a group of 32 threads, they're inside blocks as blocks have threads. A warp **is** physically executed in parallel.
- A single thread is executed in a single CUDA core. Commonly: Thread = CUDA core.

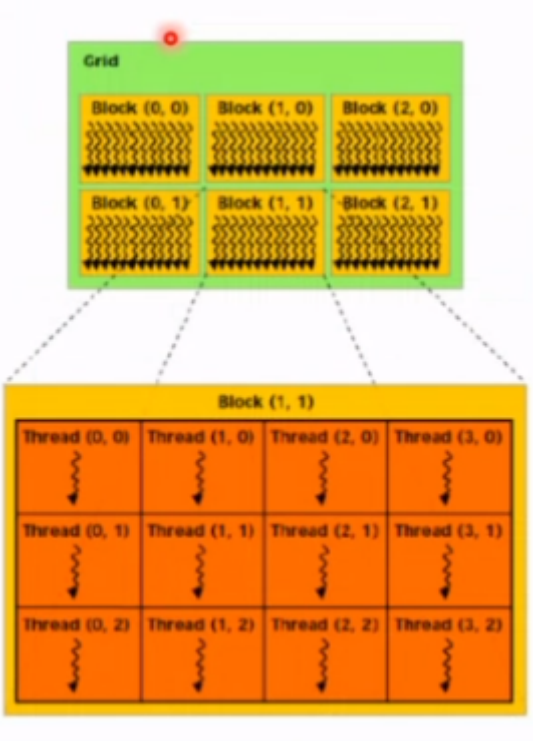


Figure 3: Image

Not every time everything runs in parallel, the first warp goes (32 threads per block) first, and then the next warp and so on. When a block is executed, not the whole block is executed, just its first 32 threads (warp), then other 32, etc.

- A GPU is a group of multiprocessors.
- A block has threads, but you can have different amounts of threads in many blocks, just careful not to exceed `threadsInBlock x Blocks` <= `maxThreadsPerMultiProcessor`.
- Grids and blocks are three dimensional.

Perspectives

- Hardware level: many cluster (multiprocessor)
- Software level: one 3D grid with blocks.

Lab 01

Print the main properties of your machine's GPU.

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>


int main()
{
    int device = 0; // to store the number of devices we have
    int* count = &device;
    cudaGetDeviceCount(count); // needs a pointer to store the
        result
    // a device is a gpu card
    printf("Device count: %d\n", device);

    cudaDeviceProp properties;
    cudaDeviceProp* pProperties = &properties;
    cudaGetDeviceProperties(pProperties, device - 1); //
       device int is an index, we have one so index is zero
    printf("Name: %s\n", properties.name); // name of the
       device
    printf("multiProcessorCount: %d\n", properties.
       multiProcessorCount);
    printf("maxBlocksPerMultiProcessor: %d\n", properties.
       maxBlocksPerMultiProcessor);
    // the sum of all the threads in each block
    printf("maxThreadsPerMultiProcessor: %d\n", properties.
       maxThreadsPerMultiProcessor);
    // max number of threads per block
    printf("maxThreadsPerBlock: %d\n", properties.
       maxThreadsPerBlock);

    // Grids dimensions
    printf("maxGridSize x axis: %d\n", properties.maxGridSize
       [0]); // max limit of blocks in x axis in the grid
    printf("maxGridSize y axis: %d\n", properties.maxGridSize
       [1]);
    printf("maxGridSize z axis: %d\n", properties.maxGridSize
       [2]);

    // Block dimensions (tweak but until the multip is <=
       1024)
    printf("maxThreadsDim x axis: %d\n", properties.
       maxThreadsDim[0]); // max limit of threads per
       dimension in block
    printf("maxThreadsDim y axis: %d\n", properties.
       maxThreadsDim[1]);
    printf("maxThreadsDim z axis: %d\n", properties.
       maxThreadsDim[2]);
```

```
35
36      return 0;
37  }
```

Output

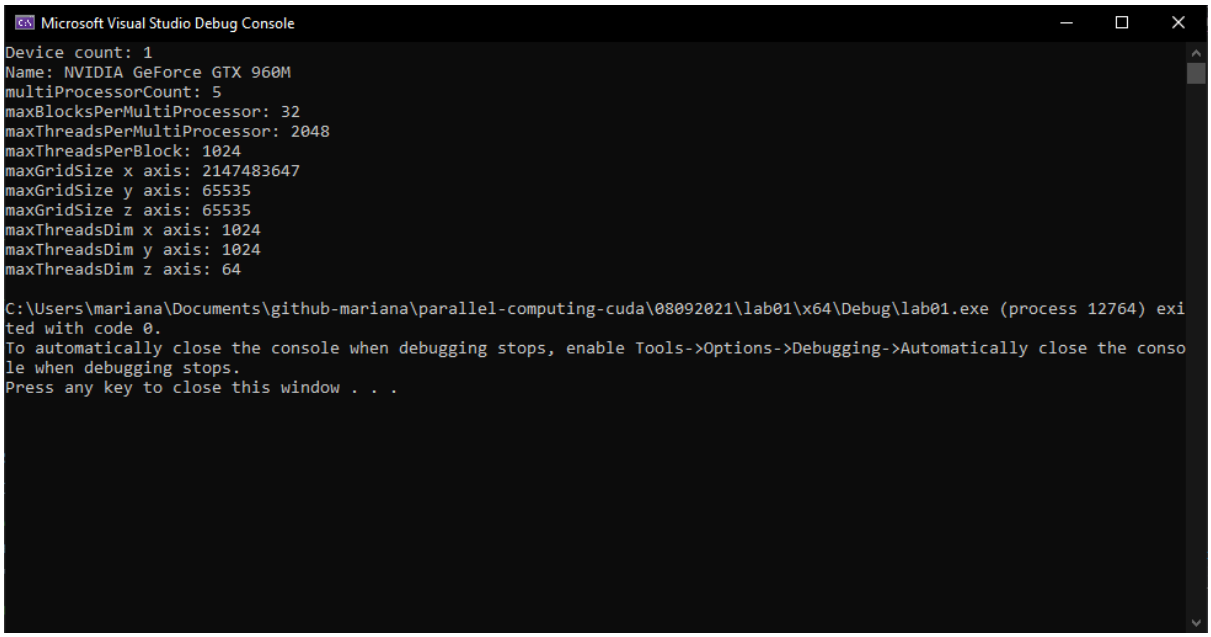


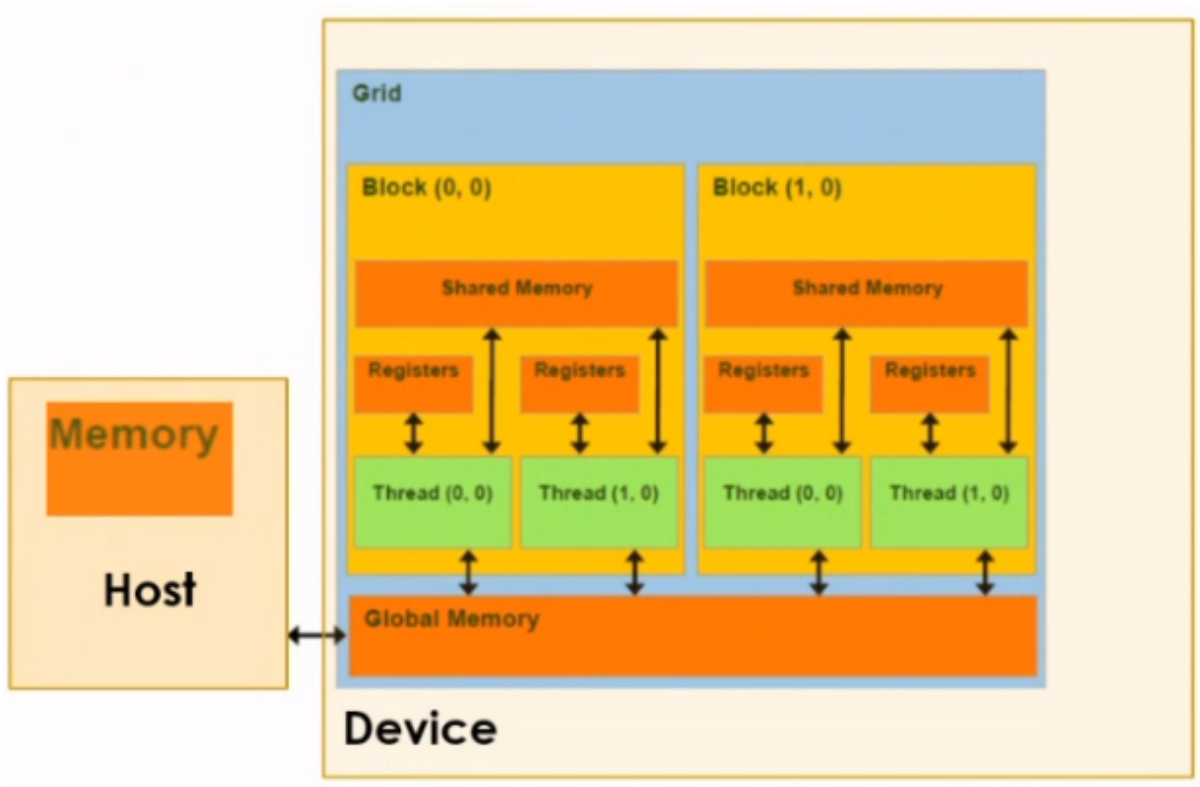
Figure 4: Image

Data Transfer and Memory Management

- A warp is the basic unit of grouping threads. We make this concept because a warp is physically executed.
- A block with 33 threads has 2 warps.
- A thread is executed in a CUDA core and a CUDA core is a processing nucleus.

A Host (CPU) and Device (GPU) have their own separate memory, CPU Memory and Global Memory respectively. CUDA creates functions to transfer info from one memory to another to connect these memories.

- **Global Memory** is a memory that is shared throughout all the blocks (and its threads) in a grid.
- **Shared Memory** is an independent memory exclusive to the threads in the block that has the said shared memory. It is a memory that each block has for its threads only, no other block has access to this.
- Each thread has its own memory, called **Register**, which is very quick access but limited in space.



Memory Management (Reserve)

The Host will reserve dynamic memory with `malloc(size)` and the Device will reserve memory in the Device's Global Memory using `cudaMalloc(void**, size)`.

Syntax & Example

- Host

```
void* malloc(size_t size);
void free(void* _Block);
```

```
float* host_mem;
host_mem = (float*)malloc(sizeof(float));
free(host_mem);
```

- Device

```
cudaMalloc(void** devPtr, size_t size);
cudaFree(void* devPtr);
```

```
float* dev_mem;
cudaMalloc((void**)&dev_mem, sizeof(float));
cudaFree(dev_mem);
```

Data Transfer

There are four types of data transfer, classified according to the direction of the transfer:

Type of Transfer	Direction Of Transfer	Description
cudaMemcpyHostToDevice	Host -> Device	Transfer data to be processed in parallel
cudaMemcpyDeviceToDevice	Device -> Device	Internal processing in the Device, without loops
cudaMemcpyDeviceToHost	Device -> Host	What was already processed in parallel, you return it to the Host to validate the info and see the results

Type of Transfer	Direction Of Transfer	Description
cudaMemcpyHostToHost	Host -> Host	Normal CPU processing, but without loops (fast copy)

Syntax & Example

```
cudaMemcpy(destination_mem, source_mem, size, typeOfTransfer);
```

```
float* host_mem;
host_mem = (float*)malloc(sizeof(float));
float* dev_mem;
cudaMalloc((void**)&dev_mem, sizeof(float));

cudaMemcpy(dev_mem, host_mem, sizeof(float),
   cudaMemcpyHostToDevice);
```

The Kernel

It is a method executed in the GPU as a mass execution. As seen before, CUDA architecure's Processing flow is done switching between CPU and GPU. The kernel is a function you execute on the Device.

Specifier (Identifier)	Called From	Executed In	Syntax	Description
__host__	Host	Host	__float__ float name()	CPU functions that are normally used in a program. You can skip the identifier, but it's good pratice.
__global__	Host	Device	__global__ void name()	This identifier is for the **kernel**. Executed in the GPU, in parallel.
__device__	Device	Device	__device__ float name()	This is executed in the device, but it's not exaclty a kernel (parallel), because it could or not be something in parallel.

- If there is no specifier before a function, it is simply taken as a normal function in CPU processing. In this way, `__host__` is just a simple CPU function as well.
- `__device__` functions are defined throughout your code and then a kernel function calls it. Could be or not a parallel process function, it is just a method the kernel might need to be done.
- A **kernel** return value type is always void. If you want to return something, you do it by a reference pass using the kernel parameters.

- The specifier `__global__` creates a kernel.
- To call a kernel is to execute code in the GPU.
- **Synchronization**: when you want to wait for an action in parallel in order to begin another sequentially or viceversa.

The Kernel Syntax

The kernel call is done in the CPU, the execution is in GPU.

```
__global__ void myKernel(arg_1, arg_2, ..., arg_n) {
    // code to be executed in the GPU
}

// from CPU you call the kernel
myKernel<<<blocks,threads>>>(arg_1, arg_2, ..., arg_n);
```

Launching a Kernel

Basic Sequence:

1. Reserve memory in the host: this memory will be used for storing the info that will be processed in parallel.
2. Initialize the data from the host: this reserved memory is initialized with the said info to process in the GPU.
3. Data transfer from host to device: all this memory is in the host, and in order to process it in the device, we need to copy it to the device.
4. Launch the kernel specifying the number of blocks and threads: give the order of the execution of the function in the GPU, taking the transfered data to be processed.
5. Result data transfer from device to host: from the device memory, we copy the info back to the host so that it can be analized.
6. Free memory (device and host) from the host.

Exercise 1

- A function that adds two integer numbers in the host (a simple add function).
- A function that adds two integer numbers in the device through the launch of a kernel.

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__host__ int addCPU(int* num1, int* num2) {
    return(*num1 + *num2);
}

// kernel: __global__
__global__ void addGPU(int* num1, int* num2, int* res)
{
    *res = *num1 + *num2;
}

int main()
{
    // reserve mem in host
    int* host_num1 = (int*)malloc(sizeof(int)); // could be a simple integer and then you pass as param the &variable
    int* host_num2 = (int*)malloc(sizeof(int));
    int* host_resCPU = (int*)malloc(sizeof(int));
    int* host_resGPU = (int*)malloc(sizeof(int));

    // reserve mem in dev
    int* dev_num1, * dev_num2, * dev_res;
    cudaMalloc((void**)&dev_num1, sizeof(int)); // &3 error // &intvar no error but you need pointers with malloc in cuda
    cudaMalloc((void**)&dev_num2, sizeof(int));
    cudaMalloc((void**)&dev_res, sizeof(int)); // this pointer points to an address in the device

    // init data
    *host_num1 = 2;
    *host_num2 = 3;
    *host_resCPU = 0;
    *host_resGPU = 0;

    // data transfer
    cudaMemcpy(dev_num1, host_num1, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_num2, host_num2, sizeof(int), cudaMemcpyHostToDevice);

    // CPU call to CPU func
    *host_resCPU = addCPU(host_num1, host_num2);
    printf("CPU result \n");
    printf("%d + %d = %d \n", *host_num1, *host_num2, *
```

```
        host_resCPU);
45
46      // CPU call to GPU func
47      addGPU <<< 1, 1 >>> (dev_num1, dev_num2, dev_res);
48      // dev_res is a pointer made with cudaMalloc (Global
        Memory)
49      cudaMemcpy(host_resGPU, dev_res, sizeof(int),
        cudaMemcpyDeviceToHost);
50      printf("GPU result \n");
51      // dev_num1 is án address in GPU, you cannot access it
        from CPU
52      printf("%d + %d = %d \n", *host_num1, *host_num2, *
        host_resGPU);
53
54      // free memory
55      free(host_num1);
56      free(host_num2);
57      free(host_resCPU);
58      free(host_resGPU);
59
60      cudaFree(dev_num1);
61      cudaFree(dev_num2);
62      cudaFree(dev_res);
63
64      return 0;
65  }
```

At the line `int* host_num1 = (int*)malloc(sizeof(int));`, Visual Studio makes a suggestion, which is the same I put in the comment on that line.

```
int main()
{
    int* host_num1 = (int*)malloc(sizeof(int)); // co
    int* host_num2 = (int*)malloc(sizeof(int));
    int* host_resCPU = (int*)malloc(sizeof(int));
    int* host_resGPU = (int*)malloc(sizeof(int));

    int* dev_num1, * dev_num2, * dev_res;
    cudaMalloc((void**)&dev_num1, sizeof(int));
    cudaMalloc((void**)&dev_num2, sizeof(int));
    cudaMalloc((void**)&dev_res, sizeof(int));

    *host_num1 = 2;
    *host_n
    *host_r
    *host_r

    return 0;
}
```

(local variable) int *host_num1
could be a simple integer and then you pass as param the &variable
Search Online
C6011: Dereferencing NULL pointer 'host_num1'.

Output Example 01

My machine produces the following output.

```
Microsoft Visual Studio Debug Console
CPU result
2 + 3 = 5
GPU result
2 + 3 = 5

C:\Users\mariana\Documents\github-mariana\parallel-computing-cuda\08162021\ex01\x64\Debug\ex01.exe (process 7512) exited
 with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Code Alternatives

```
__global__ void addGPU(int num1, int num2, int* res)
{
    *res = num1 + num2;
}
int main(){
    addGPU <<< 1, 1 >>> (2, 3, dev_res);
}
```

Practice

Exercise 1

- Function that solve a system of linear equations in the host.
- Function that solves a system of linear equations in the device through the launch of a kernel (1 block and 1 thread).

The kernel must receive all coefficients as a vector (of size 6).

A linear system with the form:

> ax + by = c dx + ey = f,

Can be solved by the formulas:

> x = (ce - bf) / (ae -bd) y = (af - cd) / (ae - bd)

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__host__ void linearSolveCPU(float* n, float* x, float* y) {
    *x = (n[2] * n[4] - n[1] * n[5]) / (n[0] * n[4] - n[1] * n
        [3]);
    *y = (n[0] * n[5] - n[2] * n[3]) / (n[0] * n[4] - n[1] * n
        [3]);
}

__global__ void linearSolveGPU(float* n, float* x, float* y)
{
    *x = (n[2] * n[4] - n[1] * n[5]) / (n[0] * n[4] - n[1] * n
        [3]);
    *y = (n[0] * n[5] - n[2] * n[3]) / (n[0] * n[4] - n[1] * n
        [3]);
}

int main()
{
    float* n_host = (float*)malloc(sizeof(float) * 6); // if
        malloc, you need to initialize all spaces one by one
    float* x_host = (float*)malloc(sizeof(float));
    float* y_host = (float*)malloc(sizeof(float));

    float* x_gpu = (float*)malloc(sizeof(float));
    float* y_gpu = (float*)malloc(sizeof(float));

```

```
27      float* n_device;
28      float* x_device;
29      float* y_device;
30
31      cudaMalloc((void**)&n_device, sizeof(float) * 6);
32      cudaMalloc((void**)&x_device, sizeof(float));
33      cudaMalloc((void**)&y_device, sizeof(float));
34
35      n_host[0] = 5;
36      n_host[1] = 1;
37      n_host[2] = 4;
38      n_host[3] = 2;
39      n_host[4] = -3;
40      n_host[5] = 5;
41
42      *x_host = 0;
43      *y_host = 0;
44      *x_gpu = 0;
45      *y_gpu = 0;
46
47      cudaMemcpy(n_device, n_host, sizeof(float) * 6,
           cudaMemcpyHostToDevice);
48      cudaMemcpy(x_device, x_host, sizeof(float),
           cudaMemcpyHostToDevice);
49      cudaMemcpy(y_device, y_host, sizeof(float),
           cudaMemcpyHostToDevice);
50
51      linearSolveCPU(n_host, x_host, y_host);
52      printf("CPU result \n");
53      printf("x = %f y = %f \n", *x_host, *y_host);
54
55      linearSolveGPU <<< 1, 1 >>> (n_device, x_device, y_device)
           ;
56      cudaMemcpy(x_gpu, x_device, sizeof(float),
           cudaMemcpyDeviceToHost);
57      cudaMemcpy(y_gpu, y_device, sizeof(float),
           cudaMemcpyDeviceToHost);
58      printf("GPU result \n");
59      printf("x = %f y = %f \n", *x_gpu, *y_gpu);
60
61      free(n_host);
62      free(x_host);
63      free(y_host);
64      free(x_gpu);
65      free(y_gpu);
66
67      cudaFree(n_device);
68      cudaFree(x_device);
69      cudaFree(y_device);
70
71      return 0;
```

```
72 }
```

Lab 03

Make a program in c/c++ in which you launch a kernel with one block and one thread. The kernel must solve a quadratic equation in the form:

> ax^2 + bx + c = 0,

where its solutions are given by:

> x1 = (-b + sqrt(b^2 - 4ac)) / 2a x2 = (-b - sqrt(b^2 - 4ac)) / 2a

For the implementation, you must consider:

1. Ask the user for coefficients a, b and c.
2. The program must show the solutions for the equation or a message stating that the solution does NOT exist if the result is an imaginary number.

Tests

- a = 1, b = -5, c = 6 -> x1 = 2, x2 = 3
- a = 1, b = 1, c = 1 -> The solution does not exist

Solution

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7
8  __global__ void solveGPU(double* dev_abc, double* dev_x1x2,
       bool* dev_error)
9  {
10     double root = (dev_abc[1] * dev_abc[1]) - (4 * dev_abc[0]
           * dev_abc[2]);
11     // printf("root: %lf\n", root);
12     if (root < 0) {
13         *dev_error = true;
14     }
15     else {
16         *dev_error = false;
```

```
17          dev_x1x2[0] = ((-1 * dev_abc[1] - sqrt(root)) / (2 *
               dev_abc[0]));
18          dev_x1x2[1] = ((-1 * dev_abc[1] + sqrt(root)) / (2 *
               dev_abc[0]));
19      }
20
21  }
22
23  int main() {
24      double* n_host = (double*)malloc(sizeof(double) * 3); //
           not cast, error
25      double* x1x2_host = (double*)malloc(sizeof(double) * 2);
26      bool* error_host = (bool*)malloc(sizeof(bool));
27
28      double* n_dev;
29      double* x1x2_dev;
30      bool* error_dev;
31      cudaMalloc((void**)&n_dev, sizeof(double) * 3);
32      cudaMalloc((void**)&x1x2_dev, sizeof(double) * 2);
33      cudaMalloc((void**)&error_dev, sizeof(bool)); // &bool
           error
34
35      for (int i = 0; i < 3; i++) {
36          printf("%c: ", char(i + 97)); //printf("%s", (i + 65))
               ; exception
37          scanf("%lf", &n_host[i]); // "A:%lf" not error, but
               input incomplete // \n weird results
38      }
39
40      x1x2_host[0] = 0;
41      x1x2_host[1] = 0;
42      *error_host = false;
43
44      cudaMemcpy(n_dev, n_host, sizeof(double) * 3,
           cudaMemcpyHostToDevice);
45      cudaMemcpy(x1x2_dev, x1x2_host, sizeof(double) * 2,
           cudaMemcpyHostToDevice); // not necessary
46      cudaMemcpy(error_dev, error_host, sizeof(bool),
           cudaMemcpyHostToDevice); // not necessary
47
48      solveGPU << < 1, 1 >> > (n_dev, x1x2_dev, error_dev);
49
50      // cout << "cuda ptr " << *error_dev << endl; // no error,
           but execption at runtime
51      cudaMemcpy(error_host, error_dev, sizeof(bool),
           cudaMemcpyDeviceToHost);
52      cudaMemcpy(x1x2_host, x1x2_dev, sizeof(double) * 2,
           cudaMemcpyDeviceToHost);
53      if (*error_host) {
54          printf("GPU Result:\n");
55          printf("The solution does not exist\n");
```

```
56      }
57      else {
58          printf("GPU Result:\n");
59          printf("x1 = %lf x2 = %lf\n", x1x2_host[0], x1x2_host
                [1]);
60      }
61
62  }
```

Other Findings

```
1   #include "cuda_runtime.h"
2   #include "device_launch_parameters.h"
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <math.h>
7   #include<iostream>
8
9   using namespace std;
10
11  __global__ void solveGPU(double* dev_abc, double* dev_x1x2,
        int* dev_error)
12  {
13      double root = (dev_abc[1] * dev_abc[1]) - (4 * dev_abc[0]
            * dev_abc[2]);
14      if (root < 0) {
15          *dev_error = true;
16      }
17      else {
18          *dev_error = false;
19          dev_x1x2[0] = ((-1 * dev_abc[1] - sqrt(root)) / (2 *
                dev_abc[0]));
20          dev_x1x2[1] = ((-1 * dev_abc[1] + sqrt(root)) / (2 *
                dev_abc[0]));
21      }
22
23  }
24
25  int main() {
26      double n_host[3] = { 0 };
27      double x1x2_host[2] = { 0 };
28      bool error_host = false;
29
30      double* n_dev;
31      double* x1x2_dev;
32      int* error_dev; // gives no error
33      cudaMalloc((void**)&n_dev, sizeof(double) * 3);
34      cudaMalloc((void**)&x1x2_dev, sizeof(double) * 2);
```

```
35      cudaMalloc((void**)&error_dev, sizeof(bool));
36
37      for (int i = 0; i < 3; i++) {
38          printf("%c: ", char(i + 97));
39          scanf("%lf", &n_host[i]);
40      }
41
42      cudaMemcpy(n_dev, n_host, sizeof(double) * 3,
           cudaMemcpyHostToDevice);
43      cudaMemcpy(x1x2_dev, x1x2_host, sizeof(double) * 2,
           cudaMemcpyHostToDevice); // not necessary
44      cudaMemcpy(error_dev, &error_host, sizeof(bool),
           cudaMemcpyHostToDevice); // not necessary
45
46      solveGPU << < 1, 1 >> > (n_dev, x1x2_dev, error_dev);
47
48
49      cudaMemcpy(&error_host, error_dev, sizeof(bool),
           cudaMemcpyDeviceToHost);
50      cudaMemcpy(x1x2_host, x1x2_dev, sizeof(double) * 2,
           cudaMemcpyDeviceToHost);
51      if (error_host) {
52          printf("GPU Result:\n");
53          printf("The solution does not exist\n");
54      }
55      else {
56          printf("GPU Result:\n");
57          printf("x1 = %lf x2 = %lf\n", x1x2_host[0], x1x2_host
               [1]);
58      }
59
60      //free(n_host); // exc
61      //free(x1x2_host); // exc
62      //free(&error_host); // exc
63
64      cudaFree(n_dev);
65      cudaFree(x1x2_dev);
66      cudaFree(error_dev);
67  }
```

```
1  int* test;
2  cudaMalloc((void**)&test, sizeof(bool)); // no error
```

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7  #include<iostream>
8
```

```
9  using namespace std;
10
11 __global__ void solveGPU(double* dev_abc, double* dev_x1x2,
       int* dev_error)
12 {
13     double root = (dev_abc[1] * dev_abc[1]) - (4 * dev_abc[0]
           * dev_abc[2]);
14     if (root < 0) {
15         *dev_error = true;
16     }
17     else {
18         *dev_error = false;
19         dev_x1x2[0] = ((-1 * dev_abc[1] - sqrt(root)) / (2 *
               dev_abc[0]));
20         dev_x1x2[1] = ((-1 * dev_abc[1] + sqrt(root)) / (2 *
               dev_abc[0]));
21     }
22
23 }
24
25 int main() {
26     double n_host[3] = { 0 };
27     double x1x2_host[2] = { 0 };
28     bool error_host = false;
29
30     double* n_dev;
31     double* x1x2_dev;
32     int* error_dev; // gives no error
33     cudaMalloc((void**)&n_dev, sizeof(double) * 3);
34     cudaMalloc((void**)&x1x2_dev, sizeof(double) * 2);
35     cudaMalloc((void**)&error_dev, sizeof(bool));
36
37     for (int i = 0; i < 3; i++) {
38         printf("%c: ", char(i + 97));
39         scanf("%lf", &n_host[i]);
40     }
41
42     cudaMemcpy(n_dev, n_host, sizeof(double) * 3,
           cudaMemcpyHostToDevice);
43     cudaMemcpy(x1x2_dev, x1x2_host, sizeof(double) * 2,
           cudaMemcpyHostToDevice); // not necessary
44     cudaMemcpy(error_dev, &error_host, sizeof(bool),
           cudaMemcpyHostToDevice); // not necessary
45
46     solveGPU << < 1, 1 >> > (n_dev, x1x2_dev, error_dev);
47
48
49     cudaMemcpy(&error_host, error_dev, sizeof(bool),
           cudaMemcpyDeviceToHost);
50     cudaMemcpy(x1x2_host, x1x2_dev, sizeof(double) * 2,
           cudaMemcpyDeviceToHost);
```

```
51      if (error_host) {
52          printf("GPU Result:\n");
53          printf("The solution does not exist\n");
54      }
55      else {
56          printf("GPU Result:\n");
57          printf("x1 = %lf x2 = %lf\n", x1x2_host[0], x1x2_host
                [1]);
58      }
59  }
```

- `cudaMalloc(void** devPtr, size_t size)`: Allocates `size` bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. Memory not cleared.
- `cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)`: Copies `count` bytes from the memory area pointed to by src to the memory area pointed to by dst. Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Thread Identification

Many Threads, One Block

- Threads (orange cubes) are contained or grouped in blocks (yellow container). In the image, there are six threads, one block. In this way, The kernel (function) will be executed by each thread at the same time (in parallel). This means that if we were to launch a kernel with this config (six threads one block), all we coded inside the kernel would be executed six times, one by each thread.

