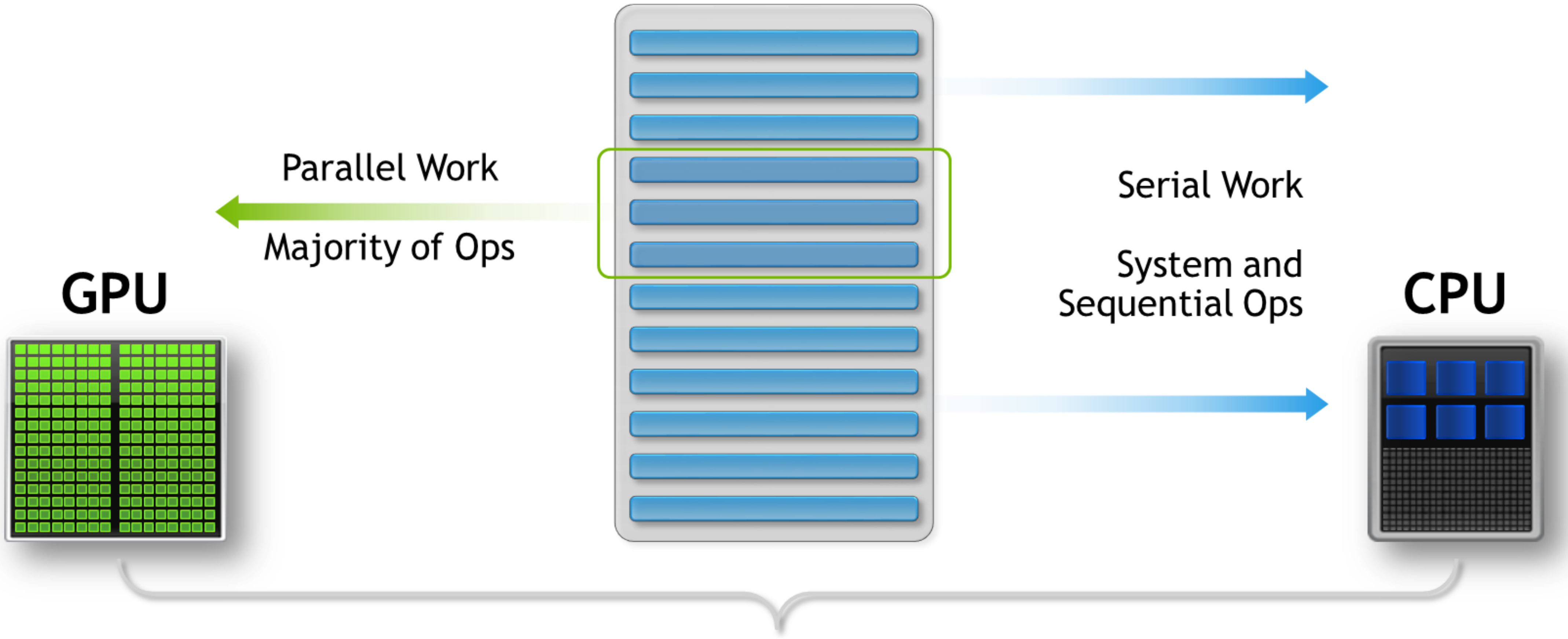
INTRODUCTION TO CUDA PROGRAMMING

INTRODUCTION TO CUDA PROGRAMMING

- Since its first release in 2007, Compute Unified Device Architecture (CUDA) has grown to become the de facto standard when it comes to using Graphic Computing Units (GPUs) for general-purpose computation, that is, non-graphics applications.
- GPUs are used to accelerate the parts of the code that are parallel in nature.
- A highly efficient CPU coupled with a high throughput GPU results in improved
- performance for the application.