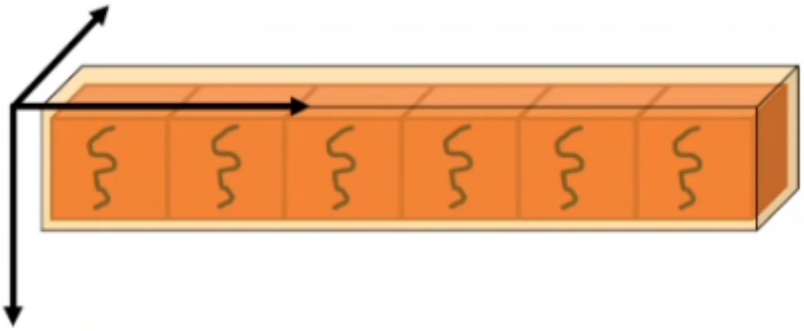


Figure 1: Image

Blocks are tridimensional, as well as grids. The above block is one dimensional.

- To identify threads:
  - `threadIdx.x`: x axis index number of the thread. If you have only one block with six threads along one axis (one dimensional block), this property will be the full id. Indexes start in zero. It's y and z components are zero in one dimensional blocks, like the following image.

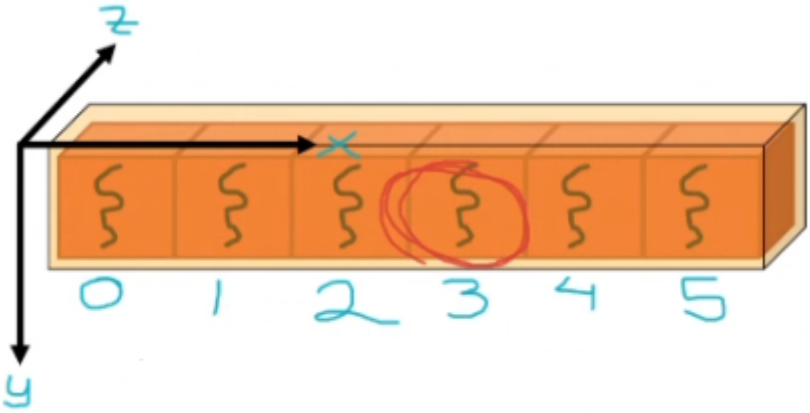


Figure 2: Image

In the case of **a one dimensional block**, a thread would be identified with only one property: `threadIdx.x`. Properties `threadIdx.y` and `threadIdx.z` are zero. For example, (3,0,0).

- `threadIdx.y`: you will also need y component if your block is bidimensional, like the below image. Z component is zero in that case as well.

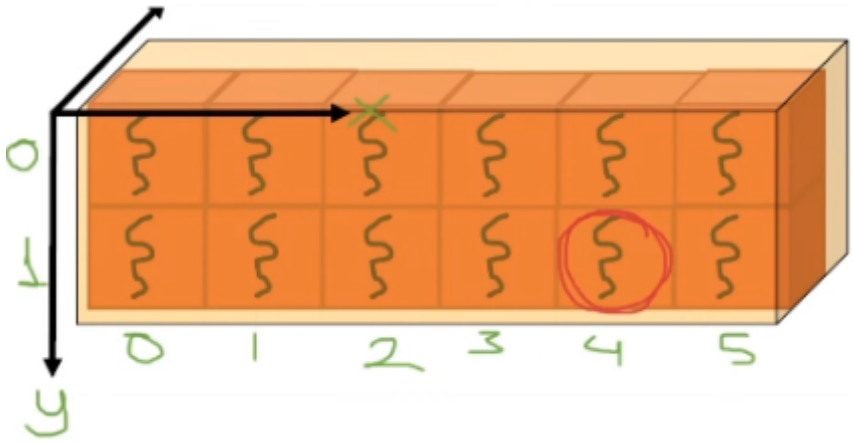


Figure 3: Image

In the case of **a two dimensional block**, to identify a thread in this single-block config, you would need two components: `threadIdx.x` and `threadIdx.y`, while `threadIdx.z` is zero. For example (4,1,0).

- `threadIdx.z`: for the case of **a three dimensional block**, you will need a third component to identify threads in it, called `threadIdx.z`, which indicates the position of the thread in the z axis. For example, (4,0,1).

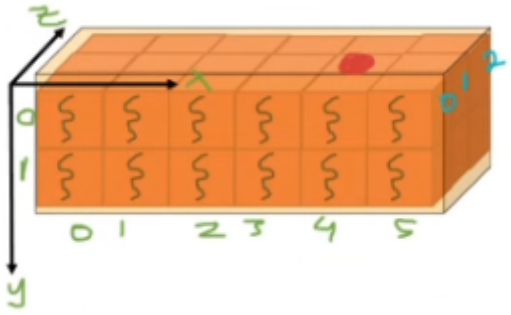


Figure 4: Image

We need to identify threads in order to give instructions to particular threads inside the kernel. Then, **these indexes** will allow us to identify a thread inside **its block**.

Most applications use **one dimensional blocks**. We said that we can identify a thread in a 1d block with its `threadIdx.x`, but this only works to identify threads if we have only 1 block.

- `globalId`: is the global id of a thread, which allows to identify a thread from **all others**. In the case of a one dimensional block, `globalID = threadIdx.x`. If the block and grid config is different, `globalId` is calculated differently.
- `dim3` objects have the properties x, y and z. We can determine the configuration of the grid (number of blocks per axis) using its default constructor `dim3 grid(3,1,1)`, for example. You can create a dim3 object to configure the dimensions of the blocks as well (threads quantity in each axis or direction of a block). Therefore,
  - `dim3 grid`: how are blocks organized in the grid, or *how many blocks will we launch with a kernel and how are these organized in the axes*. For example, `dim3 grid(3,1,1)` in the below image. (For one block, we would have `dim3 grid(1,1,1)`)

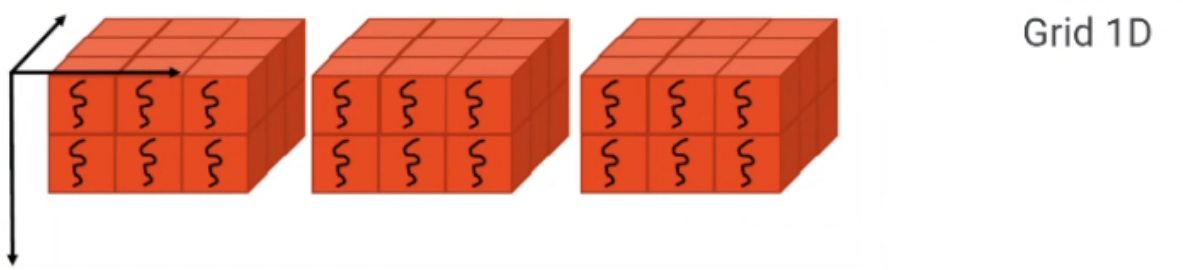


Figure 5: Image

Or bidimensional grids as with `dim3 grid(3,2,1)`:

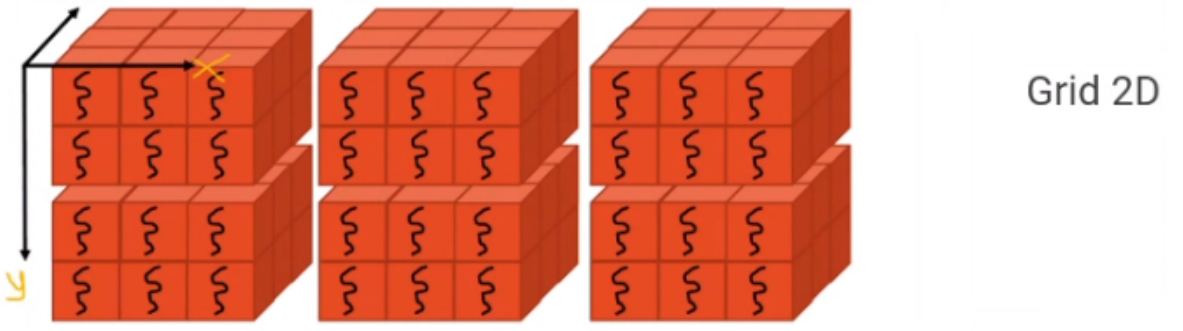


Figure 6: Image

As well as three dimensional grids with the example of `dim3 grid(3,2,2)`:

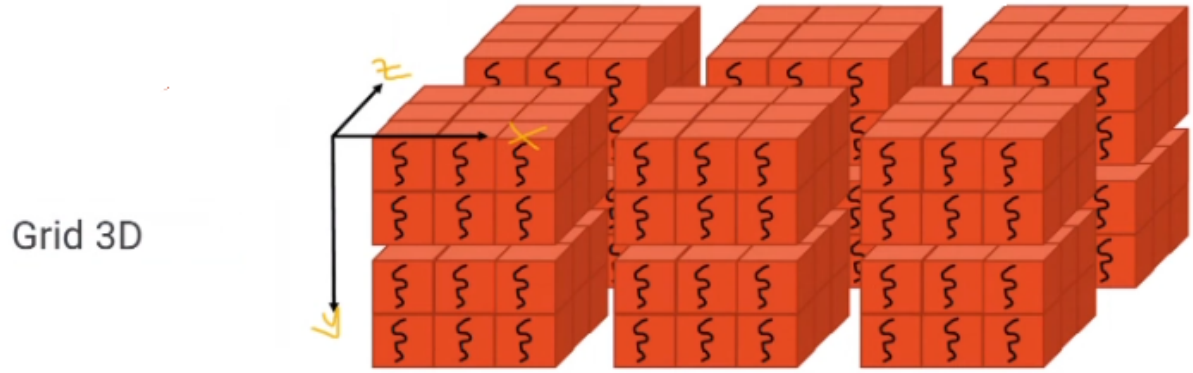


Figure 7: img

- `dim3 block`: how are threads organized in the blocks, *how a block is configured: how many threads and in which axes.*

Example 1

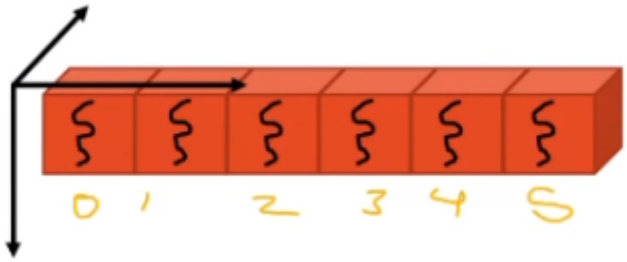


Figure 8: Image

For this case of one grid with a 1d block, the code would be:

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  __global__ void kernel()
8  {
9      printf("threadIdx.x: %d, threadIdx.y: %d, threadIdx.z: %d
           \n", threadIdx.x, threadIdx.y, threadIdx.z);
10 }
11
12 int main() {
13     dim3 grid(1, 1, 1); // dim3 grid(2, 1, 1); x =
           012345012345 y = 0 in all 12, z = 0 in all 12
```

```
14      dim3 block(6, 1, 1);
15      kernel <<< grid, block >>> ();
16
17      return 0;
18  }
```

Which would output:

```
1  threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0
2  threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0
3  threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 0
4  threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 0
5  threadIdx.x: 4, threadIdx.y: 0, threadIdx.z: 0
6  threadIdx.x: 5, threadIdx.y: 0, threadIdx.z: 0
```

Example 2

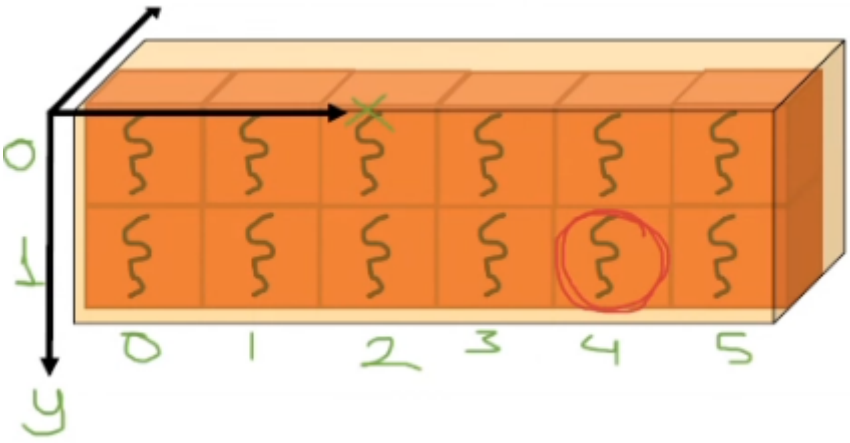


Figure 9: Image

For this case of 1 block with 6 threads in x and 2 in y axis:

```
1  __global__ void kernel()
2  {
3      printf("threadIdx.x: %d, threadIdx.y: %d, threadIdx.z: %d
           \n", threadIdx.x, threadIdx.y, threadIdx.z);
4  }
5
6  int main() {
7      dim3 grid(1, 1, 1);
8      dim3 block(6, 2, 1);
9      kernel <<< grid, block >>> ();
10
11     return 0;
12 }
```

Where its output is:

```
threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 4, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 5, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 2, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 3, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 4, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 5, threadIdx.y: 1, threadIdx.z: 0
```

Example 3

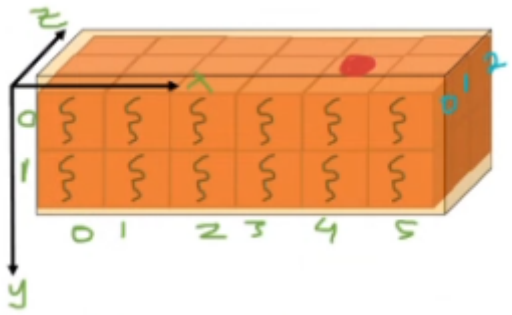


Figure 10: Image

For this case of 1 block with 6 threads in x, 2 in y and 3 in z axis:

```
__global__ void kernel()
{
    printf("threadIdx.x: %d, threadIdx.y: %d, threadIdx.z: %d
       \n", threadIdx.x, threadIdx.y, threadIdx.z);
}

int main() {
    dim3 grid(1, 1, 1);
    dim3 block(6, 2, 3);
    kernel <<< grid, block >>> ();

    return 0;
}
```

Where its output is:

```
threadIdx.x: 2, threadIdx.y: 1, threadIdx.z: 2
threadIdx.x: 3, threadIdx.y: 1, threadIdx.z: 2
```

```
threadIdx.x: 4, threadIdx.y: 1, threadIdx.z: 2
threadIdx.x: 5, threadIdx.y: 1, threadIdx.z: 2
threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 4, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 5, threadIdx.y: 0, threadIdx.z: 0
threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 2, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 3, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 4, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 5, threadIdx.y: 1, threadIdx.z: 0
threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1
threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1
threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 1
threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 1
threadIdx.x: 4, threadIdx.y: 0, threadIdx.z: 1
threadIdx.x: 5, threadIdx.y: 0, threadIdx.z: 1
threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1
threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1
threadIdx.x: 2, threadIdx.y: 1, threadIdx.z: 1
threadIdx.x: 3, threadIdx.y: 1, threadIdx.z: 1
threadIdx.x: 4, threadIdx.y: 1, threadIdx.z: 1
threadIdx.x: 5, threadIdx.y: 1, threadIdx.z: 1
threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 2
threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 2
threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 2
threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 2
threadIdx.x: 4, threadIdx.y: 0, threadIdx.z: 2
threadIdx.x: 5, threadIdx.y: 0, threadIdx.z: 2
threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 2
threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 2
```

They are not in order, because nothing guarantees that the threads will be in order. As soon as each thread finished printing, it appears on the screen.

Because of the repeated Ids, we have a *unique* Id for each thread in a block, called **globalId**.

One Block & One Dimension

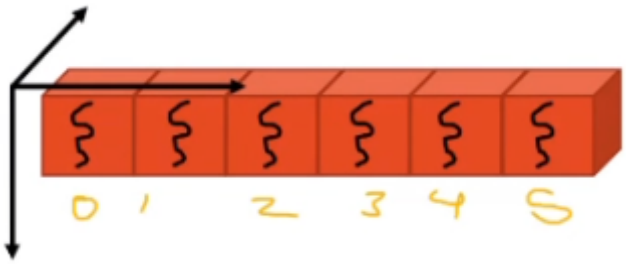


Figure 11: Image

```
__global__ void printGlobalId_oneBlockOneDim()
{
    printf("GlobalId: %d\n", threadIdx.x);
}

int main() {
    dim3 grid(1, 1, 1);
    dim3 block(6, 1, 1);
    printGlobalId_oneBlockOneDim <<< grid, block >>> ();

    return 0;
}
```

Which outputs:

```
threadIdx.x: 0
threadIdx.x: 1
threadIdx.x: 2
threadIdx.x: 3
threadIdx.x: 4
threadIdx.x: 5
```

N Blocks, 1 Axis (One Dimension)

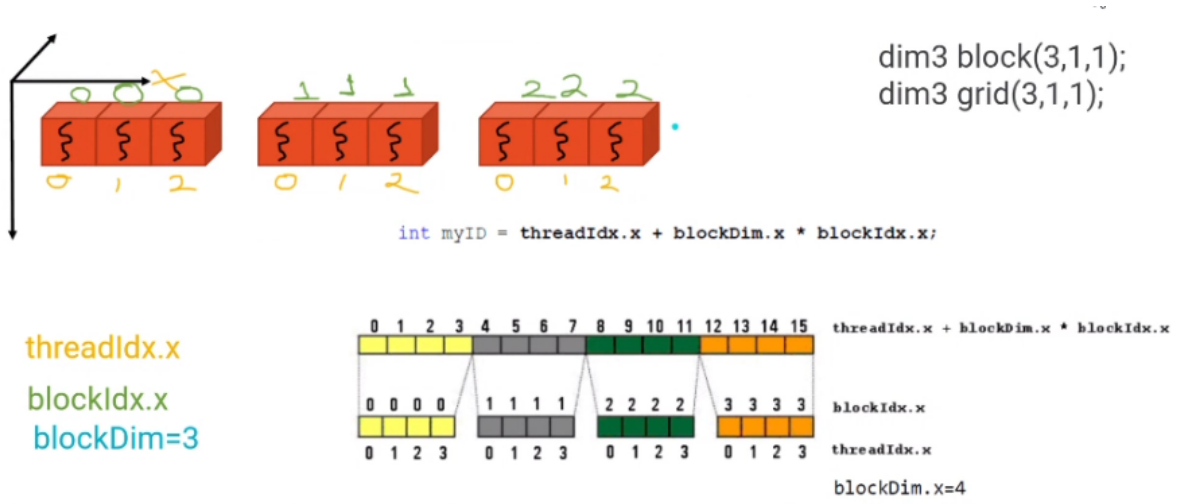


Figure 12: Image

Then, because their threadIdx.x would be 012 012 012, we need:

> int globalId = threadIdx.x + blockDim.x * blockIdx.x;

```
1  __global__ void printGlobalId_NBlocksOneDim()
2  {
3      int globalId = threadIdx.x + blockDim.x * blockIdx.x;
4      printf("GlobalId: %d\n", globalId);
5  }
6
7  int main() {
8      dim3 grid(3, 1, 1);
9      dim3 block(3, 1, 1);
10     printGlobalId_NBlocksOneDim<<< grid, block >>> ();
11
12     return 0;
13 }
```

Which outputs unique ids:

```
1  GlobalId: 6
2  GlobalId: 7
3  GlobalId: 8
4  GlobalId: 3
5  GlobalId: 4
6  GlobalId: 5
7  GlobalId: 0
8  GlobalId: 1
9  GlobalId: 2
```

Note: other config that is not N blocks in X axis, we need another formula.

To Finish

Then, completing the first function prints:

```
__global__ void kernel()
{
    int globalId = threadIdx.x + blockDim.x * blockIdx.x;
    printf("globalId: %d, threadIdx.x: %d, threadIdx.y: %d,
       threadIdx.z: %d, blockDim.x: %d, blockIdx.x %d\n",
       globalId, threadIdx.x, threadIdx.y, threadIdx.z,
       blockDim.x, blockIdx.x);
}
int main() {
    dim3 grid(3, 1, 1);
    dim3 block(4, 1, 1);
    kernel<<< grid, block >>> ();

    return 0;
}
```

Which outputs (threads (threadIdx.x) **per block** is ordered):

```
globalId: 8, threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 2
globalId: 9, threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 2
globalId: 10, threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 2
globalId: 11, threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 2
globalId: 4, threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 1
globalId: 5, threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 1
globalId: 6, threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 1
globalId: 7, threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 1
globalId: 0, threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 0
globalId: 1, threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 0
globalId: 2, threadIdx.x: 2, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 0
globalId: 3, threadIdx.x: 3, threadIdx.y: 0, threadIdx.z: 0,
   blockDim.x: 4, blockIdx.x 0
```

Other Configs (3D Grid with 3D Blocks)

- Properties `threadIdx.x`, `threadIdx.y` and `threadIdx.z` are used to identify a thread **inside its block**.
- When we have a much more complicated grid, we need additional indexes apart from these in order to calculate **gloablId** values, which will take on account other blocks in the config.
  - `blockIdx`: determines the index of a block inside a grid. Indexes start in zero. It has x,y,z values.
  - `blockDim`: how many **threads** per dimension are in a block. These are **quantity values**, not indexes, and therefore are constant for all threads. It has x,y,z values.
  - `gridDim`: how many **blocks** per dimension are in the **grid**. These are **quantity values**, not indexes, and therefore are constant for all threads. It has x,y,z values.

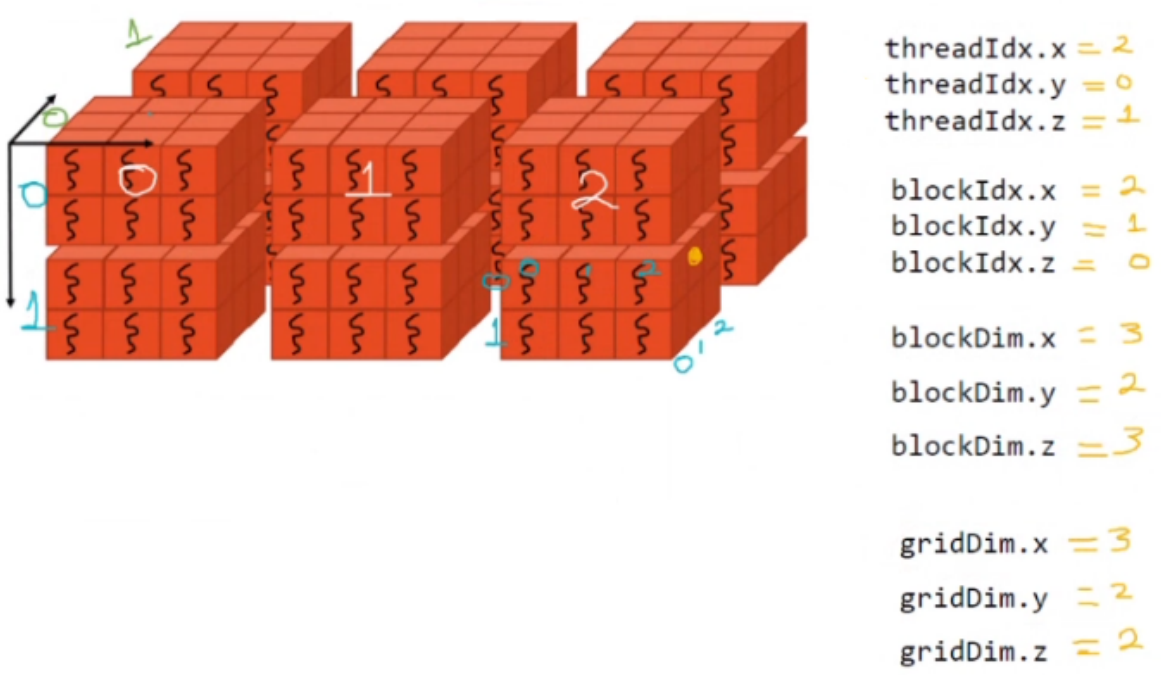


Figure 1: Image

Lab 04

Make a program in c/c++ using CUDA in which you implement a grid in 3D that has 4 threads in each dimension and 2 threads per block dimension. The program must print from kernel the values of the following indexes, considering their three dimensions:

- threadIdx
- blockIdx
- blockDim
- gridDim

Like the image below:

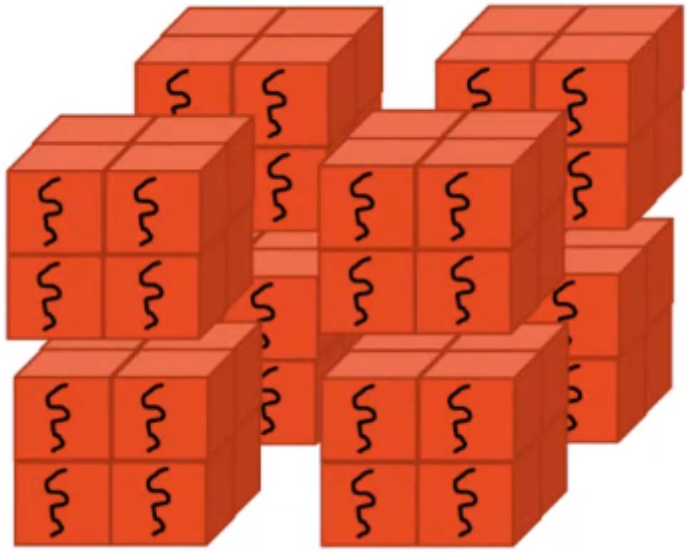


Figure 2: Image

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__global__ void kernel()
{
    printf("threadIdx.x: %d, threadIdx.y: %d, threadIdx.z: %d
       | blockIdx.x: %d, blockIdx.y: %d, blockIdx.z: %d |
       blockDim.x: %d, blockDim.y: %d, blockDim.z: %d |
       gridDim.x: %d, gridDim.y: %d, gridDim.z: %d\n",
        threadIdx.x, threadIdx.y, threadIdx.z,
        blockIdx.x, blockIdx.y, blockIdx.z,
        blockDim.x, blockDim.y, blockDim.z,
        gridDim.x, gridDim.y, gridDim.z);
}

int main() {
```

```
17      dim3 grid(2, 2, 2);
18      dim3 block(2, 2, 2);
19      kernel <<< grid, block >>> ();
20
21      return 0;
22  }
```

Which outputs 64 lines:

```
1  threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
2  threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
3  threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
4  threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
5  threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
6  threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
7  threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
8  threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
       1, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
9  threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
       0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
10 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
       0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
       : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
        2
11 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
       0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
```

```
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
12 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
13 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
14 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
15 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
16 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
17 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
18 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
19 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
20 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
21 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
22 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
23 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
24 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
```

```
      1, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
25 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
26 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
27 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
28 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
29 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
30 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
31 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
32 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
33 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
34 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
35 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
36 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
```

```
37 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
38 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
39 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
40 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 0, blockIdx.z: 0 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
41 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
42 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
43 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
44 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
45 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
46 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
47 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
48 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      0, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
49 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
```

```
        2
50 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
51 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
52 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
53 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
54 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
55 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
56 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 1, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
57 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
58 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
59 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
60 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 0 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
61 threadIdx.x: 0, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
62 threadIdx.x: 1, threadIdx.y: 0, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
```

```
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
63 threadIdx.x: 0, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
64 threadIdx.x: 1, threadIdx.y: 1, threadIdx.z: 1 | blockIdx.x:
      1, blockIdx.y: 0, blockIdx.z: 1 | blockDim.x: 2, blockDim.y
      : 2, blockDim.z: 2 | gridDim.x: 2, gridDim.y: 2, gridDim.z:
       2
```

Practice

- If you launch a kernel with 1 block and this block has only 1 dimension: `globalId = threadIdx.x`. A globalId is important because it allows us to identify a thread from another with a **unique** value.
- If we change the configuration and we launch a kernel with N one-dimensional blocks along 1 grid axis, the `globalId = threadIdx.x + blockDim.x * blockIdx.x`. The block ids are the same for threads inside its block.
- The kernel is in charge of massive processing. The kernel function will be executed in parallel N times, through the N threads: each thread will work with different data but applying the same operations to this data, which are written as the kernel code. We need to identify the thread (globalId) in order to give the thread different data to perform its kernel code.

Exercise 1

The idea is to have two vectors (arrays) of size 12 each, and sum its elements to store them in a third array, of size 12 as well: first element of A + first element of B, stored on the first element of C and so on.

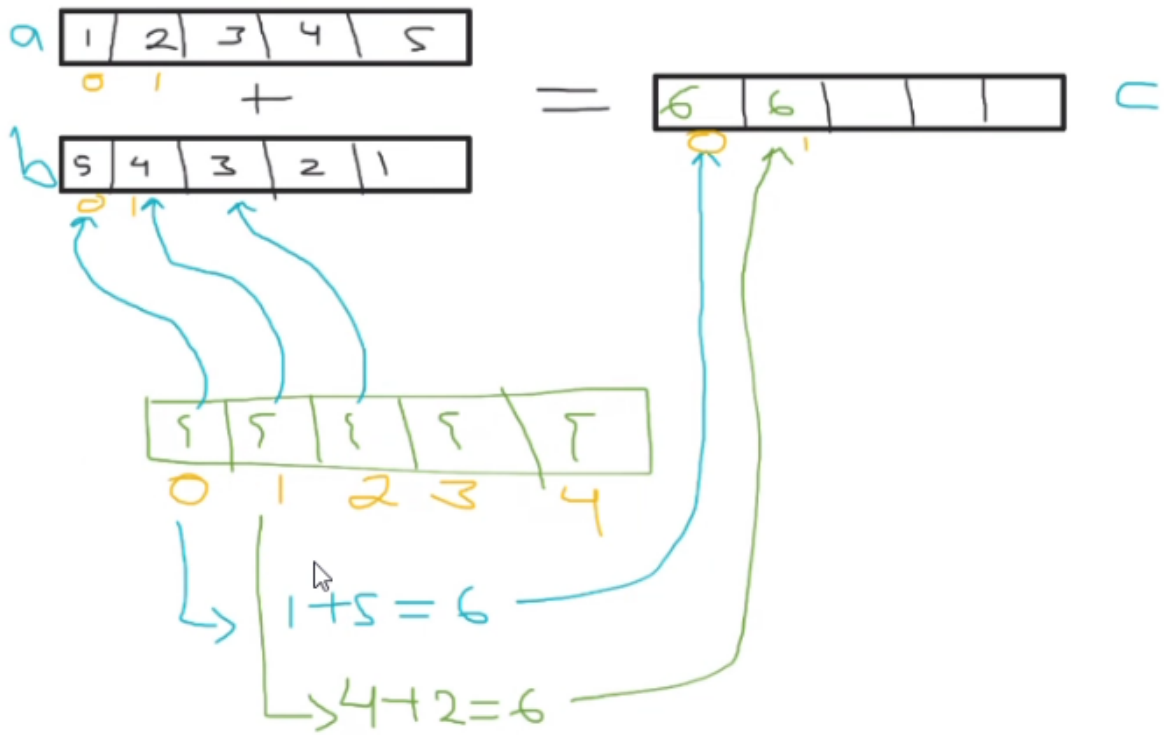


Figure 1: Image

- Each thread will be in charge of doing the sum of two elements. We need 12 sums, and therefore we will use 12 threads. We will use 1 one-dimensional block with 12 threads, in order to make it easier to direct a thread to its assigned sum of values and its assigned storage cell, using its `threadIdx.x`.
- If we were to do this with a for loop, we would be doing **sequential programming**, because first we sum the first elements of the arrays, and then move to the second elements, and so on **sequentially**: this makes it possible to do one sum at a time. With CUDA, we would reduce this time, because we would do all 12 sums at the same time.
- The instruction is the same for all threads, the only thing that changes is the data with which this operation/instruction will be done by each thread. In this case, the globalId will be useful to locate the data to process in arrays a and b, but also to assign the thread that will operate this data.
- The kernel will be executed N times, one by each of the N threads. 12 threads will execute the two lines of code inside the kernel: there will be 12 gId vars (one for each thread), this `gId` variable can be considered as a **different** variable for each thread.