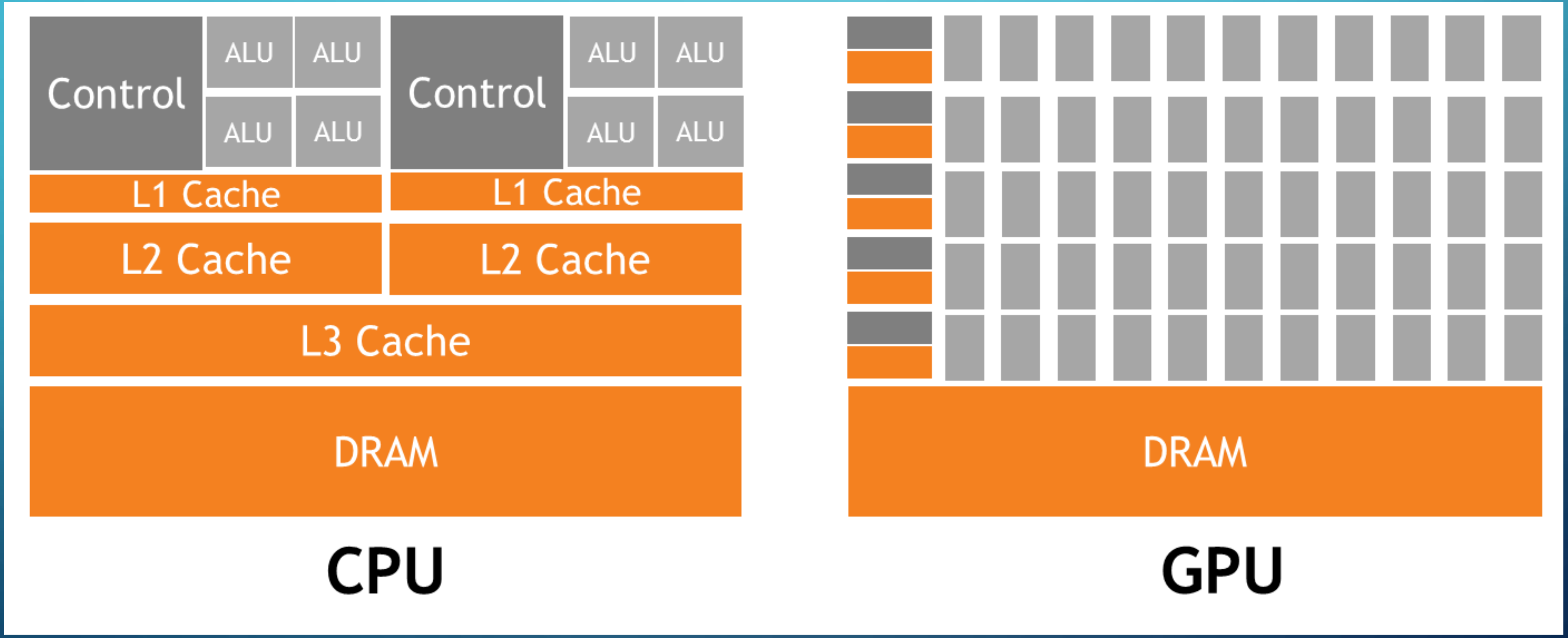


+

HETEROGENEOUS COMPUTING

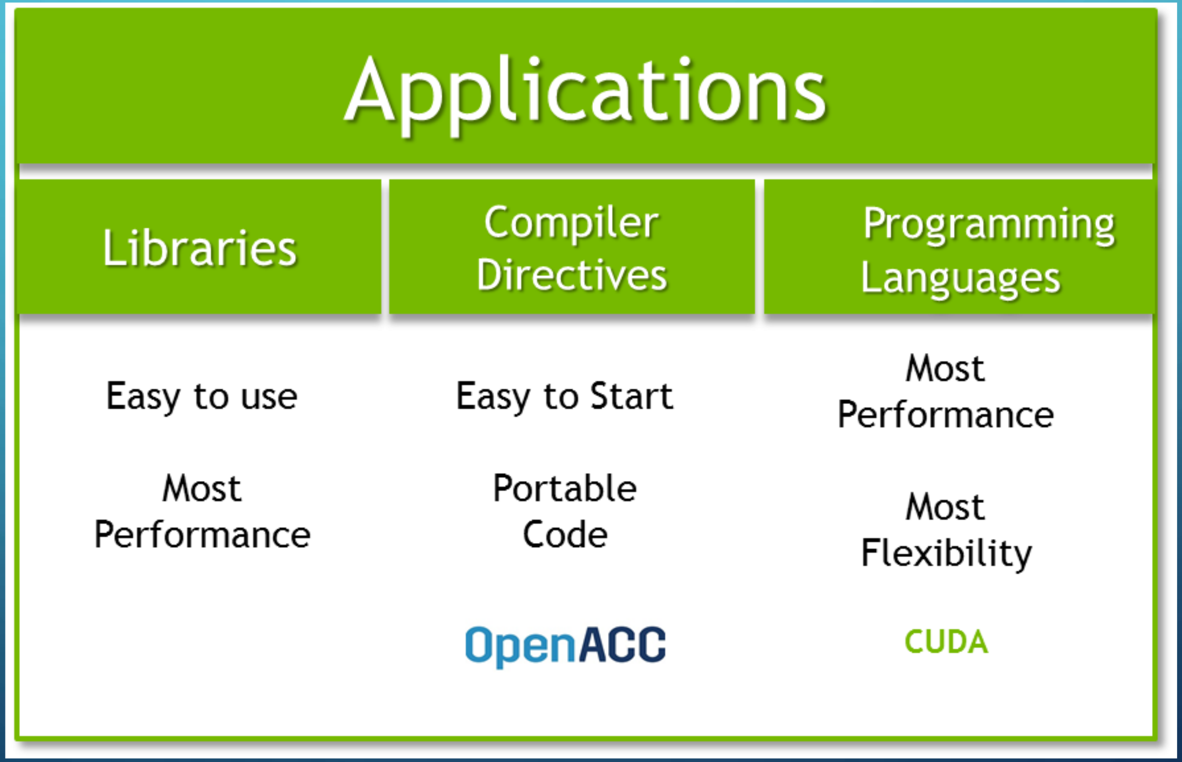
CPU & GPU

- While GPU uses a lot of transistors for computing ALUs, CPU uses it to reduce latency.



CUDA

- CUDA is a parallel computing platform and programming model architecture developed by NVIDIA that exposes general-purpose computations on GPU.