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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


__global__ void arraySum(int* dev_a, int* dev_b, int* dev_c) {
    // 1block and this with one dimension
    int gId = threadIdx.x; // there will be 12 gId variables
       when all threads are executing the kernel
    // 12 vars, one for each thread
    dev_c[gId] = dev_a[gId] + dev_b[gId];
}

int main()
{
    const int vectorSize = 12;
    int host_a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
    int host_b[] = { 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    int host_c[vectorSize] = { 0 };

    int* dev_a, * dev_b, * dev_c;
    cudaMalloc((void**)&dev_a, sizeof(int) * vectorSize);
```

```
24      cudaMalloc((void**)&dev_b, sizeof(int) * vectorSize);
25      cudaMalloc((void**)&dev_c, sizeof(int) * vectorSize);
26
27      cudaMemcpy(dev_a, host_a, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
28      cudaMemcpy(dev_b, host_b, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
29
30      dim3 grid(1, 1, 1); // or dim3 grid(1);
31      dim3 block(vectorSize, 1, 1); // or dim3 block(vectorSize)
           ;
32
33      arraySum << < grid, block >> > (dev_a, dev_b, dev_c);
34
35      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
36
37      for (int i = 0; i < vectorSize; i++) {
38          printf("%d ", host_c[i]);
39      }
40
41      cudaFree(dev_a);
42      cudaFree(dev_b);
43      cudaFree(dev_c);
44
45      return 0;
46  }
```

Output

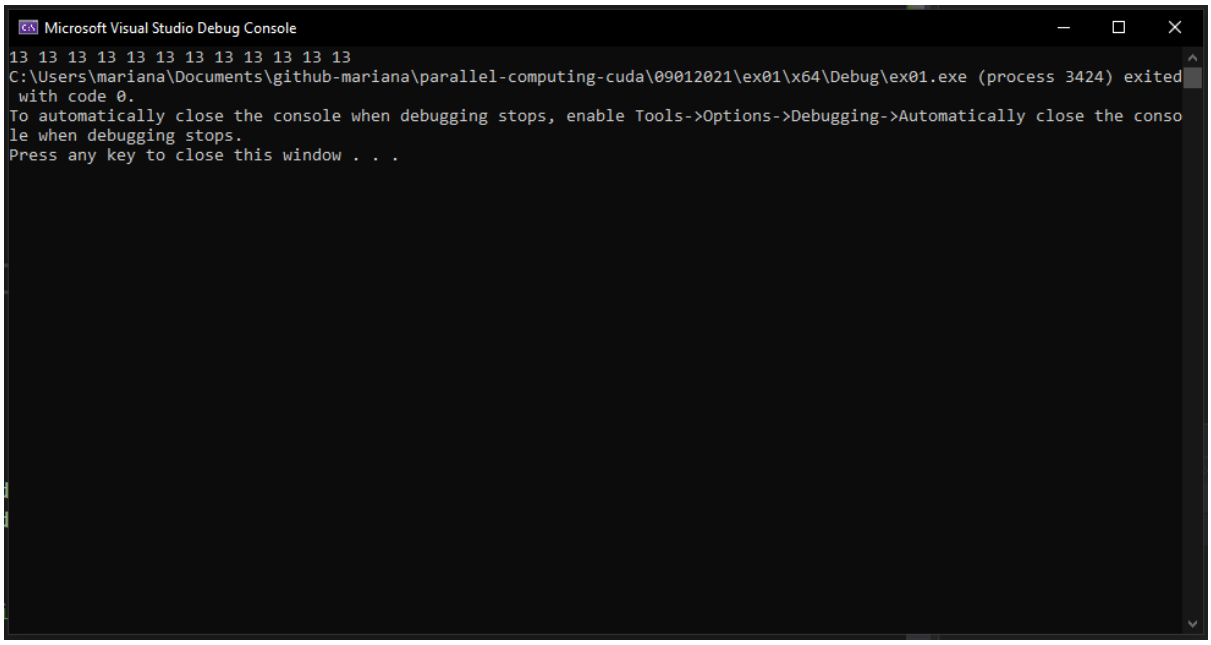


Figure 2: Image

- For example, when working with images, as they are matrices, the best thing to do is to configure the blocks to be of dimensions similar to those image matrices: a bidimensional block to use the threadIdx x and y components.

Practice

Lab 05

Make a program in c/c++ using CUDA in which you implement a kernel that adds two arrays (A and B) of integers generated randomly, considering the requirements:

- 32 threads
- A 1D grid with 4 blocks
- Use 1D blocks with 8 threads in each block
- Array size must be of 32

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h> /* srand, rand */
#include <time.h> /* time */
__global__ void sumaVectores(int* vecA, int* vecB, int* vecRes) {
    int gId = threadIdx.x + blockDim.x * blockIdx.x;
    vecRes[gId] = vecA[gId] + vecB[gId];
}
int main()
{
    const int vectorSize = 32;
    int* host_a = (int*)malloc(sizeof(int) * vectorSize);
    int* host_b = (int*)malloc(sizeof(int) * vectorSize);
    int* host_c = (int*)malloc(sizeof(int) * vectorSize);

    int* dev_a, * dev_b, * dev_c;

    cudaMalloc((void**)&dev_a, sizeof(int) * vectorSize);
    cudaMalloc((void**)&dev_b, sizeof(int) * vectorSize);
    cudaMalloc((void**)&dev_c, sizeof(int) * vectorSize);

    srand(time(NULL));

    for (int i = 0; i < vectorSize; i++) {
        int num = rand() % vectorSize + 1;
        host_a[i] = num;
        num = rand() % vectorSize + 1;
        host_b[i] = num;
    }

```

```
32      cudaMemcpy(dev_a, host_a, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
33      cudaMemcpy(dev_b, host_b, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
34      dim3 grid(4, 1, 1);
35      dim3 block(8, 1, 1);
36
37      sumaVectores << < grid, block >> > (dev_a, dev_b, dev_c);
38
39      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
40
41      printf("Vector A: \n");
42      for (int i = 0; i < vectorSize; i++) {
43          printf("%d ", host_a[i]);
44      }
45      printf("\nVector B: \n");
46      for (int i = 0; i < vectorSize; i++) {
47          printf("%d ", host_b[i]);
48      }
49      printf("\nVector C: \n");
50      for (int i = 0; i < vectorSize; i++) {
51          printf("%d ", host_c[i]);
52      }
53
54      free(host_a);
55      free(host_b);
56      free(host_c);
57      cudaFree(dev_a);
58      cudaFree(dev_b);
59      cudaFree(dev_c);
60      return 0;
61  }
```

Lab 06

Make a program in c/c++ using CUDA in which you implement a kernel that assigns the different identification indexes for the threads into three arrays (A, B and C) considering the requirements:

- Use 3 arrays of integers with 64 elements each
- The kernel must manage 64 threads
- Each thread must write in an array a thread index:
  - The thread id (threadIdx.x) in array A
  - The block index (blockIdx.x) in array B

  - The global index (globalId = thread.x + blockDim.x * blockIdx.x) in the C array

- The kernel must execute 3 times, each with the structure:
  - 1 block of 64 threads
  - 64 blocks of 1 thread
  - 4 blocks of 16 threads
- Print the results from main function, with a text indicating the config of each execution.

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h> /* srand, rand */
#include <time.h> /* time */

#include<iostream>
using namespace std;
__global__ void idKernel(int* vecA, int* vecB, int* vecC) {
    int gId = threadIdx.x + blockDim.x * blockIdx.x;

    vecA[gId] = threadIdx.x;
    vecB[gId] = blockIdx.x;
    vecC[gId] = gId;
}

void printArray(int* arr, int size, char * msg) {
    cout << msg << ": ";
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main()
{
    const int vectorSize = 64;
    int* host_a = (int*)malloc(sizeof(int) * vectorSize);
    int* host_b = (int*)malloc(sizeof(int) * vectorSize);
    int* host_c = (int*)malloc(sizeof(int) * vectorSize);

    int* dev_a, * dev_b, * dev_c;

    cudaMalloc((void**)&dev_a, sizeof(int) * vectorSize);
```

```
35      cudaMalloc((void**)&dev_b, sizeof(int) * vectorSize);
36      cudaMalloc((void**)&dev_c, sizeof(int) * vectorSize);
37
38      srand(time(NULL));
39
40      for (int i = 0; i < vectorSize; i++) {
41          host_a[i] = 0;
42          host_b[i] = 0;
43          host_c[i] = 0;
44      }
45
46      cudaMemcpy(dev_a, host_a, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
47      cudaMemcpy(dev_b, host_b, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
48      cudaMemcpy(dev_c, host_c, sizeof(int) * vectorSize,
           cudaMemcpyHostToDevice);
49
50      dim3 grid(1, 1, 1);
51      dim3 block(64, 1, 1);
52      idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
53      cudaDeviceSynchronize(); // wait until kernel finishes and
            then come back to following code
54      cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
55      cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
56      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
57
58      printf("Execution 1: 1 block 64 threads \n");
59      printArray(host_a, vectorSize, "threadIdx.x");
60      printArray(host_b, vectorSize, "blockIdx.x");
61      printArray(host_c, vectorSize, "globalId");
62
63      grid.x = 64; // (64, 1, 1)
64      block.x = 1; // (1, 1, 1)
65      idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
66      cudaDeviceSynchronize();
67      cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
68      cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
69      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
70
71      printf("\nExecution 2: 64 blocks 1 thread \n");
72      printArray(host_a, vectorSize, "threadIdx.x");
73      printArray(host_b, vectorSize, "blockIdx.x");
74      printArray(host_c, vectorSize, "globalId");
75
```

```
76      grid.x = 4;
77      block.x = 16;
78      idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
79      cudaDeviceSynchronize();
80      cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
81      cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
82      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
83
84      printf("\nExecution 3: 4 block 16 threads \n");
85      printArray(host_a, vectorSize, "threadIdx.x");
86      printArray(host_b, vectorSize, "blockIdx.x");
87      printArray(host_c, vectorSize, "globalId");
88
89      free(host_a);
90      free(host_b);
91      free(host_c);
92      cudaFree(dev_a);
93      cudaFree(dev_b);
94      cudaFree(dev_c);
95      return 0;
96  }
```

Output

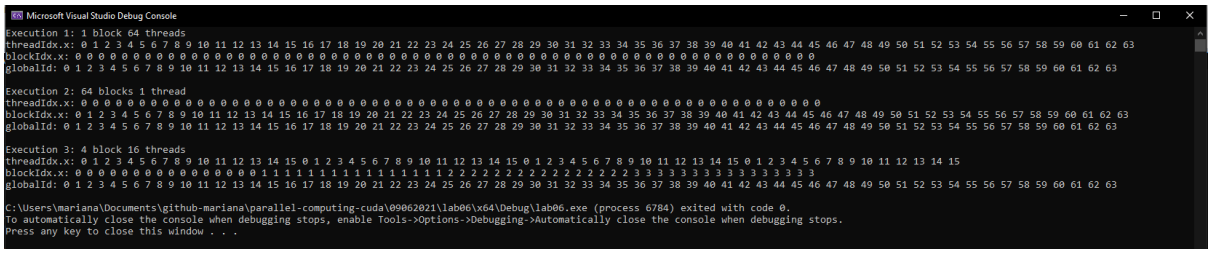


Figure 1: Image

Observations

- To use one kernel, you need to calculate the globalId as `threadIdx.x + blockDim.x * blockIdx.x` for **all** configurations, because this small formula works for a config of a 1D block along 1 dimension (axis), or more 1D blocks along that same dimension (axis).
- Whenever we launch a kernel, the instruction of the kernel start to execute in the device, but the program execution stream in main continues with the next lines, so the launch of following kernels can get mixed up, especially in printings. In order to

tell the program to **wait until a kernel execution finishes** in order to continue with the next host lines, you use the function `cudaDeviceSynchronize()` right after the launch of each kernel. With this, we are telling the host to make a pause in that line until the kernel finishes so that it can continue with the following lines. This pause means to **synchronize the executions**. Thus, this function is used whenever we want a program to wait for a kernel to finish before it continues its stream.

Error Management in CUDA

CUDA provides a way to handle errors that involve **exceeded GPU capacities** or **GPU malfunctioning**: the hardest errors to find out. These are not logic nor syntax errors.

- `cudaError_t` is a CUDA type given to handle errors, that is really an integer, and this number gives us a hint about the possible error. Every CUDA function returns an error that can be stored in this type. The only CUDA function that doesnt return a `cudaError_t` variable is a kernel itself: it must return void.

```
cudaError_t error;
error = cudaMalloc((void**)&ptr, size);
```

Types of Errors (Most Common)

- `cudaSuccess = 0`: The API call returned with no errors. In query calls, this also means the query is complete. Successful execution.
- `cudaErrorInvalidValue = 1`: This indicates that one or more parameters passed to the API function call is not within an acceptable range of values. An enum param in a CUDA function that you didnt match, or a different data type sent.
- `cudaErrorMemoryAllocation = 2`: the API call failed because it was unable to allocate enough memory to perform the requested operation. When you do not have/allocate enough space in kernel memory for a requested instruction: you would normally write on memory outside of your array, but if there is no more mem left, this happens.
- `cudaErrorInvalidConfiguration = 9`: This indicates that a kernel launch is requesting resources that can never be satisfied with the current device. Requesting too many shared memory per block than supported, as well as requesting too many threads or blocks. This happens when you have an invalid kernel config (grid/blocks): when you exceed the max num of blocks per grid or threads of the GPU card.
- `cudaErrorInvalidMemcpyDirection = 21`: the direction of the memcpy passed to API call is not one of the types specified by cudaMemcpyKind. You put another word, basically.

Process

```
cudaError_t error;
error = cudaMalloc((void**)&ptr, size);
cudaGetErrorString(error);

// after a kernel launch
```

```
error = cudaGetLastError();
cudaGetErrorString(error);
```

- `error` would be an intger, but in order to avoid checking in the docs, CUDA provides the function `cudaGetErrorString(error)` that, given an integer, it returns a string of the error details/explanation.
- To get a kernel error (otherwise a kernel is just void return), we can catch the last integer of error with `cudaGetLastError();`.

Open any project an type this funtion that will be used after any CUDA function call:

```
// host because every function call must be from host
__host__ void checkCUDAError(const char* msg){
    cudaError_t error;
    cudaDeviceSynchronize(); // avoid catching another error
       that is not the next we want
    error = cudaGetLastError(); // status of the last CUDA API
        call (maybe 0 or success, not error)
    if (error != cudaSuccess){
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}
```

Because the host/device execution is asynchronous (both at the same time), we need to take care of the sequence and sometimes you need to **pause** and wait for the kernel to finish in order to come back to the host: we need to synchronize.

Example

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h> /* srand, rand */
#include <time.h> /* time */

#include<iostream>
using namespace std;

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}
```

```
18
19 __global__ void idKernel(int* vecA, int* vecB, int* vecC) {
20     int gId = threadIdx.x + blockDim.x * blockIdx.x;
21
22     vecA[gId] = threadIdx.x;
23     vecB[gId] = blockIdx.x;
24     vecC[gId] = gId;
25 }
26
27 void printArray(int* arr, int size, char* msg) {
28     cout << msg << ": ";
29     for (int i = 0; i < size; i++) {
30         printf("%d ", arr[i]);
31     }
32     printf("\n");
33 }
34
35 int main()
36 {
37     const int vectorSize = 64;
38     int* host_a = (int*)malloc(sizeof(int) * vectorSize);
39     int* host_b = (int*)malloc(sizeof(int) * vectorSize);
40     int* host_c = (int*)malloc(sizeof(int) * vectorSize);
41
42     int* dev_a, * dev_b, * dev_c;
43
44     cudaMalloc((void**)&dev_a, sizeof(int) * vectorSize);
45     checkCUDAError("Error at cudaMalloc for dev_a");
46     cudaMalloc((void**)&dev_b, sizeof(int) * vectorSize);
47     checkCUDAError("Error at cudaMalloc for dev_b");
48     cudaMalloc((void**)&dev_c, sizeof(int) * vectorSize);
49     checkCUDAError("Error at cudaMalloc for dev_c");
50
51     srand(time(NULL));
52
53     for (int i = 0; i < vectorSize; i++) {
54         host_a[i] = 0;
55         host_b[i] = 0;
56         host_c[i] = 0;
57     }
58
59     cudaMemcpy(dev_a, host_a, sizeof(int) * vectorSize,
          cudaMemcpyHostToDevice);
60     checkCUDAError("Error at cudaMemcpy for host_a to dev_a");
61     cudaMemcpy(dev_b, host_b, sizeof(int) * vectorSize,
          cudaMemcpyHostToDevice);
62     checkCUDAError("Error at cudaMemcpy for host_b to dev_b");
63     cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
          cudaMemcpyHostToDevice); // error 1
64     checkCUDAError("Error at cudaMemcpy for host_c to dev_c");
65
```

```
66      dim3 grid(1, 1, 1);
67      dim3 block(2000, 1, 1); // max num is 1024, so here we
           will force an error
68      idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
69      checkCUDAError("Error at idKernel execution no. 1");
70      cudaDeviceSynchronize(); // wait until kernel finishes and
            then come back to following code // not needed to
           check
71      cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
72      //check also here
73      cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
74      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
75
76      printf("Execution 1: 1 block 64 threads \n");
77      printArray(host_a, vectorSize, "threadIdx.x");
78      printArray(host_b, vectorSize, "blockIdx.x");
79      printArray(host_c, vectorSize, "globalId");
80
81      grid.x = 64; // (64, 1, 1)
82      block.x = 1; // (1, 1, 1)
83      idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
84      cudaDeviceSynchronize();
85      cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
86      cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
87      cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
88
89      printf("\nExecution 2: 64 blocks 1 thread \n");
90      printArray(host_a, vectorSize, "threadIdx.x");
91      printArray(host_b, vectorSize, "blockIdx.x");
92      printArray(host_c, vectorSize, "globalId");
93
94      grid.x = 4;
95      block.x = 16;
96      idKernel << < grid, block >> > (dev_a, dev_b, dev_c);
97      cudaDeviceSynchronize();
98      cudaMemcpy(host_a, dev_a, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
99      cudaMemcpy(host_b, dev_b, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
100     cudaMemcpy(host_c, dev_c, sizeof(int) * vectorSize,
           cudaMemcpyDeviceToHost);
101
102     printf("\nExecution 3: 4 block 16 threads \n");
103     printArray(host_a, vectorSize, "threadIdx.x");
104     printArray(host_b, vectorSize, "blockIdx.x");
```

```
    printArray(host_c, vectorSize, "globalId");

    free(host_a);
    free(host_b);
    free(host_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    return 0;
}
```

Output

```
ERROR 9: invalid configuration argument (Error at idKernel execution no. 1)
Execution 1: 1 block 64 threads
threadIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
blockIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
globalId: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Execution 2: 64 blocks 1 thread
threadIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
blockIdx.x: 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 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 50 51 52 53 54 55 56 57 58 59 60 61 62 63
globalId: 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 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 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Execution 3: 4 block 16 threads
threadIdx.x: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
blockIdx.x: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
globalId: 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 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 50 51 52 53 54 55 56 57 58 59 60 61 62 63
```

ERROR 9: Kernel no. 1 is not really executed

Practice

- `cudaDeviceSynchronize();` checks **if** there's some process in kernel to wait for, else it continues normally with host lines.

Lab 07

Code a program in c/c++ using CUDA in which you implement a kernel that inverts the order of the elements of an integer vector filled randomly, and that saves the values in another vector considering the requirements:

- 32 threads
- 1 block of 1 dimension
- The kernel must be: `__global__ void flipVector(int* vector, int* flippedVector)`
- Include error management using the following function: `__host__ void checkCUDAError(const char* msg)`

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

#include <stdlib.h> /* srand, rand */
#include <time.h> /* time */

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__global__ void flipVector(int* vector, int* flippedVector) {
    int gId = threadIdx.x + blockIdx.x * blockDim.x;
    flippedVector[(blockDim.x - 1) - gId] = vector[gId];
}

```

```
24 int main() {
25     const int vectorSize = 32;
26     int* vector = (int*)malloc(sizeof(int) * vectorSize);
27     int* flippedVector = (int*)malloc(sizeof(int) * vectorSize
          );
28
29     int* devVector, * devFlippedVector;
30     cudaMalloc((void**)&devVector, sizeof(int) * vectorSize);
31     checkCUDAError("cudaMalloc: devVector");
32     cudaMalloc((void**)&devFlippedVector, sizeof(int) *
          vectorSize);
33     checkCUDAError("cudaMalloc: devFlippedVector");
34
35     srand(time(NULL));
36     printf("Original vector: \n");
37     for (int i = 0; i < vectorSize; i++) {
38         int num = rand() % vectorSize + 1;
39         vector[i] = num;
40         printf("%d ", vector[i]);
41     }
42
43     cudaMemcpy(flippedVector, vector, sizeof(int) * vectorSize
          , cudaMemcpyHostToHost);
44     checkCUDAError("cudaMemcpy: vector -> flippedVector, Host
          -> Host");
45     cudaMemcpy(devVector, vector, sizeof(int) * vectorSize,
          cudaMemcpyHostToDevice);
46     checkCUDAError("cudaMemcpy: vector -> devVector, Host ->
          Device");
47     cudaMemcpy(devFlippedVector, flippedVector, sizeof(int) *
          vectorSize, cudaMemcpyHostToDevice);
48     checkCUDAError("cudaMemcpy: flippedVector ->
          devFlippedVector, Host -> Device");
49
50     dim3 grid(1);
51     dim3 block(vectorSize);
52
53     flipVector << < grid, block >> > (devVector,
          devFlippedVector);
54     checkCUDAError("kernel: flipVector");
55
56     cudaMemcpy(flippedVector, devFlippedVector, sizeof(int) *
          vectorSize, cudaMemcpyDeviceToHost);
57     checkCUDAError("cudaMemcpy: devFlippedVector ->
          flippedVector, Device -> Host");
58
59     printf("\nFlipped vector: \n");
60     for (int i = 0; i < vectorSize; i++) {
61         printf("%d ", flippedVector[i]);
62     }
63 }
```

Output

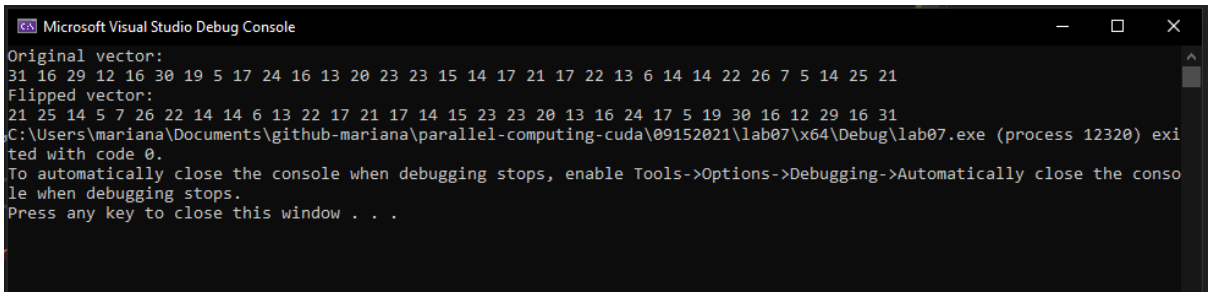


Figure 1: Image

Matrix Configurations

- The ideal is to configure the block/grid according to the data you need to process: when wroking with vectors, configure grid/blocks as vectors, when working with images, configure grid/blocks as matrix.
  - When working with one image, the block config must be a matrix too.

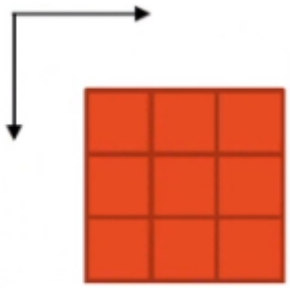


Figure 1: Image

- The kernel must always receive a **vector of information** as a parameter even though our data is a matrix, like an image. Inside the kernel we then need to unfold the matrix so that we can access an image like a vector. Thus, we need to transform the matrix to a vector only in the process of **data transference** between host and device, so that the device receives it as a vector then. But the kernel can be configured as a matrix:

```
1 dim3 grid(1);
2 dim3 block(3,3); // (3,3,1)
```

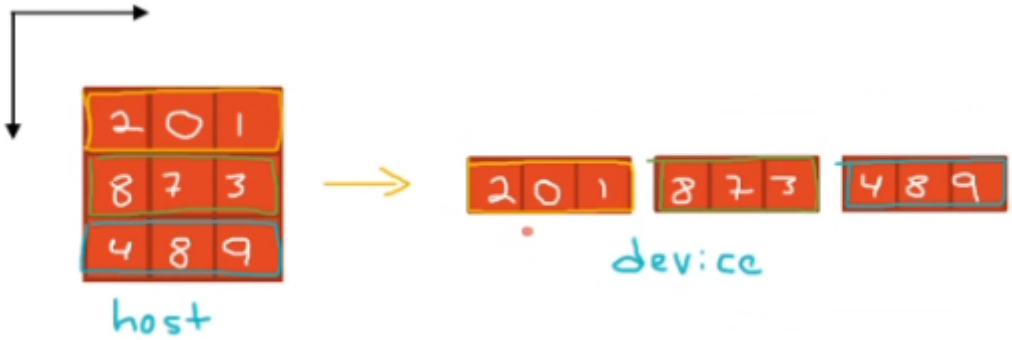


Figure 2: Image

- The idea is to send n x m threads for a n x m image.
- The coordinates to locate a thread in a single-block grid inside its 2D block are (`threadIdx.x`, `threadIdx.y`). So, if we have a config in the form of a matrix, we need

to unfold this 2D matrix block config in order to calculate the global ID, and this id will be used to access the vector we have as a param. The globalId now is calculated as:

> gId = threadIdx.x + threadIdx.y * blockDim.x

- The `threadIdx.y * blockDim.x` tells you how many rows to skip downwards through the 'Y' component.

Figure 3: Image

Example 01

Sum of matrices

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  __host__ void checkCUDAError(const char* msg) {
8      cudaError_t error;
9      cudaDeviceSynchronize();
10     error = cudaGetLastError();
11     if (error != cudaSuccess) {
12         printf("ERROR %d: %s (%s)\n", error,
               cudaGetErrorString(error), msg);
13     }
14 }
15
16 __global__ void matrixSum(int* dev_a, int* dev_b, int* dev_c)
       {
17     int gId = threadIdx.x + threadIdx.y * blockDim.x;
18     dev_c[gId] = dev_a[gId] + dev_b[gId];
19 }
20
21 int main() {
22     const int N = 3; // if 32 ok, if 33 ERROR 9: invalid
           configuration argument (matrixSum kernel error) and c
           mat is zeroed
23
```

```
24      int* host_a = (int*)malloc(sizeof(int) * N * N);
25      int* host_b = (int*)malloc(sizeof(int) * N * N);
26      int* host_c = (int*)malloc(sizeof(int) * N * N);
27
28      int* dev_a, * dev_b, * dev_c;
29      cudaMalloc((void**)&dev_a, sizeof(int) * N * N);
30      cudaMalloc((void**)&dev_b, sizeof(int) * N * N);
31      cudaMalloc((void**)&dev_c, sizeof(int) * N * N);
32
33      // init data
34      for (int i = 0; i < N * N; i++) {
35          host_a[i] = (int)(rand() % 10);
36          host_b[i] = (int)(rand() % 10);
37      }
38
39      cudaMemcpy(dev_a, host_a, sizeof(int) * N * N,
           cudaMemcpyHostToDevice);
40      cudaMemcpy(dev_b, host_b, sizeof(int) * N * N,
           cudaMemcpyHostToDevice);
41
42      dim3 block(N, N);
43      dim3 grid(1);
44
45      matrixSum << < grid, block >> > (dev_a, dev_b, dev_c);
46      checkCUDAError("matrixSum kernel error");
47
48      cudaMemcpy(host_c, dev_c, sizeof(int) * N * N,
           cudaMemcpyDeviceToHost);
49
50      printf("\nMatrix A: \n");
51      for (int i = 0; i < N; i++) {
52          for (int j = 0; j < N; j++) {
53              printf("%d ", host_a[j + i * N]);
54          }
55          printf("\n");
56      }
57
58      printf("\nMatrix B: \n");
59      for (int i = 0; i < N; i++) {
60          for (int j = 0; j < N; j++) {
61              printf("%d ", host_b[j + i * N]);
62          }
63          printf("\n");
64      }
65
66      printf("\nMatrix C: \n");
67      for (int i = 0; i < N; i++) {
68          for (int j = 0; j < N; j++) {
69              printf("%d ", host_c[j + i * N]);
70          }
71          printf("\n");
```

```
72      }
73
74      free(host_a);
75      free(host_b);
76      free(host_c);
77      cudaFree(dev_a);
78      cudaFree(dev_b);
79      cudaFree(dev_c);
80  }
```

- The property `maximumThreadsPerBlock` will tell us how big the matrix can be in order to have a thread per cell. For a 1024 limit, the matrix would be 32 x 32.

Image Processing: blur mask

The objective is, given a matrix of information, apply a blur filter using that matrix.

- Without considering the borders:

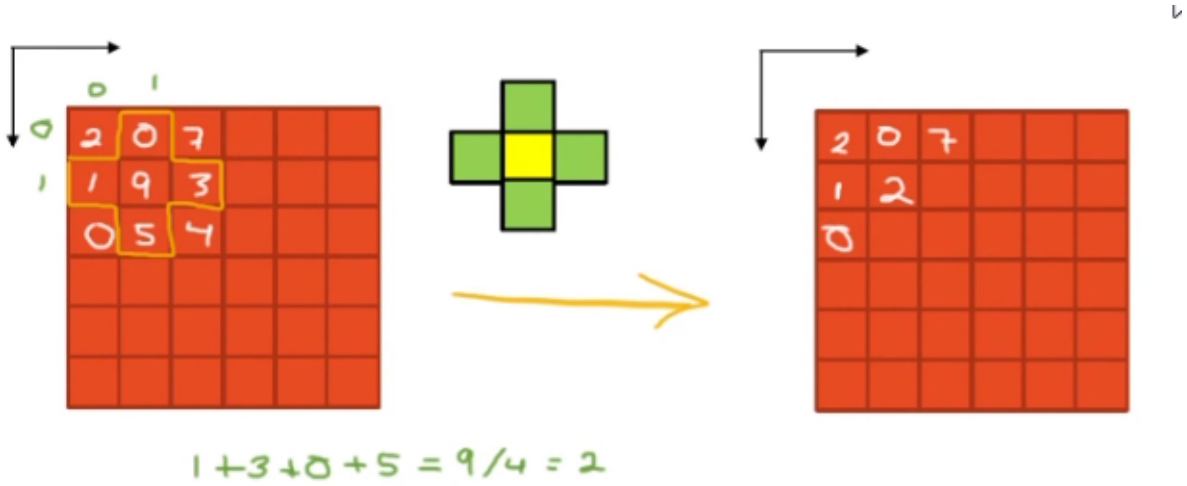


Figure 4: Image

- The border threads will not do anything, and this is managed by knowing its global Id. In order to do that, we need to unfold the matrix block config as a vector

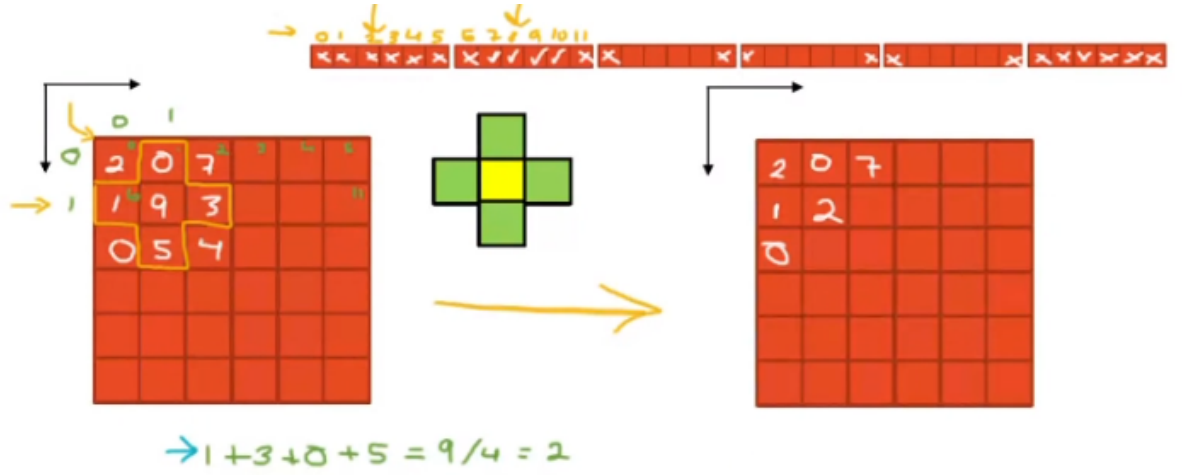


Figure 5: Image

- Now, we will use the calculated gId to get the element of the vector of information that we need to sum as a neighbour, so that each cell can now contain the average of its four neighbours.

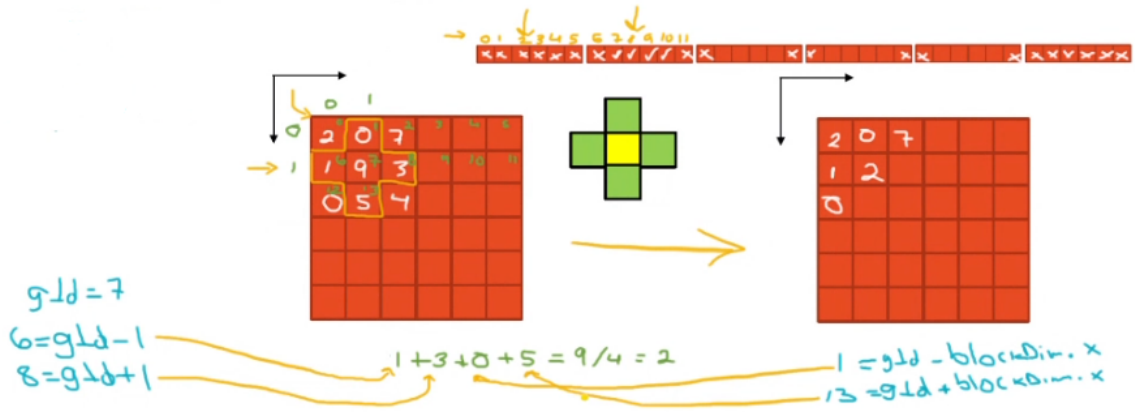


Figure 6: Image

Matrix Configurations: Practice

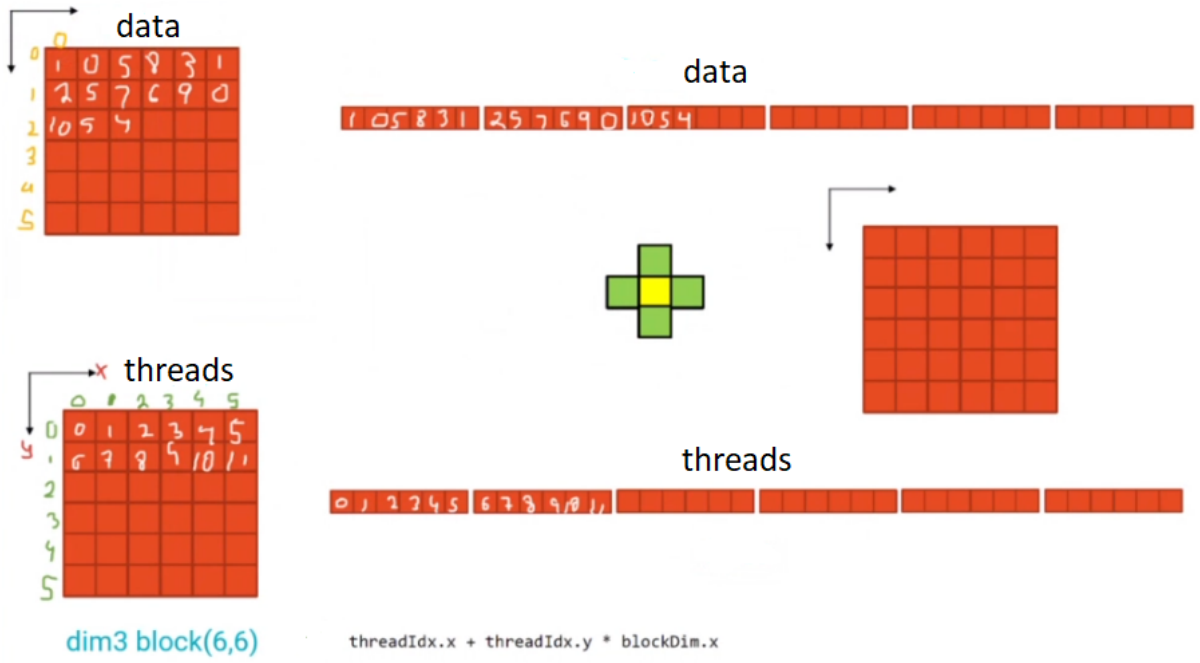


Figure 1: Image

- When we have information in matrix form, it is convenient to configure the threads in a block (block's config) as a matrix too, so that processing is easier.
- The task here is to compute the gId out of a 2D block config in order to access the vector parameter (matrix of info) that is inside the device.
- Each thread will process one cell of the matrix of information.
- Each line of code inside the kernel will be executed N times in parallel, through the N threads.
- The gId will be used to index both the information vector and the result vector inside the kernel.

Lab 08

Code a program in c/c++ using CUDA in which you implement a kernel that calculates the values of a matrix B considering the average of the 4 neighbours with respect to the information of a matrix A, and considering the requirements:

- 36 threads
- 1 2D block of 6 x 6 threads
- A and B matrices of size 6 x 6

- Matrix A must initialized with random integer values from 0 to 9
- Include error management with a function `__host__ void checkCUDAError(const char* msg)`

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__global__ void kernel(int* m, int* r) {
    int gId = threadIdx.x + threadIdx.y * blockDim.x;
    int n1 = gId - 1;
    int n2 = gId + 1;
    int n3 = gId - blockDim.x;
    int n4 = gId + blockDim.x;
    if (threadIdx.x == 0 || threadIdx.x == (blockDim.x - 1) ||
        threadIdx.y == 0 || threadIdx.y == (blockDim.y - 1)) {
        r[gId] = m[gId];
    }
    else {
        int avg = (m[n1] + m[n2] + m[n3] + m[n4]) / 4;

        r[gId] = avg;
    }
}

int main() {
    const int size = 6;

    int m[size][size] = { 0 };
    int r[size][size] = { 0 };
    int m_vec[size * size] = { 0 };
    int r_vec[size * size] = { 0 };

    int* dev_m, * dev_r;
    cudaMalloc((void**)&dev_m, sizeof(int) * size * size);
```

```
42      checkCUDAError("Error at cudaMalloc for dev_m");
43      cudaMalloc((void**)&dev_r, sizeof(int) * size * size);
44      checkCUDAError("Error at cudaMalloc for dev_r");
45
46      for (int i = 0; i < size; i++) {
47          for (int j = 0; j < size; j++) {
48              m[i][j] = (int)(rand() % 10);
49              m_vec[j + i * size] = m[i][j];
50          }
51      }
52
53      printf("Original Matrix:\n");
54      for (int i = 0; i < size; i++) {
55          for (int j = 0; j < size; j++) {
56              printf("%d ", m[i][j]);
57          }
58          printf("\n");
59      }
60
61      for (int i = 0; i < size * size; i++) {
62          //printf("%d ", m_vec[i]);
63      }
64      printf("\n");
65
66      cudaMemcpy(dev_m, m_vec, sizeof(int) * size * size,
           cudaMemcpyHostToDevice);
67      checkCUDAError("Error at cudaMemcpy Host -> Device");
68
69      dim3 grid(1);
70      dim3 block(size, size);
71      kernel << < grid, block >> > (dev_m, dev_r);
72      checkCUDAError("Error at kernel");
73
74      cudaMemcpy(r_vec, dev_r, sizeof(int) * size * size,
           cudaMemcpyDeviceToHost);
75      checkCUDAError("Error at cudaMemcpy Device -> Host");
76
77      printf("Average Matrix:\n");
78      for (int i = 0; i < size; i++) {
79          for (int j = 0; j < size; j++) {
80              r[i][j] = r_vec[j + i * size];
81              printf("%d ", r[i][j]);
82          }
83          printf("\n");
84      }
85
86      cudaFree(dev_m);
87      cudaFree(dev_r);
88  }
```

Output

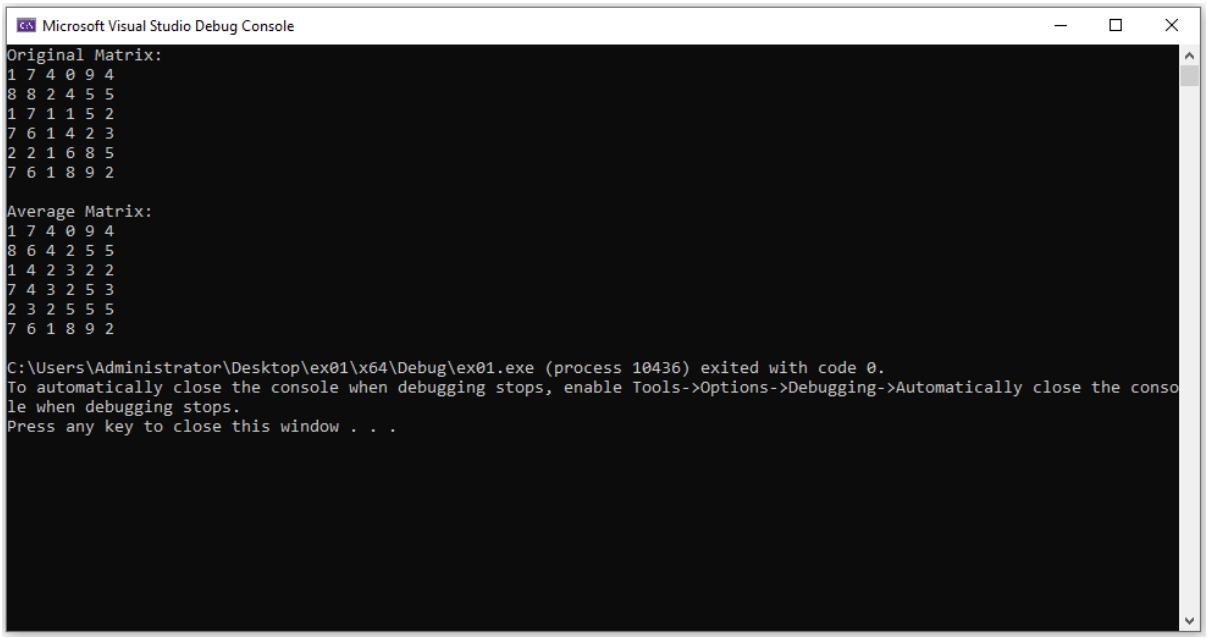


Figure 2: Image

Warps and Thread Launching

- A kernel config in the form of a matrix block is appropiate when our data has the same form of a matrix. It is appropiate because with the same way that we identify the threads launched through the kernel, that same index that allows us to identify threads is used also to identify the data we want to process. Therefore, it is doubly convenient: threads in the same form that our data.
- When we work with big matrices, we cannot use just one block, because it can only be 32 x 32. We need to use various blocks: with which configs? Through which axes?

Warps

- There is a property that shows you the size of the warp (warpSize): a warp is a **basic unit** constituted of 32 consecutive threads, and it will aid us to decide the block config to use.
- These 32 consecutive threads inside a warp use the SIMT modality (Single Instruction Multiple Threads) when executed in parallel: One instruction executed by multiple (each) threads. This means that each thread will execute the same instructions but with different data, its **own** variables or memory.
- All threads inside a block are divided into warps. Each thread will be executed in a streaming processor (SP) or CUDA Core or Nucleus. **Each block is executed in a Streaming Multiprocessor (SM), which has 32 small squares (SP) / 1 warp, if this block qty is exceeded, waiting time is required.** If a SM has 64 SP (2 warps), that means that the SM has the capacity to run 2 warps in real parallel.

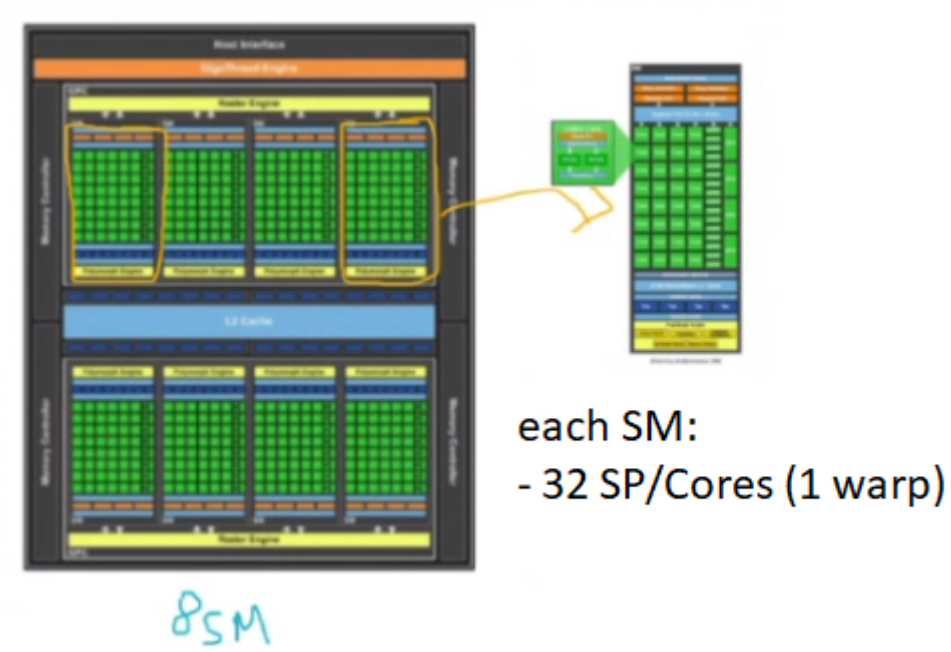


Figure 1: Image

Therefore, each SM can process in parallel 32 threads (1 warp).

- Only 32(SP per SM) x 8(SM's) = 256 threads will be run in real parallelism using all GPU resources on this 8SM/32 SP example. Otherwise (more threads launched), there will be some waiting time.
- A warp is a **basic unit** that will help us with the decision of which block config to use. Each SM is divided into warps.

Figure 2: Image

- Imagine we have 8 blocks with 128 threads each = 1024 threads, when in reality we have 8 (SM) x 32 (SP) = 256 in real parallel. Thus, each of those 128 threads are divided into warps: 4 warps per block. Each block will be divided into 4, and what will happen is that each warp of 32 threads will be taken and so on, until you run the 128 threads in 8 SM's, giving the 1024 threads you wanted in total.

  ### 1 block, 128 threads

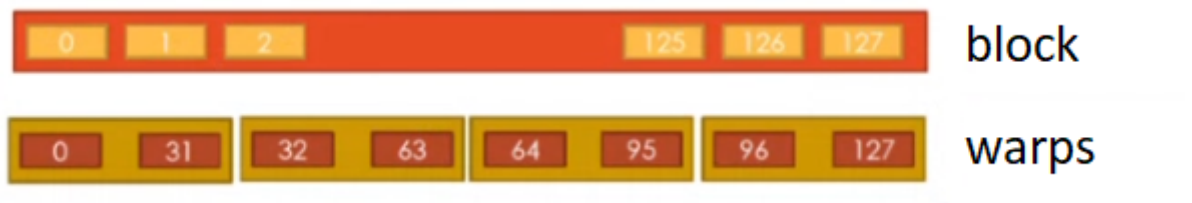


Figure 3: Image

Note: a) in this config, all warps will be executed (maybe in line) in 1 SM unit. Slower, 1 SM for 128 (you wait 4 times).

- You need 4 warps to execute a 128 thread block: a SM can only run 1 warp (32 th) at the same time. If you config a warp per SM (or more, depending on how many cores the SM has), all SM's will run its warp in parallel and avoid the waiting time.
- An SM has a number of threads, which will be grouped in warps. The SM runs all its threads (warps) in parallel. In this examples, we are saying that an SM runs only 32 threads or 1 warp in parallel.

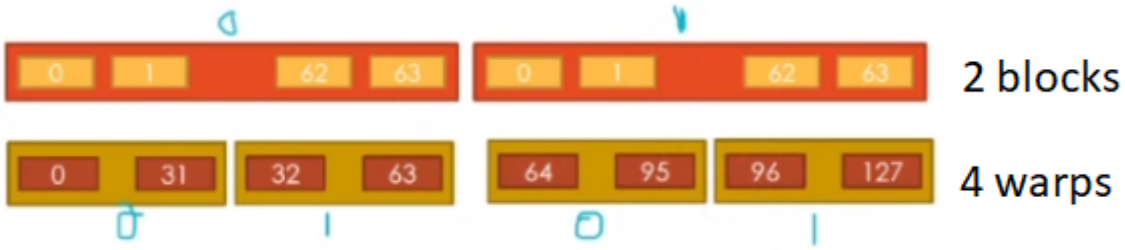


Figure 4: Image

Note: b) in this config, the first 2 warps will be executed in 1 SM, and the other 2 warps in another SM, because they are in different blocks. Faster, 1 SM for 64 (you wait 2 times). Thus, if we launch 4 blocks with 32 threads each, we would be executing all in parallel (32 threads per SM) ans still launch 128 threads. All SM's run independently their warps (block). The two-block config is slower because you're giving 64 threads to 2 SM's that run 1 warp, and in the four-block config you're giving 32 threads to 4 SM's that run 1 warp.

- Warps continue the gIds from the past warp executed: each thread will have its global id following the last id from the last thread in the last warp executed. In the end, **all threads are put along the x axis with consecutive global ids**.

- The block config affects the execution time: it is advised that the block config is always a multiple of 32 (warp): you can loose time and also waste threads launched because of other configs.
- In order to accelerate the process, try to divide the threads into the most SM units (blocks) as posible. Try also to **always have a block config that is a multiple of 32**. What if not?
  - If we launch 2 block with 40 threads each (80 total threads), we will need 4 warps, and not 3. This because **each block is divided separately into warps**. If you have 1 block with 40 threads, you will need on *that* block 2 warps: 32 in one and 8 in another. Same thing with the next block = 4 warps.

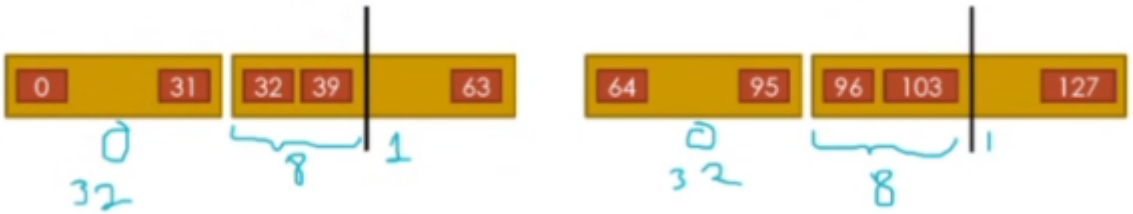


Figure 5: Image

  - Therefore, to launch 80 threads with this config we will need 4 warps = 128 threads, with 48 inactive threads. When we do not configure in 32 multiples, we use more threads than necessary: this is why we need multiples of 32, so that there are **no wasted** threads. Even if you launch a kernel with a single thread block, CUDA will need 1 warp / 32 threads.

Lab 09

If we were to launch 4 block with 42 threads each, how many warps would we need? The answer is 8 warps. Therefore, we would be reserving 256 threads to launch 168 active threads.

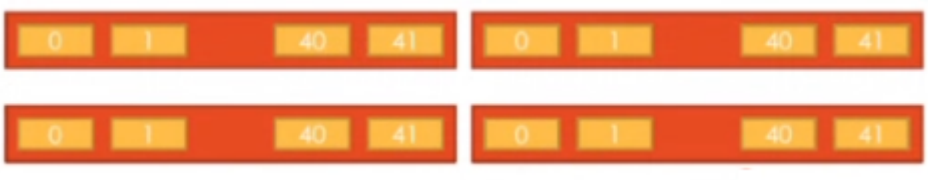


Figure 6: Image

Write a program in c/c++ using CUDA in which you implement a kernel that prints the following info for each thread:

- The thread index (threadIdx.x)
- The block index along x (blockIdx.x)
- The block index along y (blockIdx.y)
- The global index (gId)
- The index of the warp that it belongs to (warpId)
- The global block index (gBlockId)

Consider the following requirements for the launch of the kernel:

- Blocks of 42 threads in 1D
- A grid of 2 x 2 blocks
- The kernel must be: `__global__ void warpDetails()`

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__global__ void warpDetails() {
    int gId = blockIdx.y * (gridDim.x * blockDim.x) + (
       blockIdx.x * blockDim.x) + threadIdx.x;
    int warpId = threadIdx.x / 32; // index of warp per block,
        not unique
    int gBlockId = blockIdx.y * gridDim.x + blockIdx.x;
    printf("threadIdx.x: %d blockIdx.x: %d blockIdx.y: %d gId:
        %d warpId: %d gBlockId: %d\n", threadIdx.x, blockIdx.x
       , blockIdx.y, gId, warpId, gBlockId);
}

int main() {
    dim3 block(42);
    dim3 grid(2, 2);
```

```
26      warpDetails << < grid, block >> > ();
27      checkCUDAError("Error at kernel");
28
29      return 0;
30  }
```

Output

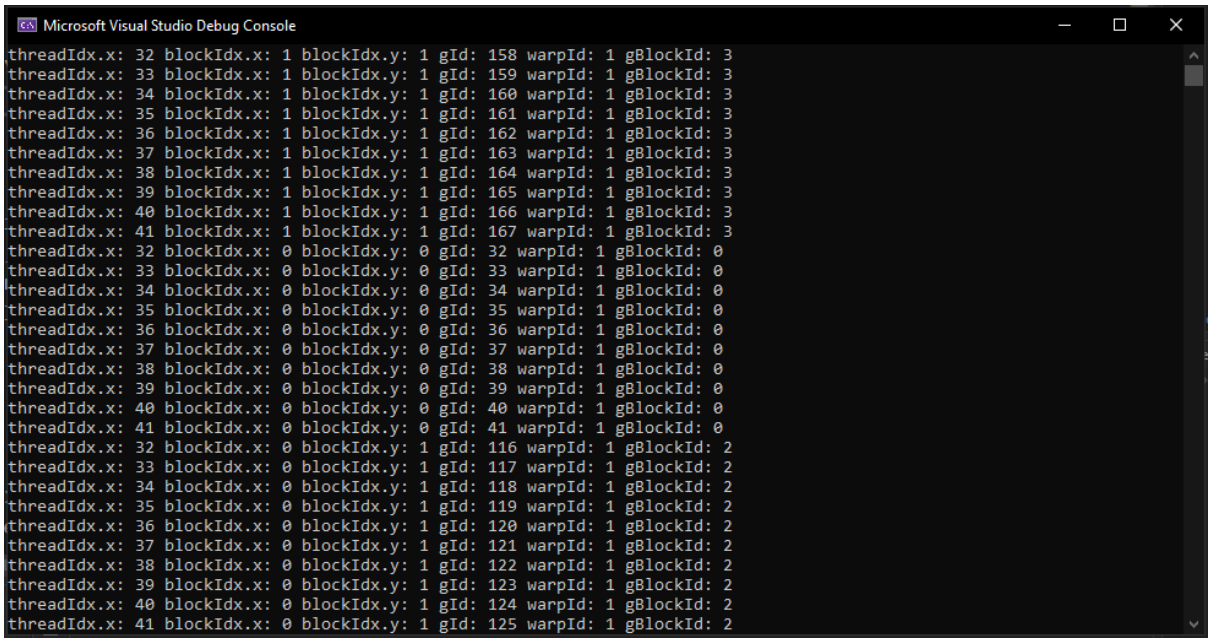


Figure 7: Image

GPU Specs Meaning

Summary

Basics

- 1 warp is the basic unit and it is constituted of 32 consecutive threads.
- All threads in a warp are executed in parallel using the modality SIMT (Single Instruction Multiple Thread).
  - SIMT means that every thread executes the same instruction but using their own copy of data, their own memory.
- The threads in a block are divided into warps.
- Each block is executed in one SM unit (Streaming Multiprocessor / Processor / Multiprocessor).
- Each thread is executed in one core.

Specifics

- Even if a block is launched with only one thread, CUDA will assign a warp of 32 threads, and so only one thread will be active and 32 threads inactive.
- Even though threads in a warp are inactive, CUDA reserves resources for all of them, that is, the resources are reserved for the whole warp.
- To have inactive threads in a warp means a lot of wasted resources.
- It is recommended to configure the number of threads in a block (block config) in multiples of 32 to assure that all threads are in an active state.

Example 1

- Due to the fact that the info is now an image (matrix), the amount of operations to perform is n x m, in this case, 128 x 96 = 12 288 pixels of information.
- Let's divide the image into blocks, where each block contains or processes a certain amount of pixels. Each block would be a 32 x 32 matrix of image pixels. In this way, we obtain 4 x 3 blocks of 32 by 32 each. The grid config for the kernel launch would be `grid(3, 4, 1)` and the idea is to use each SM to process a block of 32 x 32 (1024 threads) in parallel.