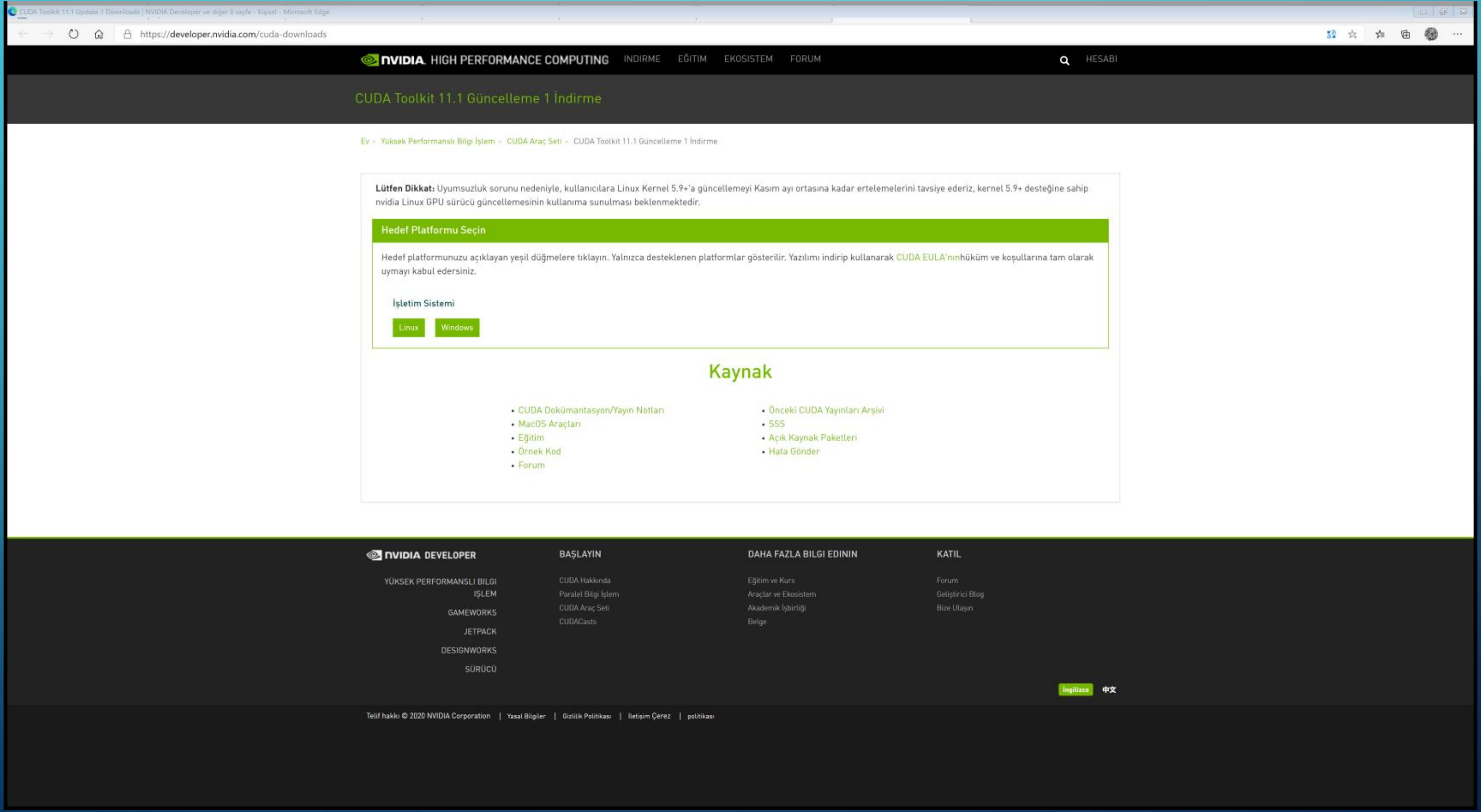


Applications

Libraries	Compiler Directives	Programming Languages
Easy to use	Easy to Start	Most Performance
Most Performance	Portable Code	Most Flexibility
	OpenACC	CUDA

INSTALLING CUDA

- CUDA is supported on Windows, Linux and MacOSX https://developer.nvidia.com/cuda-downloads



CUDA Toolkit 11.1 Update 1 Downloads | NVIDIA Developer ve diğer 6 sayfa - Kişisel - Microsoft Edge

https://developer.nvidia.com/cuda-downloads

NVIDIA. HIGH PERFORMANCE COMPUTING INDIRME EĞITIM EKOSISTEM FORUM HESABI

CUDA Toolkit 11.1 Güncelleme 1 İndirme

Ev > Yüksek Performanslı Bilgi İşlem > CUDA Araç Seti > CUDA Toolkit 11.1 Güncelleme 1 İndirme

Lütfen Dikkat: Uyumsuzluk sorunu nedeniyle, kullanıcılara Linux Kernel 5.9+'a güncellemeyi Kasım ayı ortasına kadar ertelemelerini tavsiye ederiz, kernel 5.9+ desteğine sahip nvidia Linux GPU sürücü güncellemesinin kullanıma sunulması beklenmektedir.

Hedef Platformu Seçin

Hedef platformunuzu açıklayan yeşil düğmelere tıklayın. Yalnızca desteklenen platformlar gösterilir. Yazılımı indirip kullanarak CUDA EULA'nınhüküm ve koşullarına tam olarak uymayı kabul edersiniz.

İşletim Sistemi

Linux Windows

Kaynak

- CUDA Dokümantasyon/Yayın Notları
- MacOS Araçları
- Eğitim
- Örnek Kod