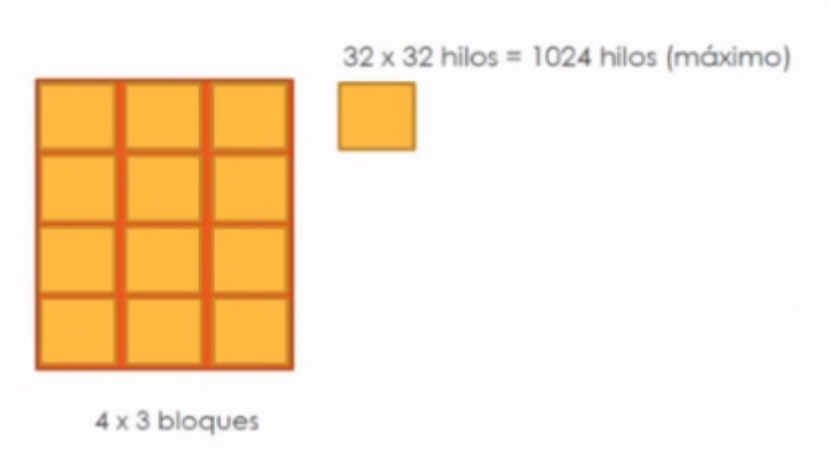


Figure 1: Image

Note: another config could be 12 1D blocks of 1024 (multiple of 32) threads along the x or y axis

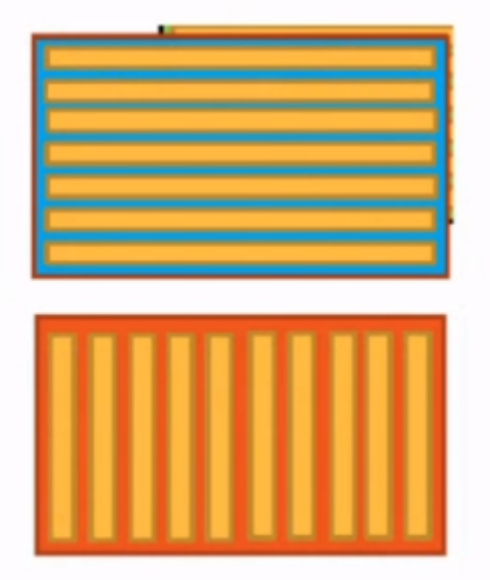


Figure 2: Image

- If all configs use the maximum amount of resources without wasting threads (blocks multiples of 32), then you choose the one that is more natural to program.

Example 2

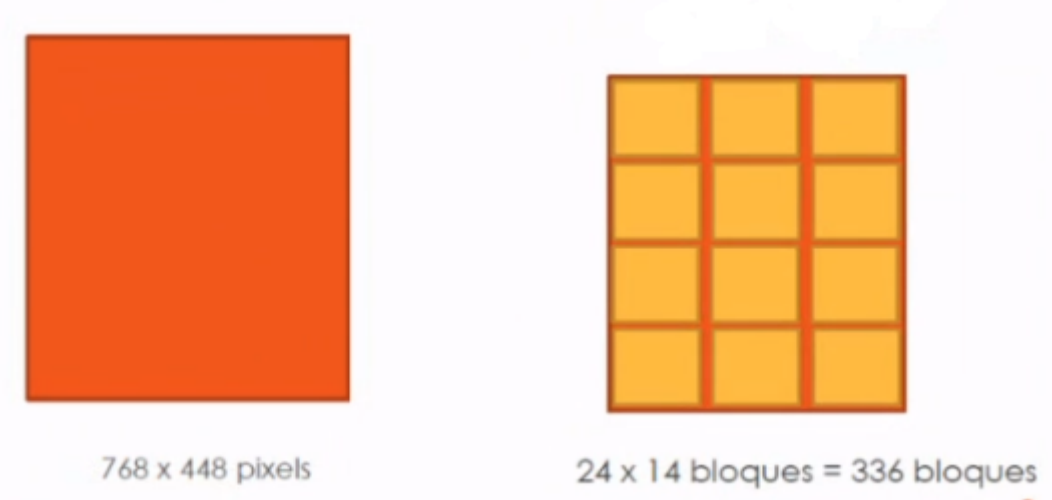


Figure 3: Image

- In this image, we now the grid config (block quantities per axis) just by dividing the dimensions 768 x 448 by 32 and we get `grid(14, 24, 1)`. Thus, 14 x 24 = 336 square blocks of 32 x 32. The 32 x 32 = 1024, which are the threads in `maxThreadsPerBlock` or in a block, not per SM. Each of these blocks would be executed by one SM, because of i). We only have 16 SM and we have 336 blocks, so in each multiprocessor there will be 21 blocks per SM, **with some waiting time**: virtually or in software would be parallel, but not at hardware level. At hardware or real parallel, it must be the number given by NVIDIA.

Know Your Specs

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <iostream>
6
7  using namespace std;
8  __host__ void checkCUDAError(const char* msg) {
9      cudaError_t error;
10     cudaDeviceSynchronize();
11     error = cudaGetLastError();
12     if (error != cudaSuccess) {
13         printf("ERROR %d: %s (%s)\n", error,
               cudaGetErrorString(error), msg);
14     }
15 }
16
```

```
17 int main()
18 {
19     cudaDeviceProp properties;
20     cudaGetDeviceProperties(&properties, 0);
21     // Device name: NVIDIA GeForce GTX 1080
22     cout << "name: " << properties.name << endl;
23     // 2560 Cores https://www.nvidia.com/es-la/geforce/
          products/10series/geforce-gtx-1080/
24     cout << "CUDA Cores: " << 2560 << endl;
25     // SM units/ Multiprocessors:
26     cout << "multiProcessorCount: " << properties.
          multiProcessorCount << endl; // 20
27     cout << "Cores per Multiprocessor: " << 2560 / properties.
          multiProcessorCount << endl; // 128
28     cout << "maxThreadsPerMultiProcessor: " << properties.
          maxThreadsPerMultiProcessor << endl; // 2048
29     cout << "maxBlocksPerMultiProcessor: " << properties.
          maxBlocksPerMultiProcessor << endl; // 32
30     return 0;
31 }
```

Careful with Your Configs

a) name	b) NVIDIA CUDA Cores (webpage)	c) multiProcessorCount (SM units)	d) Cores / SM	e) maxThreadsPerMultiProcessor	f) maxBlocksPerMultiProcessor
NVIDIA GeForce GTX 1650	1024	16	64	1024	16
NVIDIA GeForce GTX 1080	2560	20	128	2048	32
NVIDIA GeForce GTX 960M	640	5	128	2048	32

Total Capacity

NVIDIA GeForce GTX 1650	NVIDIA GeForce GTX 1080	NVIDIA GeForce GTX 960M
1 block of 1024 / SM	1 block of 2048 / SM	1 block of 2048 / SM
16 blocks of 64 / SM (16 x 64 = 1024)	32 blocks of 64 / SM (32 x 64 = 2048)	32 blocks of 64 / SM (32 x 64 = 2048)
16 blocks of 1024 total = 16 384	20 blocks of 2048 total = 40 960	5 blocks of 2048 total = 10 240
256 blocks of 64 total = 16 384	640 (32 x 20) blocks of 64 total = 40 960	160 (32 x 5) blocks of 64 total = 10 240
16 blocks of 64 (d) = 1024 **real parallel**	20 blocks of 128 (d) = 2560 **real parallel**	5 blocks of 128 (d) = 640 **real parallel**

- CUDA Cores Per SM (Multiprocessor) = 1024 (b) / 16 (c) = 64 cores per SM. This tells us that we can execute in **real parallel** 2 warps per SM *.
  - `maxThreadsPerMultiprocessor` = 1024 tells us that we can process more than 64 threads as a maximum limit per SM. This 1024 number means that:
    - i) we can launch 1 block of 1024 threads per SM. But also because of `maxBlocksPerMultiprocessor` = 16, we can also launch 16 blocks that do not supass 1024 threads. Thus, also ii) 16 blocks of 64 threads (*coinciding with the core capacity of 64/2 warps) per SM. But these exceed the 64 threads in real parallel, so these two configs would be executed **within some waiting time**.
    - i) `multiProcessorCount` = 16, plus the above fact of `maxThreadsPerMultiprocessor` = 1024, we can launch 16 blocks of 1024 threads each **in total** as maximum capacity, which sums a total of 16384 threads. ii) With 16 blocks per SM (also 16), we can have a total of 256 blocks of 64 threads each block **in total** as well, summing 16384 threads. These two totals also mean **within some waiting time**.
    - In real parallel? 16 blocks (one per SM) with 64 threads, 16 x 64 = 1024. Each block would process 64 threads in real parallel. This 1024 must coincide with CUDA Cores info from NVIDIA.
  - A block is executed in an SM unit. But, by having `maxBlocksPerMultiprocessor`, we conclude that we can execute more than one block, with waiting time.

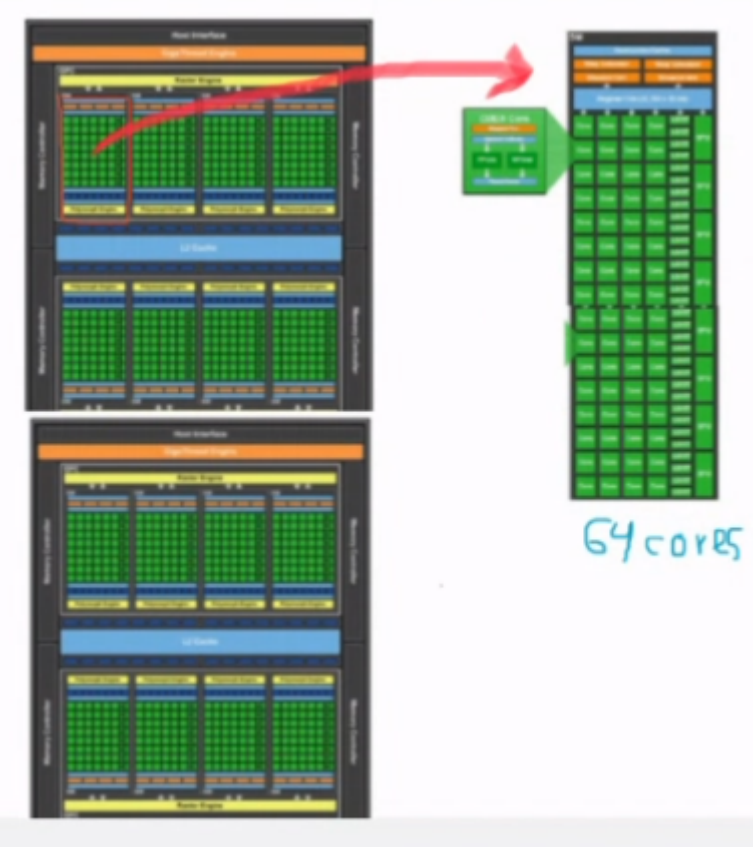


Figure 4: Image

Handy Links

- https://www.nvidia.com/en-us/geforce/gaming-laptops/geforce-gtx-960m/specifications/

Practice

- The gId is calculated from our configs, and therefore inactive threads do not interfere in the calculation of gId's. If you launch 2 blocks with 40 threads, even if you use 4 warps, the thread gIds will go from 0 to 79, for example.

Exercise

Now we will process our image of 768 x 448 pixels, divided into 14,24 blocks. For now, the image is read as grayscale.

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__global__ void complement(uchar* dev_a, uchar* dev_b) {
    // locate my current block row
    int threads_per_block = blockDim.x * blockDim.y;
    int threads_per_row = threads_per_block * gridDim.x;
    int row_offset = threads_per_row * blockIdx.y;

    // locate my current block column
    int block_offset = blockIdx.x * threads_per_block;
    int threadId_inside = blockDim.x * threadIdx.y + threadIdx
       .x;

    int gId = row_offset + block_offset + threadId_inside;
    dev_b[gId] = 255 - dev_a[gId];
}

using namespace cv;
int main() {
```

```
33
34      Mat img = imread("antenaParalelo.jpg", IMREAD_GRAYSCALE);
35
36      const int R = img.rows;
37      const int C = img.cols;
38
39      Mat imgResult(img.rows, img.cols, img.type());
40      uchar* host_a, * host_b, * dev_a, * dev_b, * pImg;
41      host_a = (uchar*)malloc(sizeof(uchar) * R * C);
42      host_b = (uchar*)malloc(sizeof(uchar) * R * C);
43      cudaMalloc((void**)&dev_a, sizeof(uchar) * R * C);
44      checkCUDAError("Error at malloc dev_a");
45      cudaMalloc((void**)&dev_b, sizeof(uchar) * R * C);
46      checkCUDAError("Error at malloc dev_b");
47
48      // matrix as vector
49      for (int i = 0; i < R; i++) {
50          pImg = img.ptr<uchar>(i); // points to a row each time
51          for (int j = 0; j < C; j++) {
52              host_a[i * C + j] = pImg[j];
53          }
54      }
55      cudaMemcpy(dev_a, host_a, sizeof(uchar) * R * C,
           cudaMemcpyHostToDevice);
56
57      dim3 block(32, 32);
58      dim3 grid(C / 32, R / 32);
59
60      complement << < grid, block >> > (dev_a, dev_b);
61      checkCUDAError("Error at kernel");
62
63      cudaMemcpy(host_b, dev_b, sizeof(uchar) * R * C,
           cudaMemcpyDeviceToHost);
64
65      for (int i = 0; i < R; i++) {
66          pImg = imgResult.ptr<uchar>(i);
67          for (int j = 0; j < C; j++) {
68              pImg[j] = host_b[i * C + j];
69          }
70      }
71
72      imshow("Image", img);
73      imshow("Image Result", imgResult);
74      waitKey(0);
75
76      free(host_a);
77      free(host_b);
78      cudaFree(dev_a);
79      cudaFree(dev_b);
80
81      return 0;
```

```
82 }
```

RGB Image Manipulation Using CUDA: Practice

To process an image in RGB, we just need to apply what we did for a grayscale image but three times, each for an RGB channel: have a matrix (vector) of data for channel R, G and B.

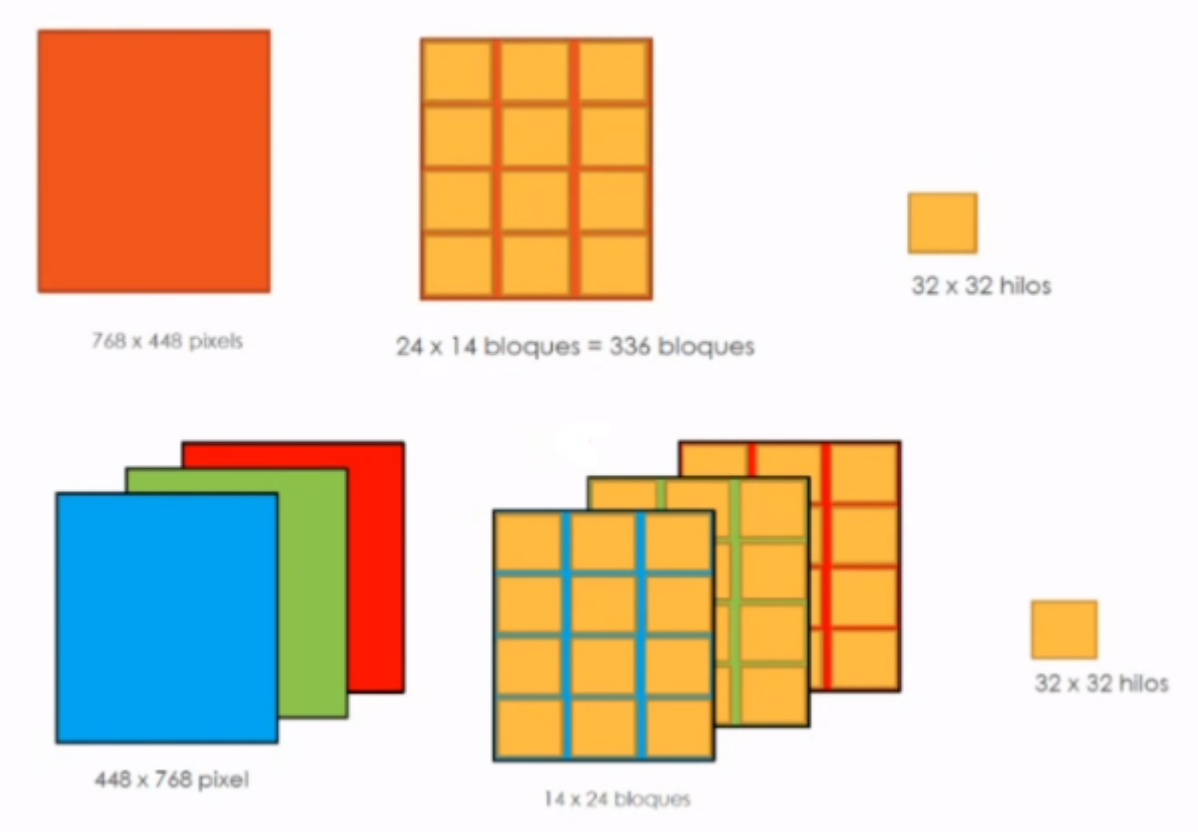


Figure 1: image

Lab 11

Write a program in c/c++ using CUDA in which you implement a kernel to modify the contrast of an RGB image, and another to modify the brightness, considering the following requirements:

- Blocks of 32 x 32 threads
- The kernel for complement, contrast and brightness should be:
  - `__global__ void complement(uchar* R, uchar* G, uchar* B)`
  - `__global__ void contrast(uchar* R, uchar* G, uchar* B, float fc)`
  - `__global__ void brightness(uchar* R, uchar* G, uchar* B, float fb)`
- Include error management with a function:

- `__host__ void checkCUDAError(const char* msg)`

Input

Figure 2: img

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__global__ void complement(uchar* R, uchar* G, uchar* B) {
    // locate my current block row
    int threads_per_block = blockDim.x * blockDim.y;
```

```
20      int threads_per_row = threads_per_block * gridDim.x;
21      int row_offset = threads_per_row * blockIdx.y;
22
23      // locate my current block column
24      int block_offset = blockIdx.x * threads_per_block;
25      int threadId_inside = blockDim.x * threadIdx.y + threadIdx
            .x;
26
27      int gId = row_offset + block_offset + threadId_inside;
28      R[gId] = 255 - R[gId];
29      G[gId] = 255 - G[gId];
30      B[gId] = 255 - B[gId];
31  }
32
33  __global__ void contrast(uchar* R, uchar* G, uchar* B, float
        fc) {
34      // locate my current block row
35      int threads_per_block = blockDim.x * blockDim.y;
36      int threads_per_row = threads_per_block * gridDim.x;
37      int row_offset = threads_per_row * blockIdx.y;
38
39      // locate my current block column
40      int block_offset = blockIdx.x * threads_per_block;
41      int threadId_inside = blockDim.x * threadIdx.y + threadIdx
            .x;
42
43      int gId = row_offset + block_offset + threadId_inside;
44      if (fc * R[gId] > 255) { R[gId] = 255; } else { R[gId] =
            fc * R[gId]; }
45      if (fc * G[gId] > 255) { G[gId] = 255; } else { G[gId] =
            fc * G[gId]; }
46      if (fc * B[gId] > 255) { B[gId] = 255; } else { B[gId] =
            fc * B[gId]; }
47  }
48
49  __global__ void brightness(uchar* R, uchar* G, uchar* B, float
        fb) {
50      // locate my current block row
51      int threads_per_block = blockDim.x * blockDim.y;
52      int threads_per_row = threads_per_block * gridDim.x;
53      int row_offset = threads_per_row * blockIdx.y;
54
55      // locate my current block column
56      int block_offset = blockIdx.x * threads_per_block;
57      int threadId_inside = blockDim.x * threadIdx.y + threadIdx
            .x;
58
59      int gId = row_offset + block_offset + threadId_inside;
60      if (fb >= 0){
61      if (fb + R[gId] > 255) { R[gId] = 255; } else { R[gId] =
            fb + R[gId]; }
```

```
62      if (fb + G[gId] > 255) { G[gId] = 255; } else { G[gId] =
            fb + G[gId]; }
63      if (fb + B[gId] > 255) { B[gId] = 255; } else { B[gId] =
            fb + B[gId]; }
64      }
65
66      if (fb < 0) {
67      if (fb + R[gId] < 0) { R[gId] = 0; } else { R[gId] = fb +
            R[gId]; }
68      if (fb + G[gId] < 0) { G[gId] = 0; } else { G[gId] = fb +
            G[gId]; }
69      if (fb + B[gId] < 0) { B[gId] = 0; } else { B[gId] = fb +
            B[gId]; }
70      }
71  }
72
73  using namespace cv;
74  int main() {
75
76      Mat img = imread("antenaRGB.jpg");
77
78      const int R = img.rows;
79      const int C = img.cols;
80
81      Mat imgComp(img.rows, img.cols, img.type());
82      Mat imgCont(img.rows, img.cols, img.type());
83      Mat imgBright(img.rows, img.cols, img.type());
84      uchar* host_r, * host_g, * host_b, * dev_r1, * dev_g1, *
            dev_b1, * dev_r2, * dev_g2, * dev_b2, * dev_r3, *
            dev_g3, * dev_b3;
85      host_r = (uchar*)malloc(sizeof(uchar) * R * C);
86      host_g = (uchar*)malloc(sizeof(uchar) * R * C);
87      host_b = (uchar*)malloc(sizeof(uchar) * R * C);
88
89      cudaMalloc((void**)&dev_r1, sizeof(uchar) * R * C);
90      checkCUDAError("Error at malloc dev_r1");
91      cudaMalloc((void**)&dev_g1, sizeof(uchar) * R * C);
92      checkCUDAError("Error at malloc dev_g1");
93      cudaMalloc((void**)&dev_b1, sizeof(uchar) * R * C);
94      checkCUDAError("Error at malloc dev_b1");
95
96      cudaMalloc((void**)&dev_r2, sizeof(uchar) * R * C);
97      checkCUDAError("Error at malloc dev_r2");
98      cudaMalloc((void**)&dev_g2, sizeof(uchar) * R * C);
99      checkCUDAError("Error at malloc dev_g2");
100     cudaMalloc((void**)&dev_b2, sizeof(uchar) * R * C);
101     checkCUDAError("Error at malloc dev_b2");
102
103     cudaMalloc((void**)&dev_r3, sizeof(uchar) * R * C);
104     checkCUDAError("Error at malloc dev_r3");
105     cudaMalloc((void**)&dev_g3, sizeof(uchar) * R * C);
```

```
106     checkCUDAError("Error at malloc dev_g3");
107     cudaMalloc((void**)&dev_b3, sizeof(uchar) * R * C);
108     checkCUDAError("Error at malloc dev_b3");
109
110     // matrix as vector
111     for (int i = 0; i < R; i++) {
112         for (int j = 0; j < C; j++) {
113             Vec3b pix = img.at<Vec3b>(i, j);
114             host_r[i * C + j] = pix[2];
115             host_g[i * C + j] = pix[1];
116             host_b[i * C + j] = pix[0];
117         }
118     }
119     cudaMemcpy(dev_r1, host_r, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
120     checkCUDAError("Error at memcpy host_r -> dev_r1");
121     cudaMemcpy(dev_g1, host_g, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
122     checkCUDAError("Error at memcpy host_r -> dev_g1");
123     cudaMemcpy(dev_b1, host_b, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
124     checkCUDAError("Error at memcpy host_r -> dev_b1");
125
126     cudaMemcpy(dev_r2, host_r, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
127     checkCUDAError("Error at memcpy host_r -> dev_r2");
128     cudaMemcpy(dev_g2, host_g, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
129     checkCUDAError("Error at memcpy host_r -> dev_g2");
130     cudaMemcpy(dev_b2, host_b, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
131     checkCUDAError("Error at memcpy host_r -> dev_b2");
132
133     cudaMemcpy(dev_r3, host_r, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
134     checkCUDAError("Error at memcpy host_r -> dev_r3");
135     cudaMemcpy(dev_g3, host_g, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
136     checkCUDAError("Error at memcpy host_r -> dev_g3");
137     cudaMemcpy(dev_b3, host_b, sizeof(uchar) * R * C,
        cudaMemcpyHostToDevice);
138     checkCUDAError("Error at memcpy host_r -> dev_b3");
139
140     dim3 block(32, 32);
141     dim3 grid(C / 32, R / 32);
142
143     complement << < grid, block >> > (dev_r1, dev_g1, dev_b1);
144     cudaDeviceSynchronize();
145     checkCUDAError("Error at kernel complement");
146
```

```
147     cudaMemcpy(host_r, dev_r1, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
148     checkCUDAError("Error at memcpy host_r <- dev_r1");
149     cudaMemcpy(host_g, dev_g1, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
150     checkCUDAError("Error at memcpy host_r <- dev_g1");
151     cudaMemcpy(host_b, dev_b1, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
152     checkCUDAError("Error at memcpy host_r <- dev_b1");
153
154     for (int i = 0; i < R; i++) {
155         for (int j = 0; j < C; j++) {
156             imgComp.at<Vec3b>(i, j)[0] = host_b[i * C + j];
157             imgComp.at<Vec3b>(i, j)[1] = host_g[i * C + j];
158             imgComp.at<Vec3b>(i, j)[2] = host_r[i * C + j];
159         }
160     }
161
162     contrast << < grid, block >> > (dev_r2, dev_g2, dev_b2,
            0.5);
163     cudaDeviceSynchronize();
164     checkCUDAError("Error at kernel contrast");
165
166     cudaMemcpy(host_r, dev_r2, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
167     checkCUDAError("Error at memcpy host_r <- dev_r2");
168     cudaMemcpy(host_g, dev_g2, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
169     checkCUDAError("Error at memcpy host_r <- dev_g2");
170     cudaMemcpy(host_b, dev_b2, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
171     checkCUDAError("Error at memcpy host_r <- dev_b2");
172
173     for (int i = 0; i < R; i++) {
174         for (int j = 0; j < C; j++) {
175             imgCont.at<Vec3b>(i, j)[0] = host_b[i * C + j];
176             imgCont.at<Vec3b>(i, j)[1] = host_g[i * C + j];
177             imgCont.at<Vec3b>(i, j)[2] = host_r[i * C + j];
178         }
179     }
180
181     brightness << < grid, block >> > (dev_r3, dev_g3, dev_b3,
            100);
182     cudaDeviceSynchronize();
183     checkCUDAError("Error at kernel brightness");
184
185     cudaMemcpy(host_r, dev_r3, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
186     checkCUDAError("Error at memcpy host_r <- dev_r3");
187     cudaMemcpy(host_g, dev_g3, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
```

```
188     checkCUDAError("Error at memcpy host_r <- dev_g3");
189     cudaMemcpy(host_b, dev_b3, sizeof(uchar) * R * C,
            cudaMemcpyDeviceToHost);
190     checkCUDAError("Error at memcpy host_r <- dev_b3");
191
192     for (int i = 0; i < R; i++) {
193         for (int j = 0; j < C; j++) {
194             imgBright.at<Vec3b>(i, j)[0] = host_b[i * C + j];
195             imgBright.at<Vec3b>(i, j)[1] = host_g[i * C + j];
196             imgBright.at<Vec3b>(i, j)[2] = host_r[i * C + j];
197         }
198     }
199
200     imshow("Image", img);
201     imshow("Image Complement", imgComp);
202     imshow("Image Contrast", imgCont);
203     imshow("Image Brightness", imgBright);
204     waitKey(0);
205
206     free(host_r);
207     free(host_g);
208     free(host_b);
209     cudaFree(dev_r1);
210     cudaFree(dev_g1);
211     cudaFree(dev_b1);
212     cudaFree(dev_r2);
213     cudaFree(dev_g2);
214     cudaFree(dev_b2);
215     cudaFree(dev_r3);
216     cudaFree(dev_g3);
217     cudaFree(dev_b3);
218
219     return 0;
220 }
```

Output

Complement

Figure 3: img

Contrast

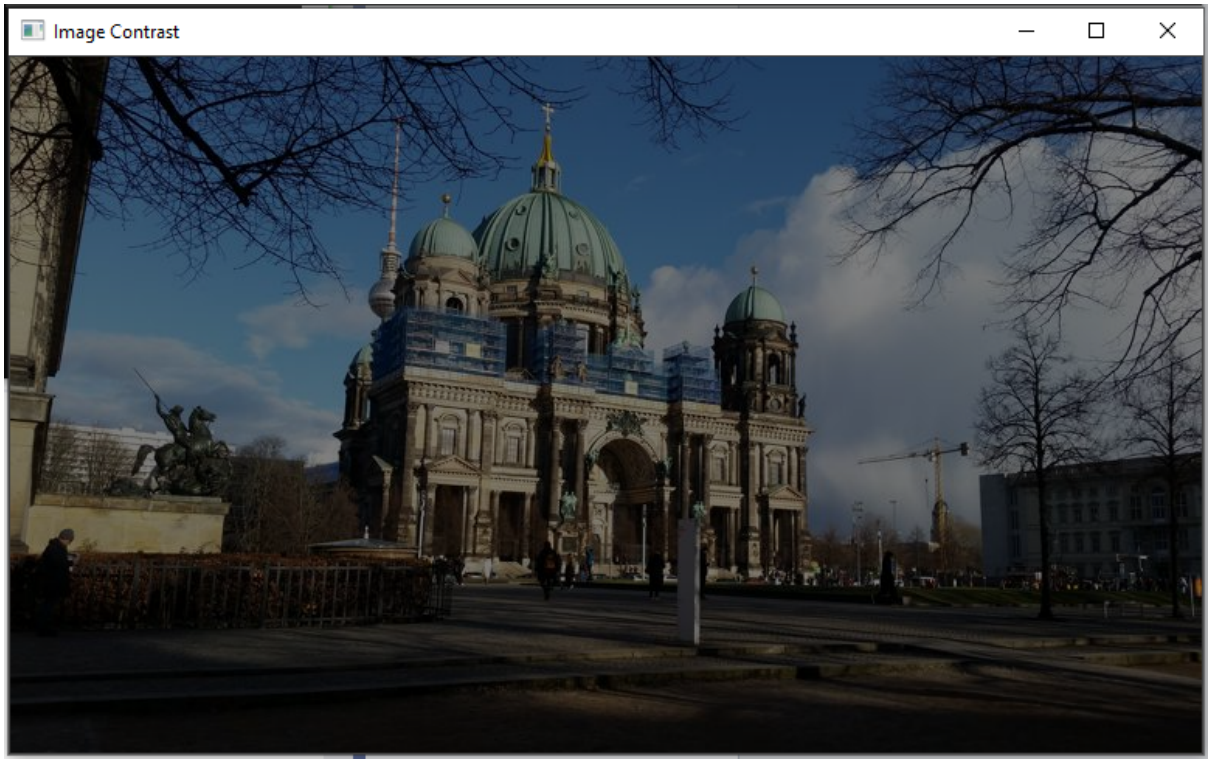


Figure 4: img

Brightness

Figure 5: img

RGB Image Manipulation: Other Options

Now, instead of processing the complement of the RGB image by reading from 3 vectors of information, let's try and read from a single vector that holds R, G and B vectors in one: vecSize x 3 this time.

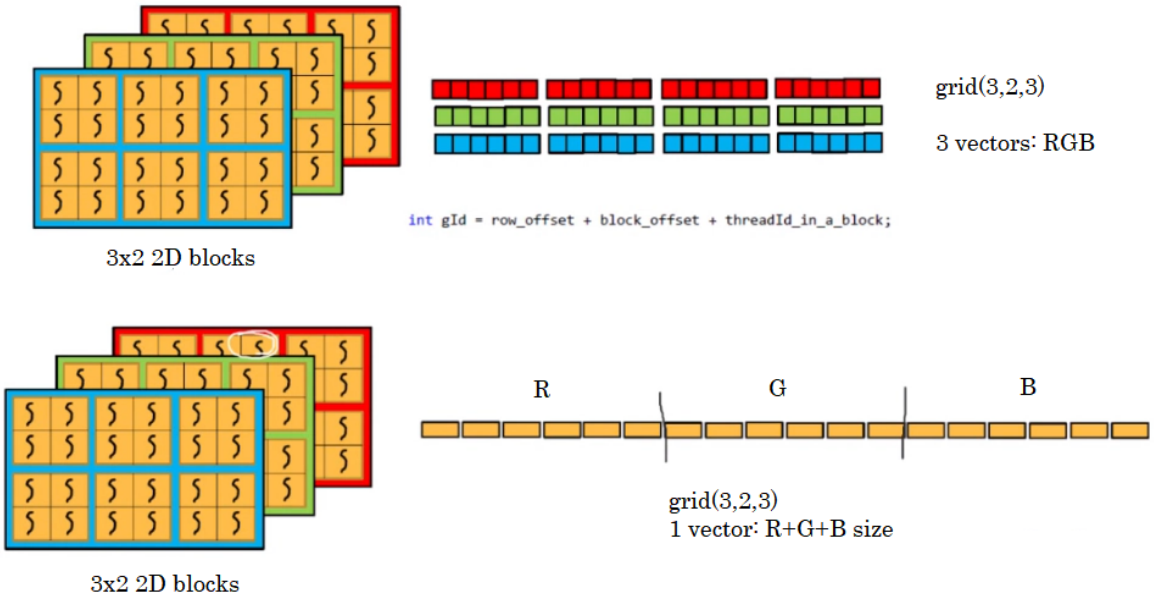


Figure 1: img

Solution

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <opencv2/opencv.hpp>
7
8  __host__ void checkCUDAError(const char* msg) {
9      cudaError_t error;
10     cudaDeviceSynchronize();
11     error = cudaGetLastError();
12     if (error != cudaSuccess) {
13         printf("ERROR %d: %s (%s)\n", error,
               cudaGetErrorString(error), msg);
14     }
15 }
16
17 __global__ void complement(uchar* RGB) {
18
```

```
19      // locate my current block row
20      int threads_per_block = blockDim.x * blockDim.y;
21      int threads_per_row = threads_per_block * gridDim.x;
22      int row_offset = threads_per_row * blockIdx.y;
23
24      // locate my current block column
25      int block_offset = blockIdx.x * threads_per_block;
26      int threadId_inside = blockDim.x * threadIdx.y + threadIdx
            .x;
27
28      // locate my current grid row
29      int thread_per_grid = (gridDim.x * gridDim.y *
            threads_per_block);
30      int gridOffset = blockIdx.z * thread_per_grid;
31
32      int gId = gridOffset + row_offset + block_offset +
            threadId_inside;
33      int C = gridDim.x * 32;
34      int R = gridDim.y * 32;
35      RGB[gId] = 255 - RGB[gId];
36  }
37
38  using namespace cv;
39  int main() {
40
41      Mat img = imread("antenaRGB.jpg");
42
43      const int R = img.rows;
44      const int C = img.cols;
45
46      Mat imgComp(img.rows, img.cols, img.type());
47      uchar* host_rgb,* dev_rgb;
48      host_rgb = (uchar*)malloc(sizeof(uchar) * R * C * 3);
49
50      cudaMalloc((void**)&dev_rgb, sizeof(uchar) * R * C * 3);
51      checkCUDAError("Error at malloc dev_r1");
52
53      // matrix as vector
54      for (int k = 0; k < 3; k++){
55          for (int i = 0; i < R; i++) {
56              for (int j = 0; j < C; j++) {
57                  Vec3b pix = img.at<Vec3b>(i, j);
58
59                  host_rgb[i * C + j + (k * R * C)] = pix[k];
60
61              }
62          }
63      }
64      cudaMemcpy(dev_rgb, host_rgb, sizeof(uchar) * R * C * 3,
            cudaMemcpyHostToDevice);
65      checkCUDAError("Error at memcpy host_rgb -> dev_rgb");
```

```
66
67      dim3 block(32, 32);
68      dim3 grid(C / 32, R / 32, 3);
69
70      complement << < grid, block >> > (dev_rgb);
71      cudaDeviceSynchronize();
72      checkCUDAError("Error at kernel complement");
73
74      cudaMemcpy(host_rgb, dev_rgb, sizeof(uchar) * R * C * 3,
           cudaMemcpyDeviceToHost);
75      checkCUDAError("Error at memcpy host_rgb <- dev_rgb");
76
77      for (int k = 0; k < 3; k++) {
78          for (int i = 0; i < R; i++) {
79              for (int j = 0; j < C; j++) {
80                  imgComp.at<Vec3b>(i, j)[k] = host_rgb[i * C +
                       j + (k * R * C)];
81              }
82          }
83      }
84
85
86      imshow("Image", img);
87      imshow("Image Complement", imgComp);
88      waitKey(0);
89
90      free(host_rgb);
91      cudaFree(dev_rgb);
92
93      return 0;
94  }
```

Kernel Validation

Verify just once that the kernel operations are correctly performed, which is crutial when working with large amounts of data. This consists in a comparison between the result of the same operation done both in CPU and GPU.

We would need to add a function that does that same pixel operation in the host, to continue the same examples with image processing, for example. This is basically making the kernel function but in the host.

Validation: execute the same kernel operations in the host and compare the results given by the CPU and GPU.

Exercise

Execute a kernel validation for the last kernel where we made the complement of an RGB image. The kernel validation includes two functions: one for performing the complement in the CPU and one for comparing the CPU image and the GPU image.

Solution

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <opencv2/opencv.hpp>
7
8  using namespace cv;
9
10 __host__ void checkCUDAError(const char* msg) {
11     cudaError_t error;
12     cudaDeviceSynchronize();
13     error = cudaGetLastError();
14     if (error != cudaSuccess) {
15         printf("ERROR %d: %s (%s)\n", error,
               cudaGetErrorString(error), msg);
16     }
17 }
18
19 __global__ void complement(uchar* RGB) {
20
21     // locate my current block row
22     int threads_per_block = blockDim.x * blockDim.y;
23     int threads_per_row = threads_per_block * gridDim.x;
24     int row_offset = threads_per_row * blockIdx.y;
```

```
25
26      // locate my current block column
27      int block_offset = blockIdx.x * threads_per_block;
28      int threadId_inside = blockDim.x * threadIdx.y + threadIdx
            .x;
29
30      // locate my current grid row
31      int thread_per_grid = (gridDim.x * gridDim.y *
            threads_per_block);
32      int gridOffset = blockIdx.z * thread_per_grid;
33
34      int gId = gridOffset + row_offset + block_offset +
            threadId_inside;
35      RGB[gId] = 255 - RGB[gId];
36  }
37
38  __host__ void complementCPU(Mat* original, Mat* comp) {
39      for (int i = 0; i < original->rows; i++) {
40          for (int j = 0; j < original->cols; j++) {
41              comp->at<Vec3b>(i, j)[0] = 255 - original->at<
                    Vec3b>(i, j)[0];
42              comp->at<Vec3b>(i, j)[1] = 255 - original->at<
                    Vec3b>(i, j)[1];
43              comp->at<Vec3b>(i, j)[2] = 255 - original->at<
                    Vec3b>(i, j)[2];
44          }
45      }
46  }
47
48  __host__ bool validationKernel(Mat img1, Mat img2) {
49      Vec3b* pImg1, * pImg2;
50      for (int k = 0; k < 3; k++) {
51          for (int i = 0; i < img1.rows; i++) {
52              pImg1 = img1.ptr<Vec3b>(i);
53              pImg2 = img2.ptr<Vec3b>(i);
54              for (int j = 0; j < img1.cols; j++) {
55                  if (pImg1[j][k] != pImg2[j][k]) {
56                      printf("Error at kernel validation\n");
57                      return true;
58                  }
59              }
60          }
61      }
62      printf("Kernel validation successful\n");
63      return false;
64  }
65
66  int main() {
67
68      Mat img = imread("antenaRGB.jpg");
69
```

```
70      const int R = img.rows;
71      const int C = img.cols;
72
73      Mat imgComp(img.rows, img.cols, img.type());
74      Mat imgCompCPU(img.rows, img.cols, img.type());
75      uchar* host_rgb, * dev_rgb;
76      host_rgb = (uchar*)malloc(sizeof(uchar) * R * C * 3);
77
78      cudaMalloc((void**)&dev_rgb, sizeof(uchar) * R * C * 3);
79      checkCUDAError("Error at malloc dev_r1");
80
81      // matrix as vector
82      for (int k = 0; k < 3; k++) {
83          for (int i = 0; i < R; i++) {
84              for (int j = 0; j < C; j++) {
85                  Vec3b pix = img.at<Vec3b>(i, j);
86
87                  host_rgb[i * C + j + (k * R * C)] = pix[k];
88
89              }
90          }
91      }
92      cudaMemcpy(dev_rgb, host_rgb, sizeof(uchar) * R * C * 3,
           cudaMemcpyHostToDevice);
93      checkCUDAError("Error at memcpy host_rgb -> dev_rgb");
94
95      dim3 block(32, 32);
96      dim3 grid(C / 32, R / 32, 3);
97
98      complement << < grid, block >> > (dev_rgb);
99      cudaDeviceSynchronize();
100     checkCUDAError("Error at kernel complement");
101
102     cudaMemcpy(host_rgb, dev_rgb, sizeof(uchar) * R * C * 3,
           cudaMemcpyDeviceToHost);
103     checkCUDAError("Error at memcpy host_rgb <- dev_rgb");
104
105     for (int k = 0; k < 3; k++) {
106         for (int i = 0; i < R; i++) {
107             for (int j = 0; j < C; j++) {
108                 imgComp.at<Vec3b>(i, j)[k] = host_rgb[i * C +
                       j + (k * R * C)];
109             }
110         }
111     }
112
113     complementCPU(&img, &imgCompCPU);
114     bool error = validationKernel(imgCompCPU, imgComp);
115
116     if (error) {
117         printf("Check kernel operations\n");
```

```
118             return 0;
119         }
120 
121 
122         imshow("Image", img);
123         imshow("Image Complement CPU", imgCompCPU);
124         imshow("Image Complement GPU", imgComp);
125         waitKey(0);
126 
127         free(host_rgb);
128         cudaFree(dev_rgb);
129 
130         return 0;
131 }
```

Practice

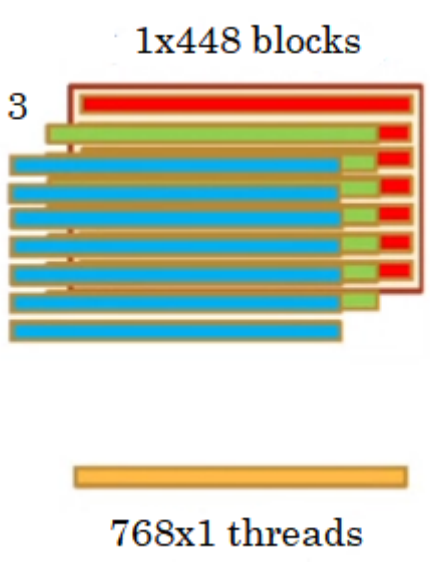


Figure 1: img

Lab 13

Write a program in c/c++ using CUDA in which you implement a kernel to calculate an RGB image complement. The kernel must be verified, consider the requirements:

- The complement of an image is defined as:

$I(x,y) = 255 - I(x,y)$

- Blocks of 768 x 1 threads.
- A grid of 1 x 448 x 3 blocks.
- The kernel signature should be: `__global__ void complement(uchar* RGB)`
- The CPU complement function signature should be: `__host__ void complementCPU (Mat* original, Mat* comp)`
- The kernel validation signature should be: `__host__ bool validationKernel(Mat img1, Mat img2)`

Solution

```
1 #include "cuda_runtime.h"
2 #include "device_launch_parameters.h"
3
4 #include <stdio.h>
```

```
5  #include <stdlib.h>
6  #include <opencv2/opencv.hpp>
7
8  using namespace cv;
9
10 __host__ void checkCUDAError(const char* msg) {
11     cudaError_t error;
12     cudaDeviceSynchronize();
13     error = cudaGetLastError();
14     if (error != cudaSuccess) {
15         printf("ERROR %d: %s (%s)\n", error,
               cudaGetErrorString(error), msg);
16     }
17 }
18
19 __global__ void complement(uchar* RGB) {
20
21     // locate my current block row
22     int threads_per_block = blockDim.x;
23     int threads_per_row = threads_per_block * gridDim.x;
24     int row_offset = threads_per_row * blockIdx.y;
25
26     // locate my current block column
27     int block_offset = blockIdx.x * threads_per_block;
28
29     // locate my current grid row
30     int thread_per_grid = (gridDim.x * gridDim.y *
          threads_per_block);
31     int gridOffset = blockIdx.z * thread_per_grid;
32
33     int gId = gridOffset + row_offset + block_offset +
          threadIdx.x;
34     RGB[gId] = 255 - RGB[gId];
35 }
36
37 __host__ void complementCPU(Mat* original, Mat* comp) {
38     for (int i = 0; i < original->rows; i++) {
39         for (int j = 0; j < original->cols; j++) {
40             comp->at<Vec3b>(i, j)[0] = 255 - original->at<
                  Vec3b>(i, j)[0];
41             comp->at<Vec3b>(i, j)[1] = 255 - original->at<
                  Vec3b>(i, j)[1];
42             comp->at<Vec3b>(i, j)[2] = 255 - original->at<
                  Vec3b>(i, j)[2];
43         }
44     }
45 }
46
47 __host__ bool validationKernel(Mat img1, Mat img2) {
48     Vec3b* pImg1, * pImg2;
49     for (int k = 0; k < 3; k++) {
```

```
50          for (int i = 0; i < img1.rows; i++) {
51              pImg1 = img1.ptr<Vec3b>(i);
52              pImg2 = img2.ptr<Vec3b>(i);
53              for (int j = 0; j < img1.cols; j++) {
54                  if (pImg1[j][k] != pImg2[j][k]) {
55                      printf("Error at kernel validation\n");
56                      return true;
57                  }
58              }
59          }
60      }
61      printf("Kernel validation successful\n");
62      return false;
63  }
64
65  int main() {
66
67      Mat img = imread("antenaRGB.jpg");
68
69      const int R = img.rows;
70      const int C = img.cols;
71
72      Mat imgComp(img.rows, img.cols, img.type());
73      Mat imgCompCPU(img.rows, img.cols, img.type());
74      uchar* host_rgb, * dev_rgb;
75      host_rgb = (uchar*)malloc(sizeof(uchar) * R * C * 3);
76
77      cudaMalloc((void**)&dev_rgb, sizeof(uchar) * R * C * 3);
78      checkCUDAError("Error at malloc dev_r1");
79
80      // matrix as vector
81      for (int k = 0; k < 3; k++) {
82          for (int i = 0; i < R; i++) {
83              for (int j = 0; j < C; j++) {
84                  Vec3b pix = img.at<Vec3b>(i, j);
85
86                  host_rgb[i * C + j + (k * R * C)] = pix[k];
87
88              }
89          }
90      }
91      cudaMemcpy(dev_rgb, host_rgb, sizeof(uchar) * R * C * 3,
           cudaMemcpyHostToDevice);
92      checkCUDAError("Error at memcpy host_rgb -> dev_rgb");
93
94      //dim3 block(32, 32);
95      //dim3 grid(C / 32, R / 32, 3); // 24 14
96      dim3 block(C, 1, 1); // 768
97      dim3 grid(1, R, 3); // 448
98
99      complement << < grid, block >> > (dev_rgb);
```

```
100     cudaDeviceSynchronize();
101     checkCUDAError("Error at kernel complement");
102
103     cudaMemcpy(host_rgb, dev_rgb, sizeof(uchar) * R * C * 3,
            cudaMemcpyDeviceToHost);
104     checkCUDAError("Error at memcpy host_rgb <- dev_rgb");
105
106     for (int k = 0; k < 3; k++) {
107         for (int i = 0; i < R; i++) {
108             for (int j = 0; j < C; j++) {
109                 imgComp.at<Vec3b>(i, j)[k] = host_rgb[i * C +
                        j + (k * R * C)];
110             }
111         }
112     }
113
114     complementCPU(&img, &imgCompCPU);
115     bool error = validationKernel(imgCompCPU, imgComp);
116
117     if (error) {
118         printf("Check kernel operations\n");
119         return 0;
120     }
121
122
123     imshow("Image", img);
124     imshow("Image Complement CPU", imgCompCPU);
125     imshow("Image Complement GPU", imgComp);
126     waitKey(0);
127
128     free(host_rgb);
129     cudaFree(dev_rgb);
130
131     return 0;
132 }
```

Input

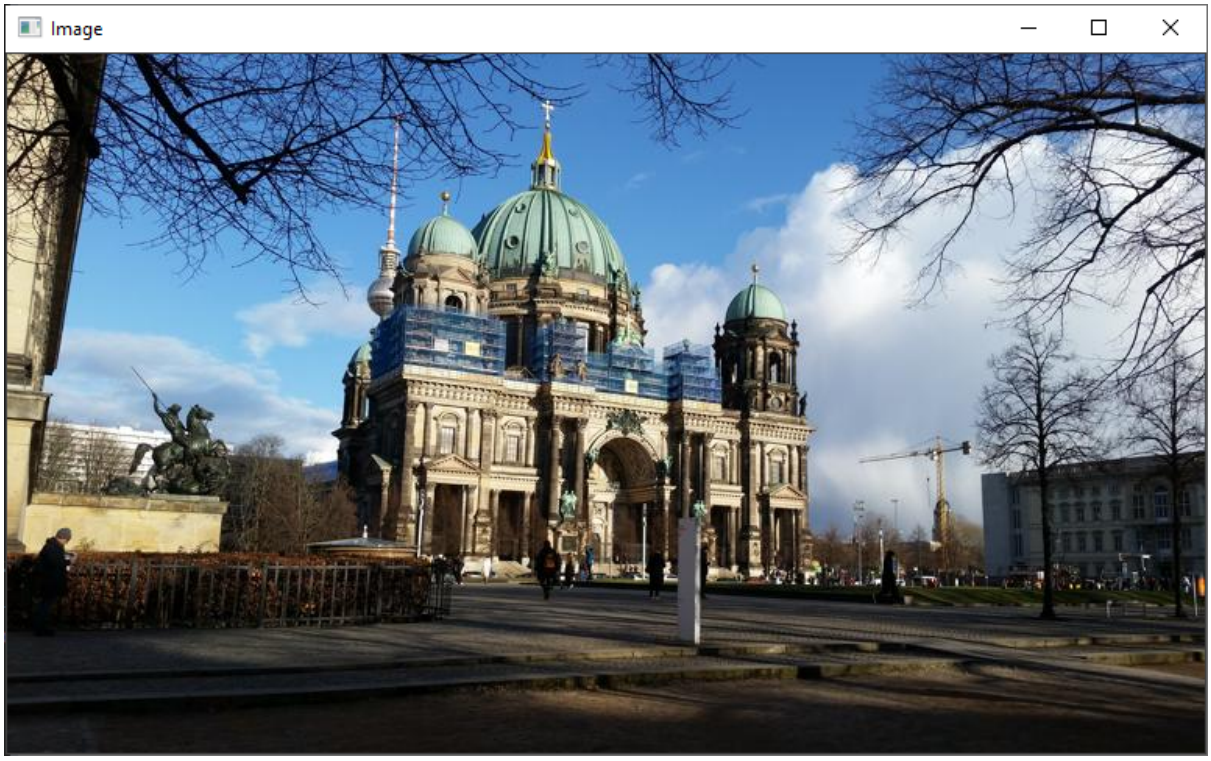


Figure 2: img

Output

Figure 3: img

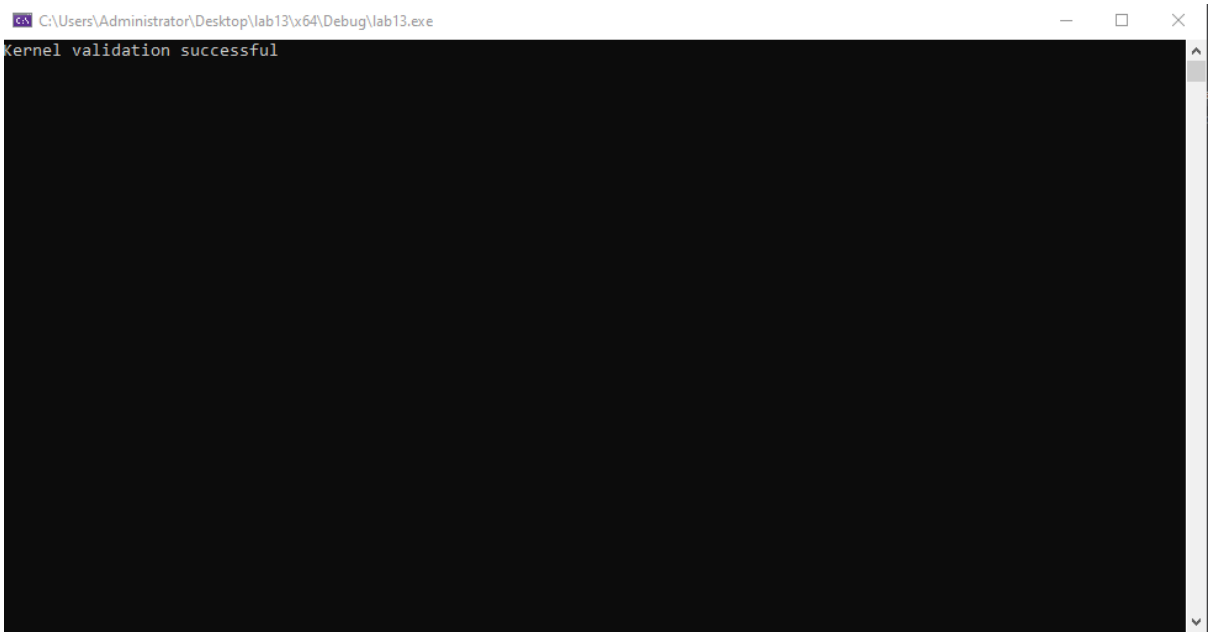

C:\Users\Administrator\Desktop\lab13\x64\Debug\lab13.exe
Kernel validation successful

Figure 4: img

Configs Cheatsheet

1D Grid & 1D Block

Figure 1: img

- Block along x axis

```
1 int gId = threadIdx.x;
```

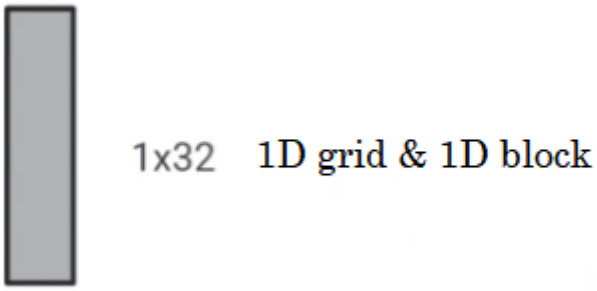


Figure 2: img

- Block along y axis

```
1 int gId = threadIdx.y;
```

1D Grid & N 1D Blocks

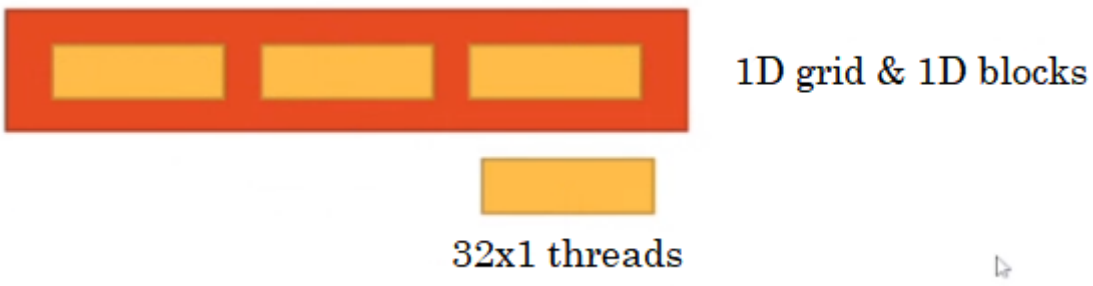


Figure 3: img

- N 1D blocks along x axis

```
int threadsPerBlock = blockDim.x;
int blockOffset = threadsPerBlock * blockIdx.x;
int idInsideBlock = threadIdx.x;
int gId = blockOffset + idInsideBlock;
```

1D Grid & N 2D Blocks

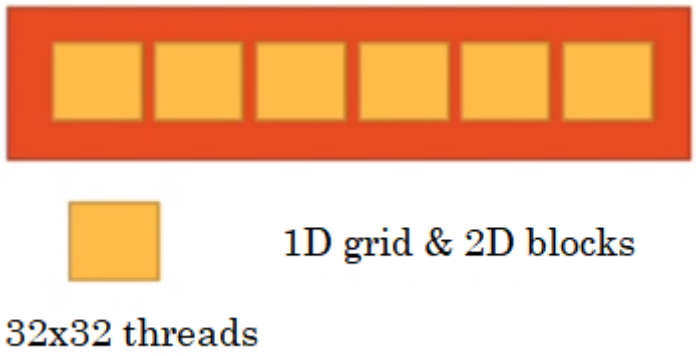


Figure 4: img

- N 2D blocks along x axis

```
int threadsPerBlock = blockDim.x * blockDim.y;
int blockOffset = threadsPerBlock * blockIdx.x;
int idInsideBlock = blockDim.x * threadIdx.y + threadIdx.x;
int gId = blockOffset + idInsideBlock;
```

1D Grid & N 1D Blocks

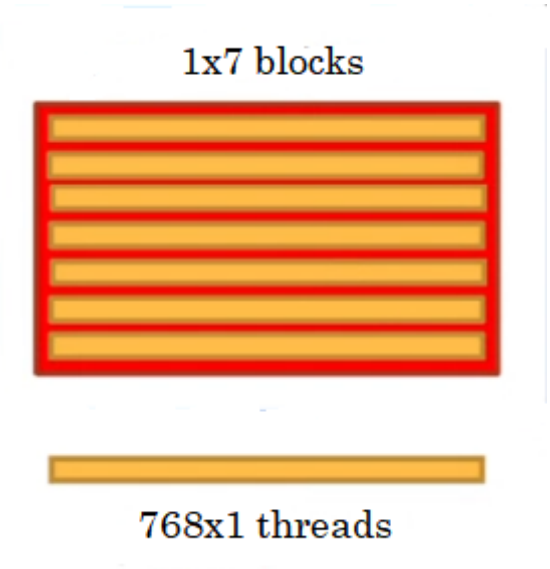


Figure 5: img

- N 1D blocks along y axis

```
int threadsPerBlock = blockDim.x;
int rowOffset = threadsPerBlock * blockIdx.y;
int idInsideBlock = threadIdx.x;
int gId = rowOffset + idInsideBlock;
```

1D Grid & N 1D Blocks

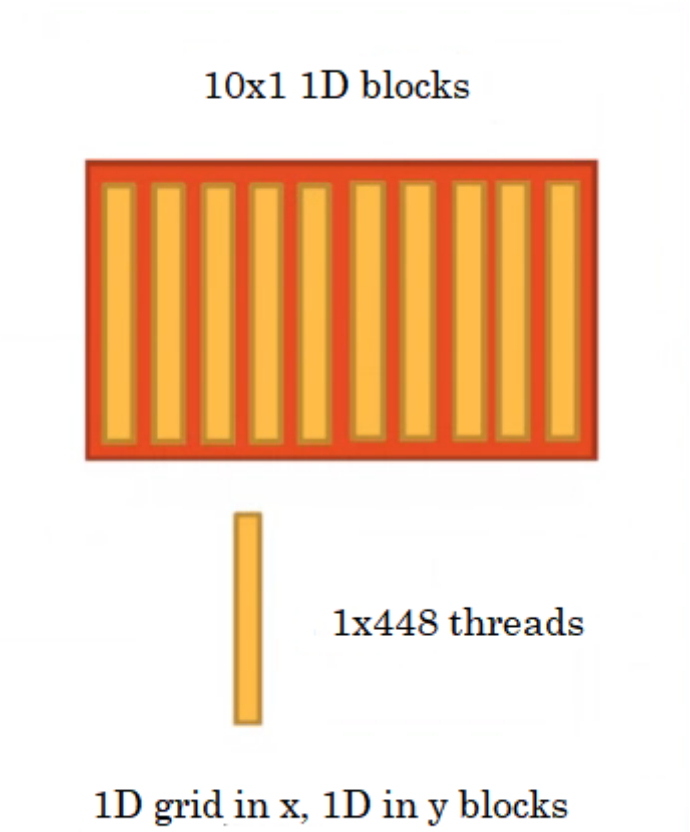


Figure 6: img

- 1D grid along x and 1D blocks along its y

```
int threadsPerBlock = blockDim.y;
int blockOffset = threadsPerBlock * blockIdx.x;
int idInsideBlock = threadIdx.y;
int gId = blockOffset + idInsideBlock;
```

2D Grid & 2D Blocks

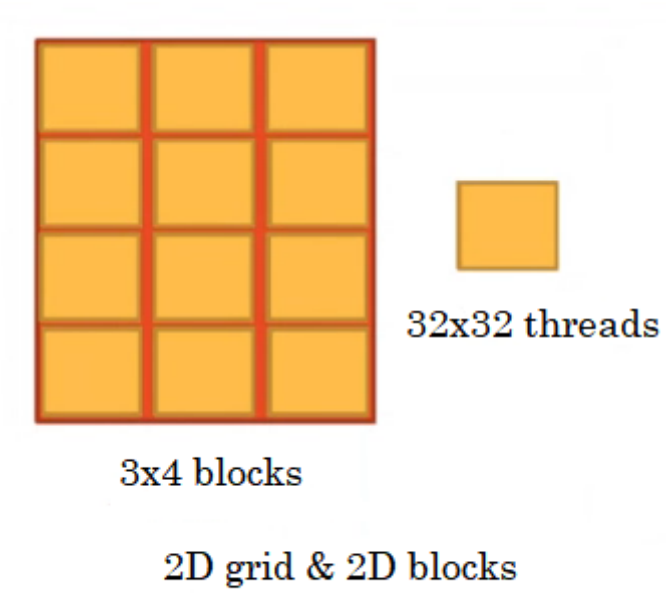


Figure 7: img

- 2D grid (x and y) and 2D blocks (x and y)

```
int threadsPerBlock = blockDim.x * blockDim.y;
int threadsPerRow = threadsPerBlock * gridDim.x;
int rowOffset = threadsPerRow * blockIdx.y;
int blockOffset = threadsPerBlock * blockIdx.x;
int idInsideBlock = blockDim.x * threadIdx.y + threadIdx.x;
int gId = rowOffset + blockOffset + idInsideBlock;
```

3D Grid & 2D Blocks

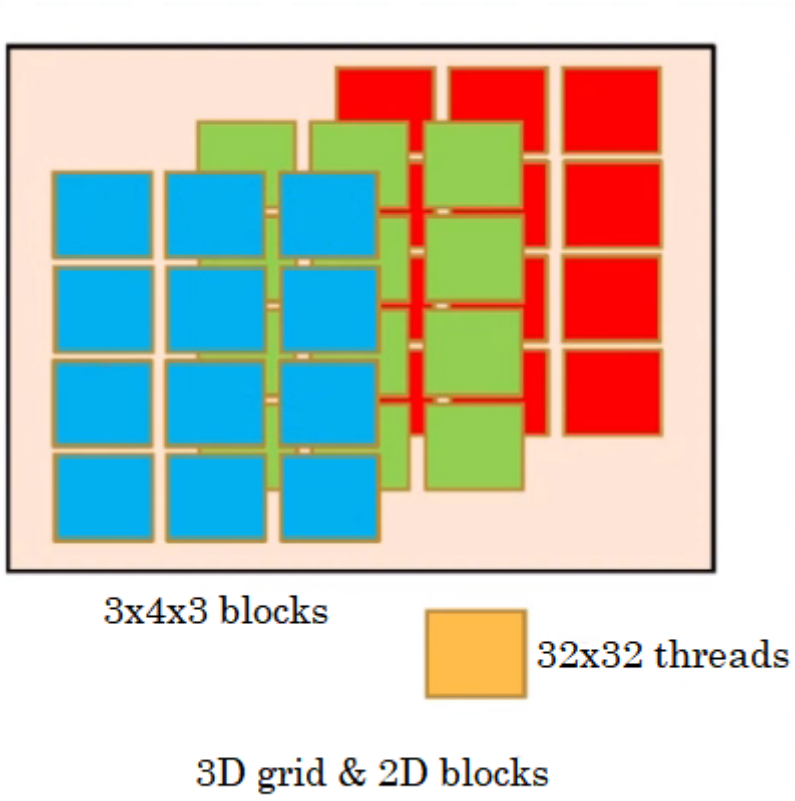


Figure 8: img

- dim3 grid(3, 4, 3) and dim3 block(32, 32, 1)

```
int threadsPerBlock = blockDim.x * blockDim.y;
int threadsPerRow = threadsPerBlock * gridDim.x;
int rowOffset = threadsPerRow * blockIdx.y;
int blockOffset = threadsPerBlock * blockIdx.x;
int idInsideBlock = blockDim.x * threadIdx.y + threadIdx.x;
int threadsPerGrid = threadsPerBlock * gridDim.x * gridDim.y;
int gridOffset = threadsPerGrid * blockIdx.z;
int gId = gridOffset + rowOffset + blockOffset + idInsideBlock
    ;
```

Exercises

What would be the gId formula for the configs below?

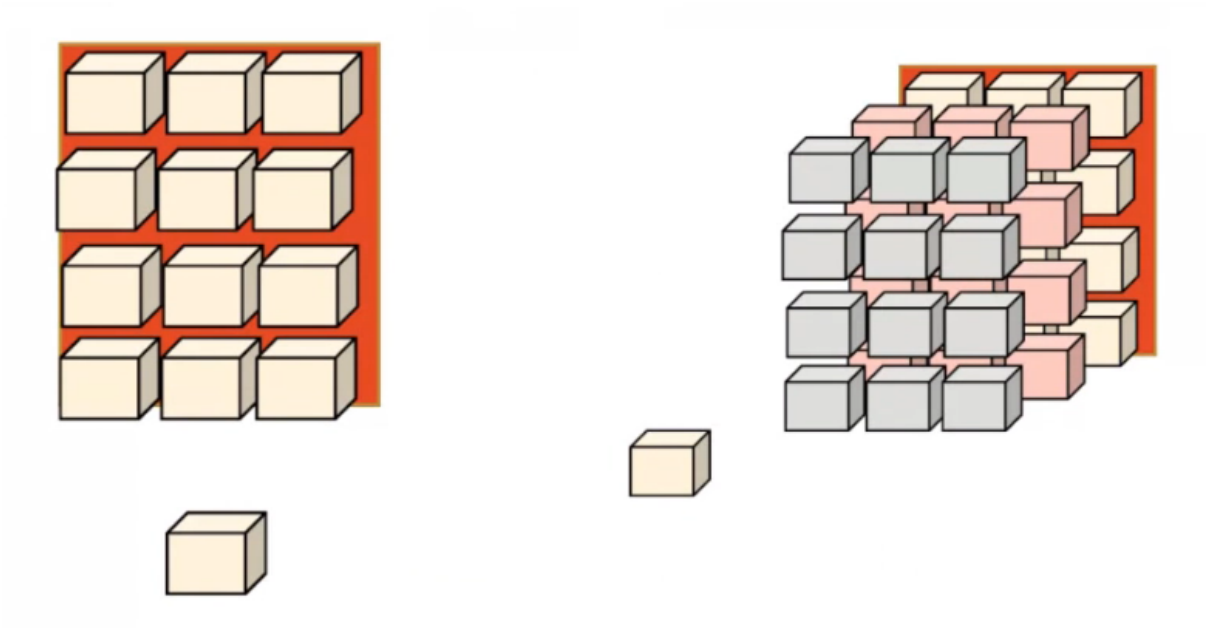


Figure 9: img

Processing Time in the GPU and CPU

If we were to test the processing time of the same operation over and over, say z = 2x + y, performed both by the CPU and GPU, there would be a noticeable difference if the size of the vectors to store this repeated operations is enormous (102,400 cells). Consider the image below:

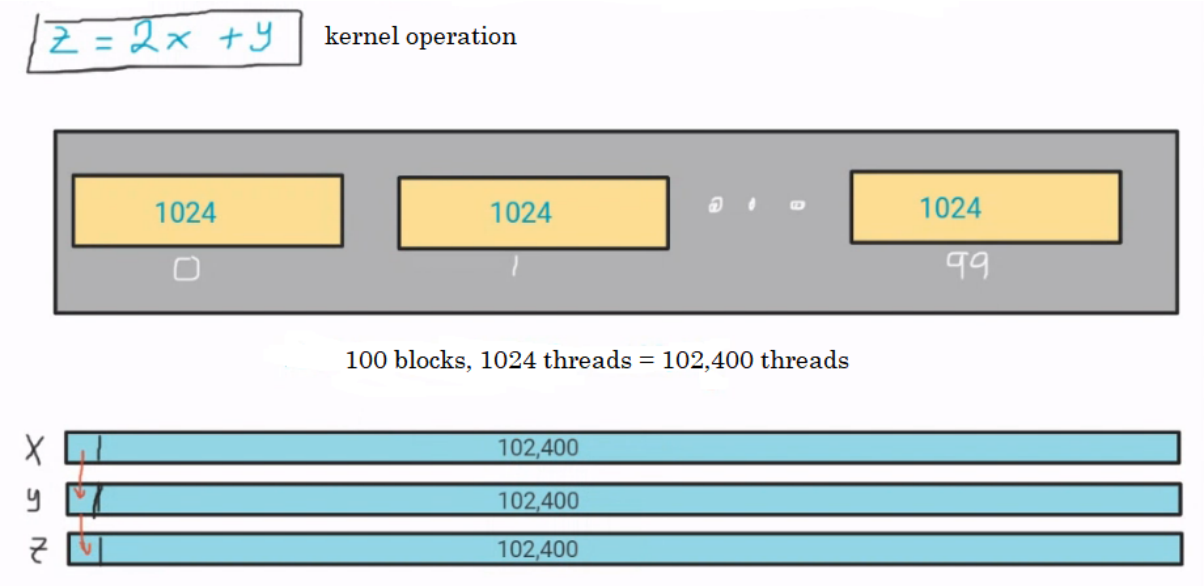


Figure 1: img

Implementation

```cpp
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <iostream>
#include <time.h>

using namespace std;

__global__ void GPU_fn(int* x, int* y, int* z) {
    int gId = blockIdx.x * blockDim.x + threadIdx.x;
    z[gId] = 2 * x[gId] + y[gId];
}


__host__ void CPU_fn(int* x, int* y, int* z, int vecSize) {
    for (int i = 0; i < vecSize; i++) {
        z[i] = 2 * x[i] + y[i];
    }
}
```

```
21
22 __host__ void checkCUDAError(const char* msg) {
23     cudaError_t error;
24     cudaDeviceSynchronize();
25     error = cudaGetLastError();
26     if (error != cudaSuccess) {
27         printf("ERROR %d: %s (%s)\n", error,
              cudaGetErrorString(error), msg);
28     }
29 }
30
31 __host__ void validate(int* result_CPU, int* result_GPU, int N
      ) {
32     for (int i = 0; i < N; i++) {
33         if (result_CPU[i] != result_GPU[i]) {
34             printf("The vectors are not equal\n");
35             return;
36         }
37     }
38     printf("Kernel validated successfully\n");
39     return;
40 }
41
42 int main()
43 {
44     cudaDeviceProp prop;
45     cudaGetDeviceProperties(&prop, 0);
46     printf("maxThreadsPerBlock: %d\n", prop.maxThreadsPerBlock
          ); // 1024 in all its block dimension
47     printf("maxThreadsDim[0]: %d\n", prop.maxThreadsDim[0]);
          // 1024 in a block's x dim
48     // 100 blocks x 1024 threads = 102 400 threads
49     // x,y,z vectors of size 1024
50
51     int numBlocks = 100000; // add one zero and you get ERROR
          2: run out of global memory
52     int numThreadsPerBlock = 1024;
53     int vecSize = numBlocks * numThreadsPerBlock;
54
55
56     int* hostx = (int*)malloc(vecSize * sizeof(int));
57     int* hosty = (int*)malloc(vecSize * sizeof(int));
58     int* hostzCPU = (int*)malloc(vecSize * sizeof(int));
59     int* hostzGPU = (int*)malloc(vecSize * sizeof(int));
60
61     int* devx, * devy, * devz;
62     cudaMalloc((void**)&devx, vecSize * sizeof(int));
63     checkCUDAError("Error at cudaMalloc: devx");
64     cudaMalloc((void**)&devy, vecSize * sizeof(int));
65     checkCUDAError("Error at cudaMalloc: devy");
66     cudaMalloc((void**)&devz, vecSize * sizeof(int));
```

```
67      checkCUDAError("Error at cudaMalloc: devz");
68
69      for (int i = 0; i < vecSize; i++) {
70          hostx[i] = 1;
71          hosty[i] = 2;
72      }
73
74      cudaMemcpy(devx, hostx, vecSize * sizeof(int),
           cudaMemcpyHostToDevice);
75      cudaMemcpy(devy, hosty, vecSize * sizeof(int),
           cudaMemcpyHostToDevice);
76      dim3 block(numThreadsPerBlock);
77      dim3 grid(numBlocks);
78
79      cudaEvent_t startGPU;
80      cudaEvent_t endGPU;
81      cudaEventCreate(&startGPU);
82      cudaEventCreate(&endGPU); // be able to mark the time
83      cudaEventRecord(startGPU); // save current time
84      GPU_fn << <grid, block >> > (devx, devy, devz);
85      cudaEventRecord(endGPU);
86      cudaEventSynchronize(endGPU); // so that cudaEventRecord(
           startGPU) and cudaEventSynchronize(endGPU) are not done
            at the same time
87      float elapsedTimeGPU;
88      cudaEventElapsedTime(&elapsedTimeGPU, startGPU, endGPU);
89      cudaMemcpy(hostzGPU, devz, vecSize * sizeof(int),
           cudaMemcpyDeviceToHost);
90
91      clock_t startCPU = clock(); // save current time
92      CPU_fn(hostx, hosty, hostzCPU, vecSize);
93      clock_t endCPU = clock();
94      float elapsedTimeCPU = endCPU - startCPU;
95      printf("Time elapsed CPU: %f miliseconds\n",
           elapsedTimeCPU);
96      printf("Time elapsed GPU: %f miliseconds\n",
           elapsedTimeGPU);
97
98      validate(hostzCPU, hostzGPU, vecSize);
99
100     free(hostx);
101     free(hosty);
102     free(hostzCPU);
103     free(hostzGPU);
104     cudaFree(devx);
105     cudaFree(devy);
106     cudaFree(devz);
107
108     cudaEventDestroy(startGPU);
109     cudaEventDestroy(endGPU);
110
```

```
111         return 0;
112     }
```

Output

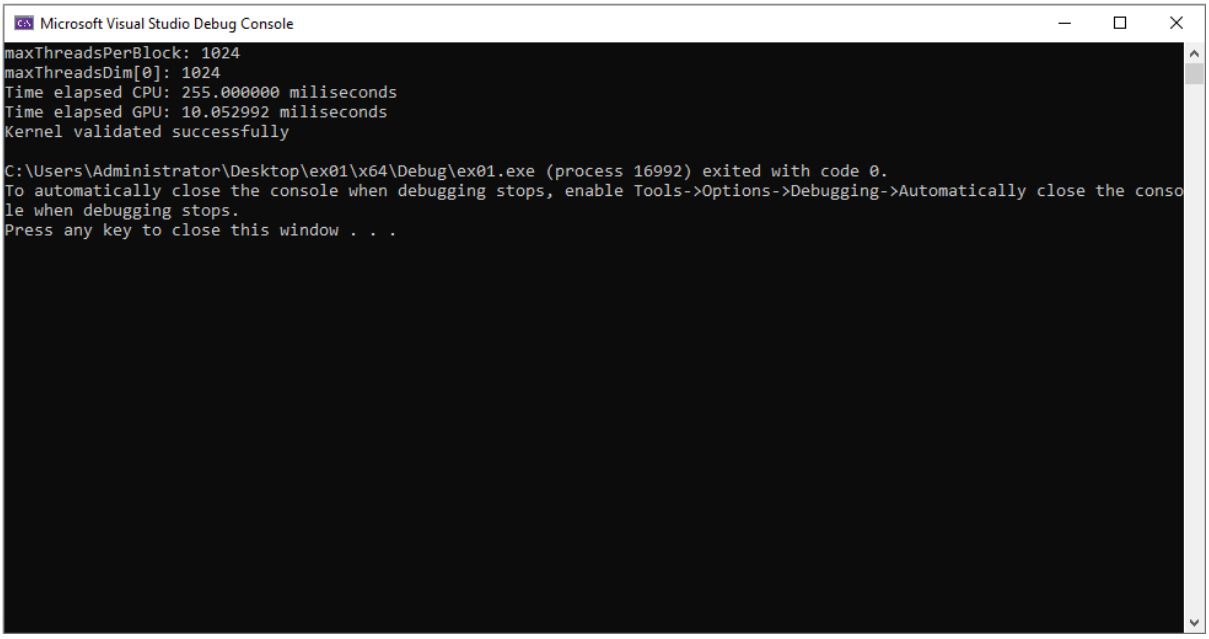


Figure 2: img

Parallel Reduction

It is a semi-parallel technique that consists in dividing the vector size into 2 and also divide the amount of threads into two: if you have a size 8 vector, you then use 4 threads to sum the index 0 and 4, 1 and 5, and so on. Then when the half vector is covered, the 4 first elements will have the sum of the first and last 4 elements. Then, you repeat the process for the 4 cells remaining: now 2 have two elements of the first and last 2 elements of the remaining vector, etc.

At the end, you will have one cell with the sum of all the vector elements. This cell will be the first cell of the original array, to then return this value in the kernel. Basically have to divide in two and add up the firsts and lasts, the divide that in two again, until you sum up all the elements.

It is not completely in parallel, but it is done in less iterations than a **sequential programming technique such as a for loop**.

- The only condition is that **the vector size is even**.

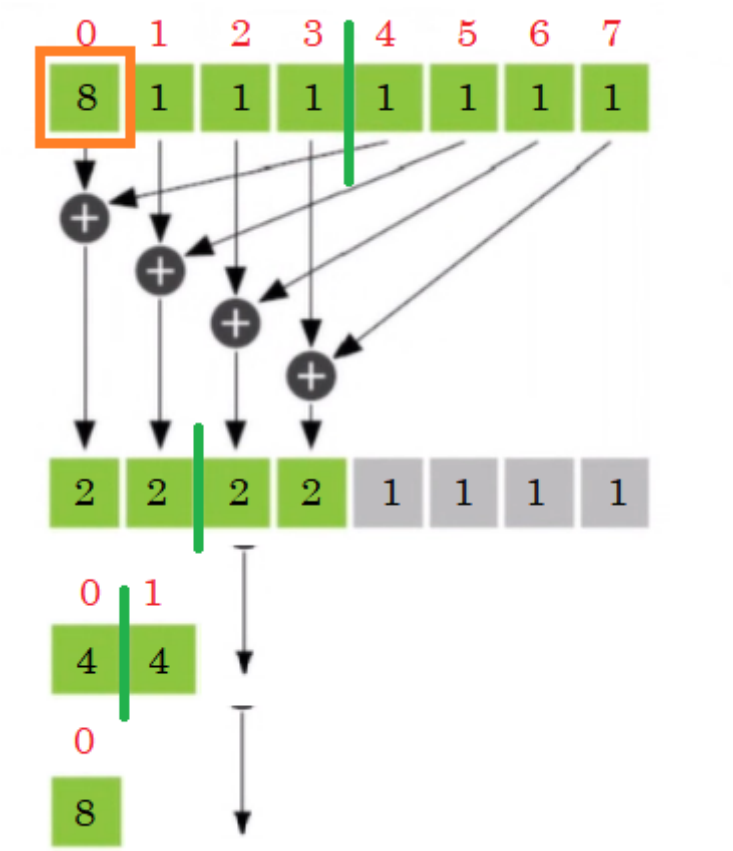


Figure 1: img

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

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

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__host__ void validate(int* result_CPU, int* result_GPU, int
   size) {
    if (*result_CPU != *result_GPU) {
        printf("The results are not equal.\n");
        return;
    }
    printf("Kernel validated successfully.\n");
    return;
}

__host__ void CPU_fn(int *v, int* sum, const int size) {
    for (int i = 0; i < size; i++) {
        *sum += v[i];
    }
}

__global__ void kernel(int* v, int* sum) {
    int gId = threadIdx.x;
    int step = blockDim.x;

    while (step) {
        if (gId < step) {
            v[gId] = v[gId] + v[gId + step];
        }
        step = step / 2;
    }
    if (gId == 0) {
        *sum = v[gId];
    }
}

int main() {

    const int size = 1024;
```

```
49      int* v = (int*)malloc(sizeof(int) * size);
50      int sumCPU = 0;
51      int sumGPU = 0;
52
53      int* dev_v, *sum;
54      cudaMalloc((void**)&dev_v, sizeof(int) * size);
55      cudaMalloc((void**)&sum, sizeof(int));
56
57      for (int i = 0; i < size; i++) {
58          v[i] = 1;
59      }
60
61      cudaMemcpy(dev_v, v, sizeof(int) * size,
           cudaMemcpyHostToDevice);
62      cudaMemcpy(sum, &sumGPU, sizeof(int),
           cudaMemcpyHostToDevice);
63
64      dim3 grid(1);
65      dim3 block(size);
66
67      kernel <<< grid, block >>> (dev_v, sum);
68      cudaMemcpy(&sumGPU, sum, sizeof(int),
           cudaMemcpyDeviceToHost);
69      printf("GPU sum: %d\n", sumGPU);
70
71      CPU_fn(v, &sumCPU, size);
72      printf("CPU sum: %d\n", sumCPU);
73
74      validate(&sumCPU, &sumGPU, size);
75
76      return 0;
77  }
```

Output

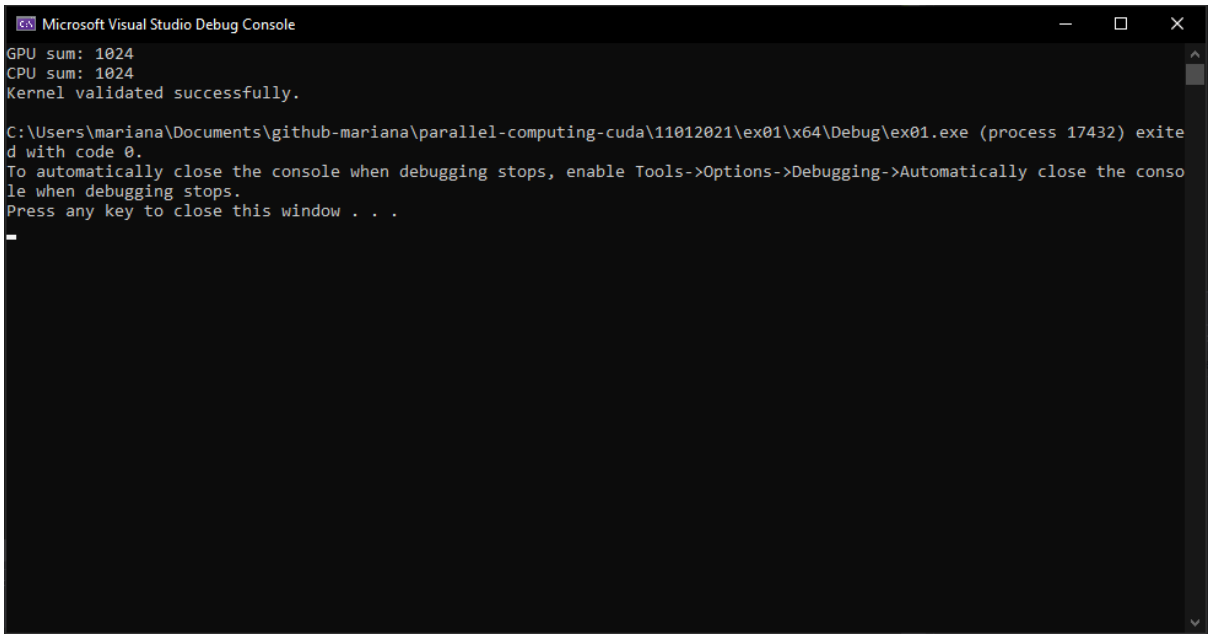


Figure 2: img

Warp Divergence

Warp Divergence happens when you launch the threads in a warp to do **different things** among them. For example, if you, inside a kernel, differentiate the activities of the threads using their global ID and send, for example, the even threads to do this and the odds to do something else. Last time, in parallel reduction, you provoked this divergence.

The problem with this divergence is that it theoretically reduces **velocity by half** than the one your kernel would have if threads in a warp didn't diverge, **per if condition**: if you have one if statement, you reduce the speed by half; two if statements, you reduce that half speed by half again, and so on.

What can we do to avoid this? Instead of using the **gId**, use the **warpId** to divide the tasks among the threads. This is not warp divergence, because **all threads of each warp are doing the same thing**.

- Example

This has divergence:

```
if (gId % 2 == 0){
    // activity 1
} else {
    // activity 2
}
```

This has no divergence:

```
int warpId = gId / 32;
if (warpId % 2 == 0){
    // activity 1
} else {
    // activity 2
}
```

Especifically, in parallel reduction we cannot use this to avoid divergence. When you cannot avoid divergence, another alternative is to fix this divergence by using the space of memory called **Shared Memory**. Up until now, we have just used **global memory** when we have variables inside the kernel. The memory we used with `cudaMalloc()` was the memory called **Global Memory**.

Shared Memory is independent memory per block, where all threads in a block can access the memory of *only* the block they belong to. It is a faster memory in terms of access (read/write) when compared to global memory, because it is closer to the processor: therefore it is recommended when you have data that is constantly used or referenced

inside a kernel. Parallel Redutcion is apt to this, because the elements of a vector are constantly referenced. The advise would be to store the vector in **Shared Memory**.

Nevertheless, we have less Shared Memory than the memory we have as Global Memory. You can check the capacity of these memories using the `cudaGetDeviceProperties()`, and it is given by block. Take care that your program does not surpass the capacity of Shared Memory when you write a kernel that requires it.

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

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    size_t sharedMemory = prop.sharedMemPerBlock;
    printf("sharedMemPerBlock: %zd bytes\n", sharedMemory); //
        %zd is used to print size_t values
}
```

Which outputs:

```
sharedMemPerBlock: 49152 bytes
```

Parallel Reduction With Divergence But Using Shared Memory Fix

Solution

- 1D Grid with 1D block of 1024 threads in the x axis.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#define vecSize 1024

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__host__ void validate(int* result_CPU, int* result_GPU, int
   size) {
```

```
18      if (*result_CPU != *result_GPU) {
19          printf("The results are not equal.\n");
20          return;
21      }
22      printf("Kernel validated successfully.\n");
23      return;
24  }
25
26  __host__ void CPU_fn(int* v, int* sum, const int size) {
27      for (int i = 0; i < size; i++) {
28          *sum += v[i];
29      }
30  }
31
32  __global__ void kernel_divergent(int* v, int* sum) {
33      int gId = threadIdx.x;
34      int step = vecSize / 2;
35
36      while (step) {
37          if (gId < step) {
38              v[gId] = v[gId] + v[gId + step];
39          }
40          step = step / 2;
41      }
42      if (gId == 0) {
43          *sum = v[gId];
44      }
45  }
46
47  __global__ void kernel_divergent_fixed(int* v, int* sum) {
48      int gId = threadIdx.x;
49      // vector in shared memory
50      __shared__ int vectorShared[vecSize];
51      vectorShared[gId] = v[gId];
52      // old GPU's need thread synchronize when using shared
           memory: shared memory is visible for all blocks
53      __syncthreads();
54      int step = vecSize / 2; // avoid using blockDim to avoid
           unexpected behaviours
55
56      while (step) {
57          //printf("%d\n", step);
58          if (gId < step) {
59              vectorShared[gId] = vectorShared[gId] +
                   vectorShared[gId + step];
60          }
61          step = step / 2;
62      }
63      if (gId == 0) {
64          *sum = vectorShared[gId];
65      }
```

```
66 }
67
68 int main() {
69
70     const int size = 1024;
71     int* v = (int*)malloc(sizeof(int) * size);
72     int sumCPU = 0;
73     int sumGPU = 0;
74
75     int* dev_v, * sum;
76     cudaMalloc((void**)&dev_v, sizeof(int) * size);
77     cudaMalloc((void**)&sum, sizeof(int));
78
79     for (int i = 0; i < size; i++) {
80         v[i] = 1;
81     }
82
83     cudaMemcpy(dev_v, v, sizeof(int) * size,
           cudaMemcpyHostToDevice);
84     cudaMemcpy(sum, &sumGPU, sizeof(int),
           cudaMemcpyHostToDevice);
85
86     dim3 grid(1);
87     dim3 block(size);
88
89     kernel_divergent_fixed << < grid, block >> > (dev_v, sum);
90     cudaMemcpy(&sumGPU, sum, sizeof(int),
           cudaMemcpyDeviceToHost);
91     printf("GPU sum: %d\n", sumGPU);
92
93     CPU_fn(v, &sumCPU, size);
94     printf("CPU sum: %d\n", sumCPU);
95
96     validate(&sumCPU, &sumGPU, size);
97
98     return 0;
99 }
```

Output

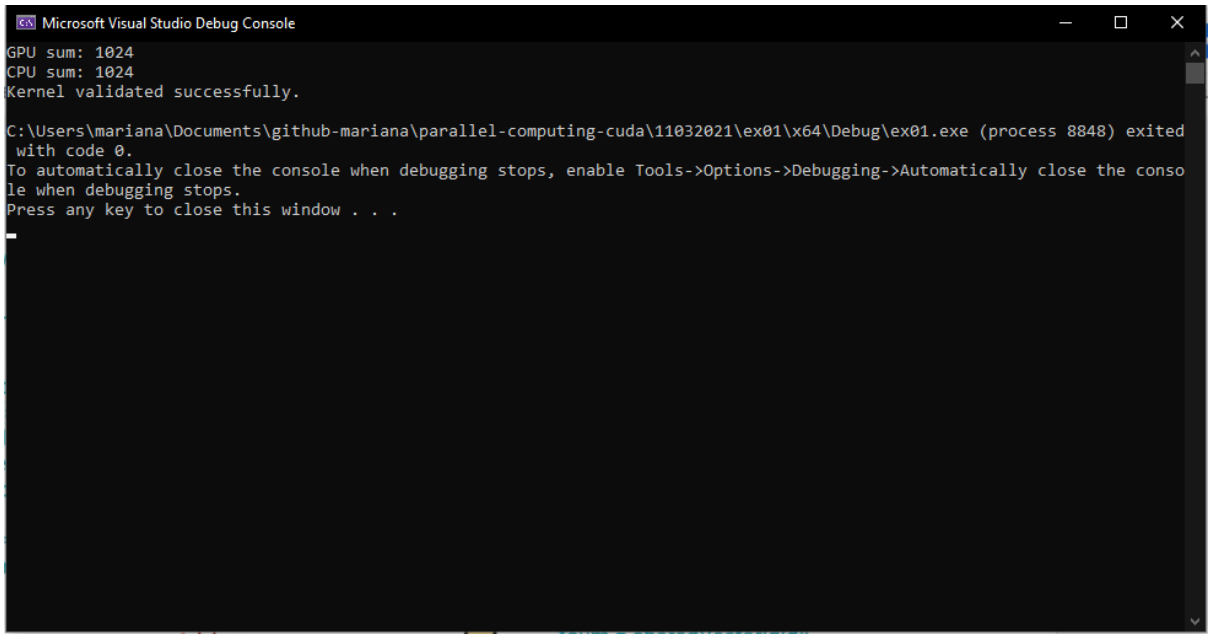


Figure 1: img

Shared Memory with More than 1 Block

Imagine we launch four blocks in a kernel, each block with 8 threads, as shown below.

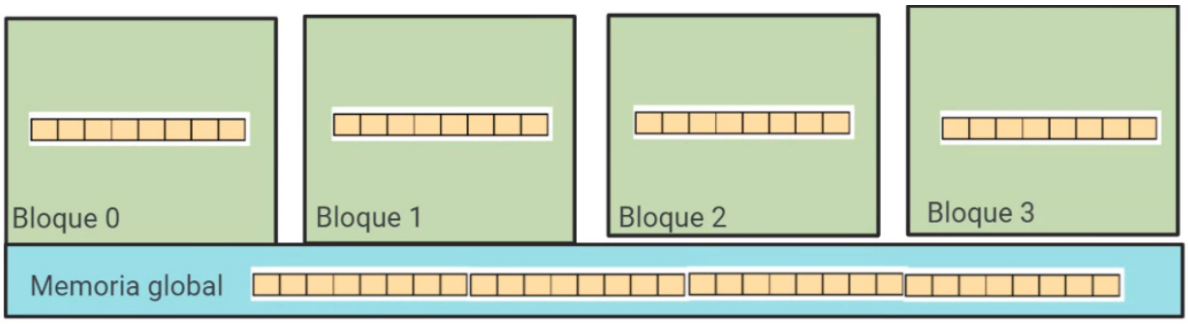


Figure 1: img

Thus, we will manipulate a vector of size 32 (4x8), the same size as the total threads launched.

In Global Memory, we have the vector thanks to `cudaMalloc()`. Global Memory is a memory that all threads from all blocks can see.

Last time, in the kernel we typed `__shared__ int vector[32]`, so that in a block there was a 32 vector in shared memory. But now, when launch many blocks the shared memory vector `__shared__ int vector[32]` will not be reserved 4 times, but the vector size will be split among all the blocks launched: each block has now a vector of size 8.

But now we need to copy from the Global Memory vector of size 32 into the shared vector size 8 in each block: the first block copies the first 8 values, and so on. The gId is computed:

```
int gId = threadIdx.x + blockDim.x * blockIdx.x;
```

So that it looks as follows:

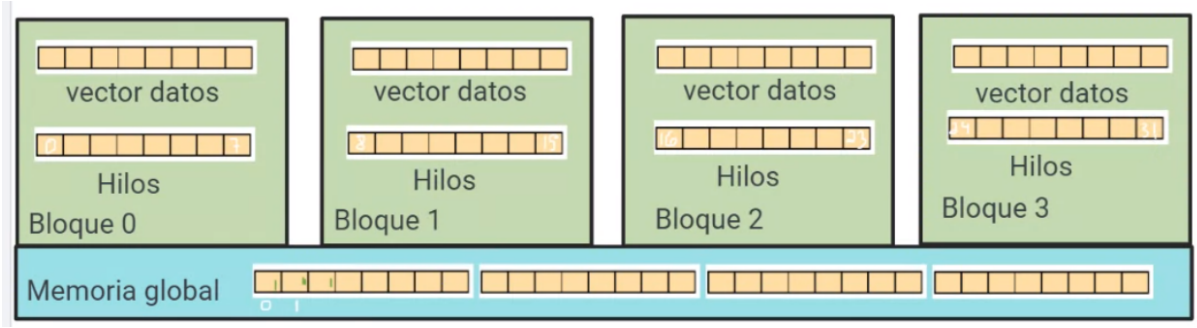


Figure 2: img

The block 1 cannot write anything on the shared vector of another block. Then, how to manage the shared vector indices in order to copy the global vector to the shared one?

The shared vector can be accessed with the **same** indices as if it was on global memory, because the reservation is global from the kernel, and thus the indices have continuity among blocks.

```
__global__ void kernel(int *v){
    int gId = threadIdx.x + blockDim.x * blockIdx.x;
    __shared__ int vector[32];
    vector[gId] = v[gId];
}
```

Now, instead of partitioning the full 32 size vector, now we split into half the shared vector of size 8 that is in each block, and do the same thing over and over to perform a parallel reduction.

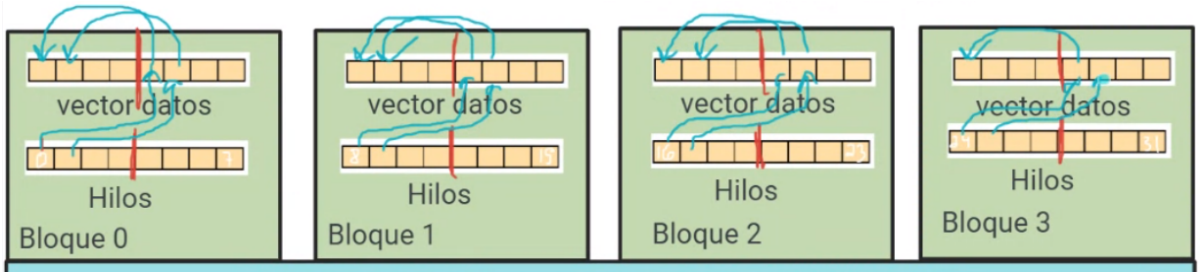


Figure 3: img

But what will be our step variable? Instead of 32 /2, we need 8 / 2 initially. Consider that we have the following defined macros (that are known to the kernel):

```
#define numThreadsPerBlock 8
#define numBlocks 4
#define vecSize numThreadsPerBlock*numBlocks
```

Thus,

```
__global__ void kernel(int *v){
    int gId = threadIdx.x + blockDim.x * blockIdx.x;
    __shared__ int vector[32];
    vector[gId] = v[gId];
    int step = numThreadsPerBlock / 2;
}
```

and now, how to choose the first 4 threads of each block, in order to begin the parallel reduction step?

Figure 4: img

```
__global__ void kernel(int *v){
    int gId = threadIdx.x + blockDim.x * blockIdx.x;
    __shared__ int vector[32];
    vector[gId] = v[gId];
    int step = numThreadsPerBlock / 2;
    if (threadIdx.x < step){ // the first 4 threads per block: 0 1 2 3
        // use gId to move from block to block spaces in the shared vector (always the first 4)
        // then add step to take the next half value and sum it up
        vector[gId] += vector[gId + step];
    }
}
```

But where is our final result? consider that the parallel reduction now is done per block:

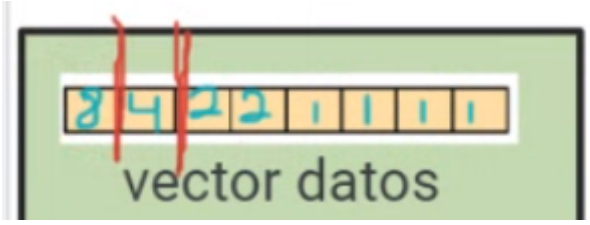


Figure 5: img

In the end, because the block is accessing only to its shared vector piece, thus the partial sum is in the first element of each block's vector of size 8, but we cannot access them accross blocks. This is where the **global memory comes back**: we copy the number 8 in each block shared vector's first cell to the global memory vector, using the global id of such thread so that they write it in the global memory vector as:

Figure 6: img

```
__global__ void kernel(int *v){
    int gId = threadIdx.x + blockDim.x * blockIdx.x;
    __shared__ int vector[32];
    vector[gId] = v[gId];
    int step = numThreadsPerBlock / 2;
    if (threadIdx.x < step){ // the first 4 threads per block: 0 1 2 3
        vector[gId] += vector[gId + step];
    }
```

```
9       if (threadIdx.x == 0){ // copy the partial results to the
            global mem vector
10          v[gId] = vector[gId];
11      }
12  }
```

Now back in the GM vector, we need to apply again a parallel reduction again, but to do that we need to copy all the 8's to the first 4 cells of the GM vector:

Figure 7: img

We will choose these 0 1 2 3 threads by choosing the threads with the gIds that are **smaller** than the number of blocks:

```
1   __global__ void kernel(int *v){
2       int gId = threadIdx.x + blockDim.x * blockIdx.x;
3       __shared__ int vector[32];
4       vector[gId] = v[gId];
5       int step = numThreadsPerBlock / 2;
6       if (threadIdx.x < step){ // the first 4 threads per block:
            0 1 2 3
7           vector[gId] += vector[gId + step];
8       }
9       if (threadIdx.x == 0){ // copy the partial results to the
            global mem vector
10          v[gId] = vector[gId];
11      }
12      if (gId < numBlocks){ // choose the first 4 threads to
            copy in the first 4 cells the partial sums
13          v[gId] = v[gId*numThreadsPerBlock]; // 0 <- 0*8, 1 <-
                1*8 ...
14      }
15  }
```

Now that we have the partial sums in the beginning cells of the GM vector, we need to add all these using parallel reduction. The new step now is:

> int step = numBlocks / 2;

```
1   __global__ void kernel(int *v){
2       int gId = threadIdx.x + blockDim.x * blockIdx.x;
3       __shared__ int vector[32];
4       vector[gId] = v[gId];
5       // synch
6       int step = numThreadsPerBlock / 2;
7       if (threadIdx.x < step){ // the first 4 threads per block:
            0 1 2 3
```

```
8           vector[gId] += vector[gId + step];
9           // synch
10      }
11      if (threadIdx.x == 0){ // copy the partial results to the
            global mem vector
12          v[gId] = vector[gId];
13          // synch
14      }
15      if (gId < numBlocks){ // choose the first 4 threads to
            copy in the first 4 cells the partial sums
16          v[gId] = v[gId*numThreadsPerBlock]; // 0 <- 0*8, 1 <-
                1*8 ...
17          // apply parallel reduction over v
18          step = numBlocks / 2;
19          if (gId < step){
20              v[gId] += v[gId + step];
21          }
22      }
23      if (gId == 0){
24          *sum = v[0];
25      }
26  }
```

In the end now we have the total result of the sum in the first cell of the GM vector. Remember to synchronize threads after each shared memory vector usage.

Note: requisite for parallel reduction is that the vector size is an even number (divisible by 2)

Implementation

```
1   #include "cuda_runtime.h"
2   #include "device_launch_parameters.h"
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <iostream>
7
8   #define numBlocks 8
9   #define threadsPerBlock 1024
10
11  using namespace std;
12
13  __host__ void checkCUDAError(const char* msg) {
14      cudaError_t error;
15      cudaDeviceSynchronize();
16      error = cudaGetLastError();
17      if (error != cudaSuccess) {
18          printf("ERROR %d: %s (%s)\n", error,
                cudaGetErrorString(error), msg);
```

```
19      }
20  }
21
22  __host__ void validate(int* result_CPU, int* result_GPU) {
23      if (*result_CPU != *result_GPU) {
24          printf("[FAILED] Kernel validation.\n");
25          return;
26      }
27      printf("[SUCCESS] Kernel validation.\n");
28      return;
29  }
30
31  __host__ void CPU_reduction(int* v, int* sum) {
32      for (int i = 0; i < numBlocks * threadsPerBlock; i++) {
33          *sum += v[i];
34      }
35  }
36
37  __global__ void GPU_reduction(int* v, int* sum) {
38      __shared__ int vector[numBlocks * threadsPerBlock];
39      int gId = threadIdx.x + blockDim.x * blockIdx.x;
40
41      vector[gId] = v[gId];
42      __syncthreads();
43      int step = threadsPerBlock / 2;
44      while (step) {
45          if (threadIdx.x < step) {
46              vector[gId] = vector[gId] + vector[gId + step];
47              __syncthreads();
48          }
49          step = step / 2;
50      }
51      __syncthreads();
52      if (threadIdx.x == 0) { // copy the partial results to the
            global mem vector
53          //printf("SM->vector[%d]: %d\n", gId, vector[gId]);
54          v[gId] = vector[gId];
55          __syncthreads();
56          //printf("GM->v[%d]: %d\n", gId, v[gId]);
57      }
58      if (gId < numBlocks) { // choose the first 4 threads to
           copy in the first 4 cells the partial sums
59          v[gId] = v[gId * threadsPerBlock]; // 0 <- 0*8, 1 <-
               1*8 ...
60          __syncthreads();
61          //printf("%d<-%d\n", v[gId], v[gId * threadsPerBlock])
               ;
62      }
63      int new_step = numBlocks / 2;
64      while (new_step) {
65          if (gId < new_step) {
```

```
66              v[gId] += v[gId + new_step];
67          }
68          new_step = new_step / 2;
69      }
70      __syncthreads();
71      if (gId == 0) {
72          *sum = v[gId];
73      }
74  }
75
76  int main() {
77
78      int* dev_a, * dev_sum;
79      int host_sum = 0, CPU_sum = 0;
80      int* host_a = (int*)malloc(sizeof(int) * numBlocks *
           threadsPerBlock);
81      cudaMalloc((void**)&dev_a, sizeof(int) * numBlocks *
           threadsPerBlock);
82      cudaMalloc((void**)&dev_sum, sizeof(int));
83
84      for (int i = 0; i < numBlocks * threadsPerBlock; i++) {
85          host_a[i] = 1;
86      }
87
88      cudaMemcpy(dev_a, host_a, sizeof(int) * numBlocks *
           threadsPerBlock, cudaMemcpyHostToDevice);
89
90      dim3 grid(numBlocks, 1, 1);
91      dim3 block(threadsPerBlock, 1, 1);
92      GPU_reduction << < grid, block >> > (dev_a, dev_sum);
93      cudaDeviceSynchronize();
94      checkCUDAError("Error at kernel");
95
96      printf("N: %d\n", numBlocks * threadsPerBlock);
97      cudaMemcpy(&host_sum, dev_sum, sizeof(int),
           cudaMemcpyDeviceToHost);
98      printf("GPU result: %d\n", host_sum);
99
100     CPU_reduction(host_a, &CPU_sum);
101     printf("CPU result: %d\n", CPU_sum);
102
103     validate(&CPU_sum, &host_sum);
104
105     return 0;
106 }
```

Output

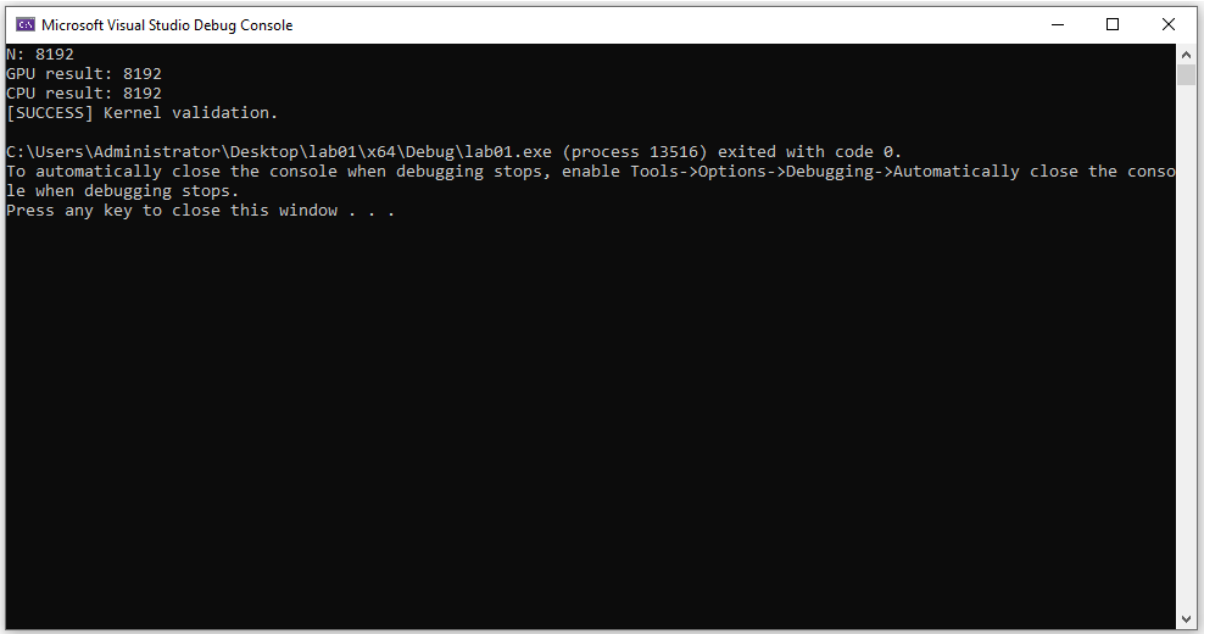


Figure 8: img

Constant Memory

Apart from all the other memories shown in the previous diagram, we have Constant Memory:

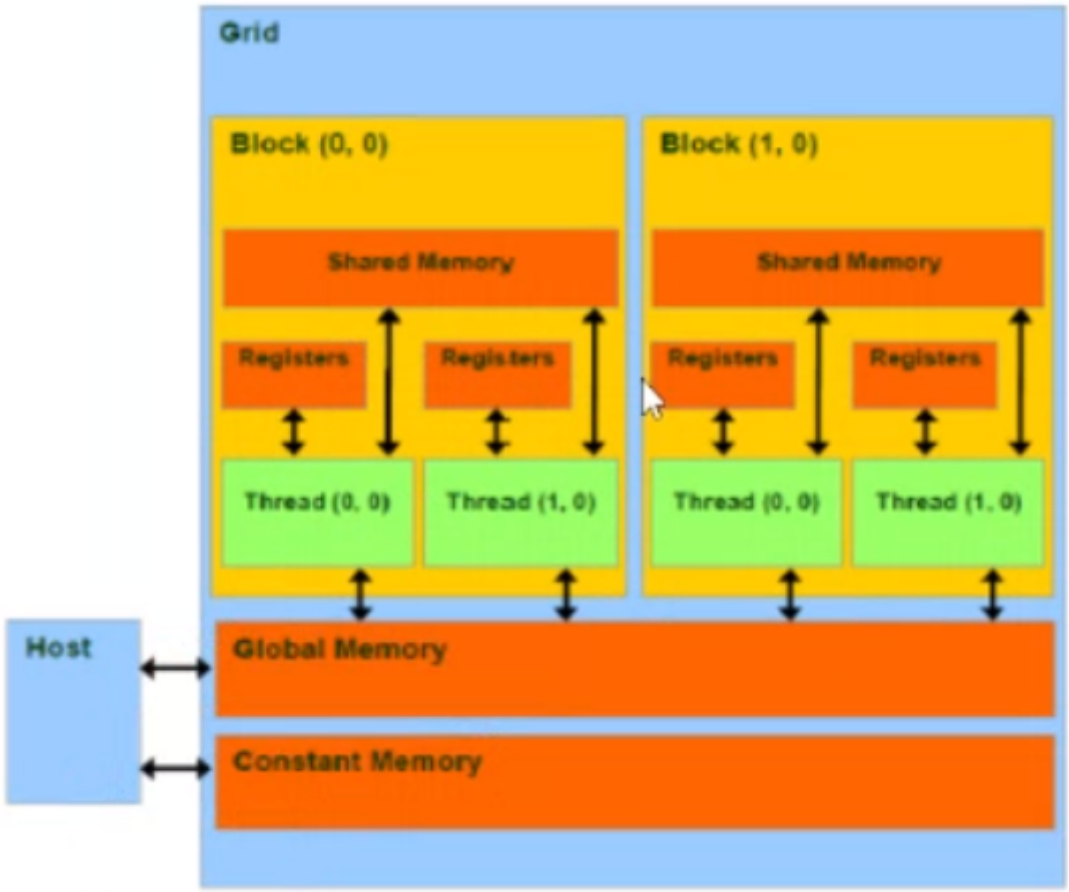


Figure 1: img

The difference with this memory is that it is a **read-only** memory access, thus it is used from data that we just need to read. This memory is inside the Device, but its reservation it's done outside a kernel:

```
1 #define N 32
2 __constant__ int dev_A[N*N];
3 __global__ void kernel(int* dev_A, int* dev_B){
4
5 }
6 int main(){
7
8 }
```

Before using this constant memory, we used to send the information stored in global

memory (*devPtr) as a parameter dev_A, and another vector for the results dev_B. Now, we can replace the reservation of dev_A in the Global Memory. After tne reservation, we used `cudaMemcpy()` to copy the data to dev_A. We also can forget about this step, and instead we transfer the data from the host to dev_A using `cudaMemcpyToSymbol(dev_A, host_A, sizeof(int)*N*N)`: this new function does not require the direction of the transference.

Constant Memory is advised to be used when we only have read-only data, because the memory access is faster than Global Memory.

Implementation

Given a square matrix of size N, output the transpose of such matrix using Constant Memory.

Figure 2: img

Generate a vector in the host: `[1,2,3,4,5,6,7,8,9]`, and copy them to the constant memory using `cudaMemcpyToSymbol(dev_A, host_A, sizeof(int)*N*N)`, now in dev_A. The result will be stored in dev_B as `[1,4,7,2,5,8,3,6,9]`. We do not need to send dev_A as parameter to the kernel, only parameter dev_B is sent. Thus, the result is in Global Memory, in dev_B. Therefore, we only need `cudaMalloc` for dev_B.

- 1 1D grid
- 1 2D block

```
dim3 grid(1);
dim3 block(N, N);
```

- Use validation for the kernel

Solution

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

```

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

#define N 32
__constant__ int dev_A[N * N];

using namespace std;

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
           cudaGetErrorString(error), msg);
    }
}

__host__ void validate(int* result_CPU, int* result_GPU) {
    for (int i = 0; i < N * N; i++) {
        if (*result_CPU != *result_GPU) {
            printf("[FAILED] The results are not equal.\n");
            return;
        }
    }
    printf("[SUCCESS] Kernel validation.\n");
    return;
}

__host__ void CPU_transpose(int* vector, int* res) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[(i * N) + j] = vector[(N * j) + i];
        }
    }
}

__global__ void GPU_transpose(int* res) {
    int gId = threadIdx.x + (blockDim.x * threadIdx.y);
    res[gId] = dev_A[N * threadIdx.x + threadIdx.y];
}

__host__ void printMtx(int* mtx) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << mtx[(i * N) + j] << " ";
        }
        cout << endl;
    }
}
```

```
54
55 int main() {
56
57     int* dev_B;
58     int* host_B = (int*)malloc(sizeof(int) * N * N);
59     int* cpu_B = (int*)malloc(sizeof(int) * N * N);
60     int* host_A = (int*)malloc(sizeof(int) * N * N);
61
62     cudaMalloc((void**)&dev_B, sizeof(int) * N * N);
63     checkCUDAError("Error at cudaMalloc: dev_B");
64
65     for (int i = 0; i < N * N; i++) {
66         host_A[i] = i + 1;
67     }
68
69     cudaMemcpyToSymbol(dev_A, host_A, sizeof(int) * N * N);
70     checkCUDAError("Error at MemcpyToSymbol");
71
72     dim3 grid(1);
73     dim3 block(N, N);
74     GPU_transpose << < grid, block >> > (dev_B);
75     checkCUDAError("Error at kernel");
76     cudaMemcpy(host_B, dev_B, sizeof(int) * N * N,
           cudaMemcpyDeviceToHost);
77     checkCUDAError("Error at Memcpy host_B <- dev_B");
78
79     CPU_transpose(host_A, cpu_B);
80
81     printf("Input: \n");
82     printMtx(host_A);
83     printf("CPU: \n");
84     printMtx(cpu_B);
85     printf("GPU: \n");
86     printMtx(host_B);
87
88     validate(cpu_B, host_B);
89
90     free(host_B);
91     free(cpu_B);
92     free(host_A);
93     cudaFree(dev_B);
94
95     return 0;
96 }
```

Output

```
GPU:
1 33 65 97 129 161 193 225 257 289 321 353 385 417 449 481 513 545 577 609 641 673 705 737 769 801 833 865 897 929 961 993
2 34 66 98 130 162 194 226 258 290 322 354 386 418 450 482 514 546 578 610 642 674 706 738 770 802 834 866 898 930 962 994
3 35 67 99 131 163 195 227 259 291 323 355 387 419 451 483 515 547 579 611 643 675 707 739 771 803 835 867 899 931 963 995
4 36 68 100 132 164 196 228 260 292 324 356 388 420 452 484 516 548 580 612 644 676 708 740 772 804 836 868 900 932 964 996
5 37 69 101 133 165 197 229 261 293 325 357 389 421 453 485 517 549 581 613 645 677 709 741 773 805 837 869 901 933 965 997
6 38 70 102 134 166 198 230 262 294 326 358 390 422 454 486 518 550 582 614 646 678 710 742 774 806 838 870 902 934 966 998
7 39 71 103 135 167 199 231 263 295 327 359 391 423 455 487 519 551 583 615 647 679 711 743 775 807 839 871 903 935 967 999
8 40 72 104 136 168 200 232 264 296 328 360 392 424 456 488 520 552 584 616 648 680 712 744 776 808 840 872 904 936 968 1000
9 41 73 105 137 169 201 233 265 297 329 361 393 425 457 489 521 553 585 617 649 681 713 745 777 809 841 873 905 937 969 1001
10 42 74 106 138 170 202 234 266 298 330 362 394 426 458 490 522 554 586 618 650 682 714 746 778 810 842 874 906 938 970 1002
11 43 75 107 139 171 203 235 267 299 331 363 395 427 459 491 523 555 587 619 651 683 715 747 779 811 843 875 907 939 971 1003
12 44 76 108 140 172 204 236 268 300 332 364 396 428 460 492 524 556 588 620 652 684 716 748 780 812 844 876 908 940 972 1004
13 45 77 109 141 173 205 237 269 301 333 365 397 429 461 493 525 557 589 621 653 685 717 749 781 813 845 877 909 941 973 1005
14 46 78 110 142 174 206 238 270 302 334 366 398 430 462 494 526 558 590 622 654 686 718 750 782 814 846 878 910 942 974 1006
15 47 79 111 143 175 207 239 271 303 335 367 399 431 463 495 527 559 591 623 655 687 719 751 783 815 847 879 911 943 975 1007
16 48 80 112 144 176 208 240 272 304 336 368 400 432 464 496 528 560 592 624 656 688 720 752 784 816 848 880 912 944 976 1008
17 49 81 113 145 177 209 241 273 305 337 369 401 433 465 497 529 561 593 625 657 689 721 753 785 817 849 881 913 945 977 1009
18 50 82 114 146 178 210 242 274 306 338 370 402 434 466 498 530 562 594 626 658 690 722 754 786 818 850 882 914 946 978 1010
19 51 83 115 147 179 211 243 275 307 339 371 403 435 467 499 531 563 595 627 659 691 723 755 787 819 851 883 915 947 979 1011
20 52 84 116 148 180 212 244 276 308 340 372 404 436 468 500 532 564 596 628 660 692 724 756 788 820 852 884 916 948 980 1012
21 53 85 117 149 181 213 245 277 309 341 373 405 437 469 501 533 565 597 629 661 693 725 757 789 821 853 885 917 949 981 1013
22 54 86 118 150 182 214 246 278 310 342 374 406 438 470 502 534 566 598 630 662 694 726 758 790 822 854 886 918 950 982 1014
23 55 87 119 151 183 215 247 279 311 343 375 407 439 471 503 535 567 599 631 663 695 727 759 791 823 855 887 919 951 983 1015
24 56 88 120 152 184 216 248 280 312 344 376 408 440 472 504 536 568 600 632 664 696 728 760 792 824 856 888 920 952 984 1016
25 57 89 121 153 185 217 249 281 313 345 377 409 441 473 505 537 569 601 633 665 697 729 761 793 825 857 889 921 953 985 1017
26 58 90 122 154 186 218 250 282 314 346 378 410 442 474 506 538 570 602 634 666 698 730 762 794 826 858 890 922 954 986 1018
27 59 91 123 155 187 219 251 283 315 347 379 411 443 475 507 539 571 603 635 667 699 731 763 795 827 859 891 923 955 987 1019
28 60 92 124 156 188 220 252 284 316 348 380 412 444 476 508 540 572 604 636 668 700 732 764 796 828 860 892 924 956 988 1020
29 61 93 125 157 189 221 253 285 317 349 381 413 445 477 509 541 573 605 637 669 701 733 765 797 829 861 893 925 957 989 1021
30 62 94 126 158 190 222 254 286 318 350 382 414 446 478 510 542 574 606 638 670 702 734 766 798 830 862 894 926 958 990 1022
31 63 95 127 159 191 223 255 287 319 351 383 415 447 479 511 543 575 607 639 671 703 735 767 799 831 863 895 927 959 991 1023
32 64 96 128 160 192 224 256 288 320 352 384 416 448 480 512 544 576 608 640 672 704 736 768 800 832 864 896 928 960 992 1024
[SUCCESS] Kernel validation.

C:\Users\mariana\Documents\github-mariana\parallel-computing-cuda\11102021\ex01\x64\Debug\ex01.exe (process 15120) exited with
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when
Press any key to close this window . . .
```

Figure 3: img