- Forum
- Önceki CUDA Yayınları Arşivi
- SSS
- Açık Kaynak Paketleri
- Hata Gönder

NVIDIA DEVELOPER

YÜKSEK PERFORMANSLI BILGI IŞLEM
GAMEWORKS
JETPACK
DESIGNWORKS
SÜRÜCÜ

BAŞLAYIN
CUDA Hakkında
Paralel Bilgi İşlem
CUDA Araç Seti
CUDACasts

DAHA FAZLA BILGI EDININ
Eğitim ve Kurs
Araçlar ve Ekosistem
Akademik İşbirliği
Belge

KATIL
Forum
Geliştirici Blog
Bize Ulaşın

İngilizce 中文

Telif hakkı © 2020 NVIDIA Corporation | Yasal Bilgiler | Gizlilik Politikası | İletişim Çerez | politikası

HELLO WORLD

- In CUDA, there are two processors that work with each other.
- The host is usually referred to as the CPU, while the device is usually referred to as the GPU.
- The host is responsible for calling the device functions.
- Part of the code that runs on the GPU is called device code, while the serial code that runs on the CPU is called host code.

HELLO WORLD

```
•  #include<stdio.h>
• #include<stdlib.h>
• __global__ void print_from_gpu(void) {
•               printf("Hello World! from thread [%d,%d] \
•                                From device\n", threadIdx.x,blockIdx.x);
• }
• int main(void) {
•               printf("Hello World from host!\n");
•               print_from_gpu<<<1,1>>>();
•               cudaDeviceSynchronize();
• return 0;
• }
```

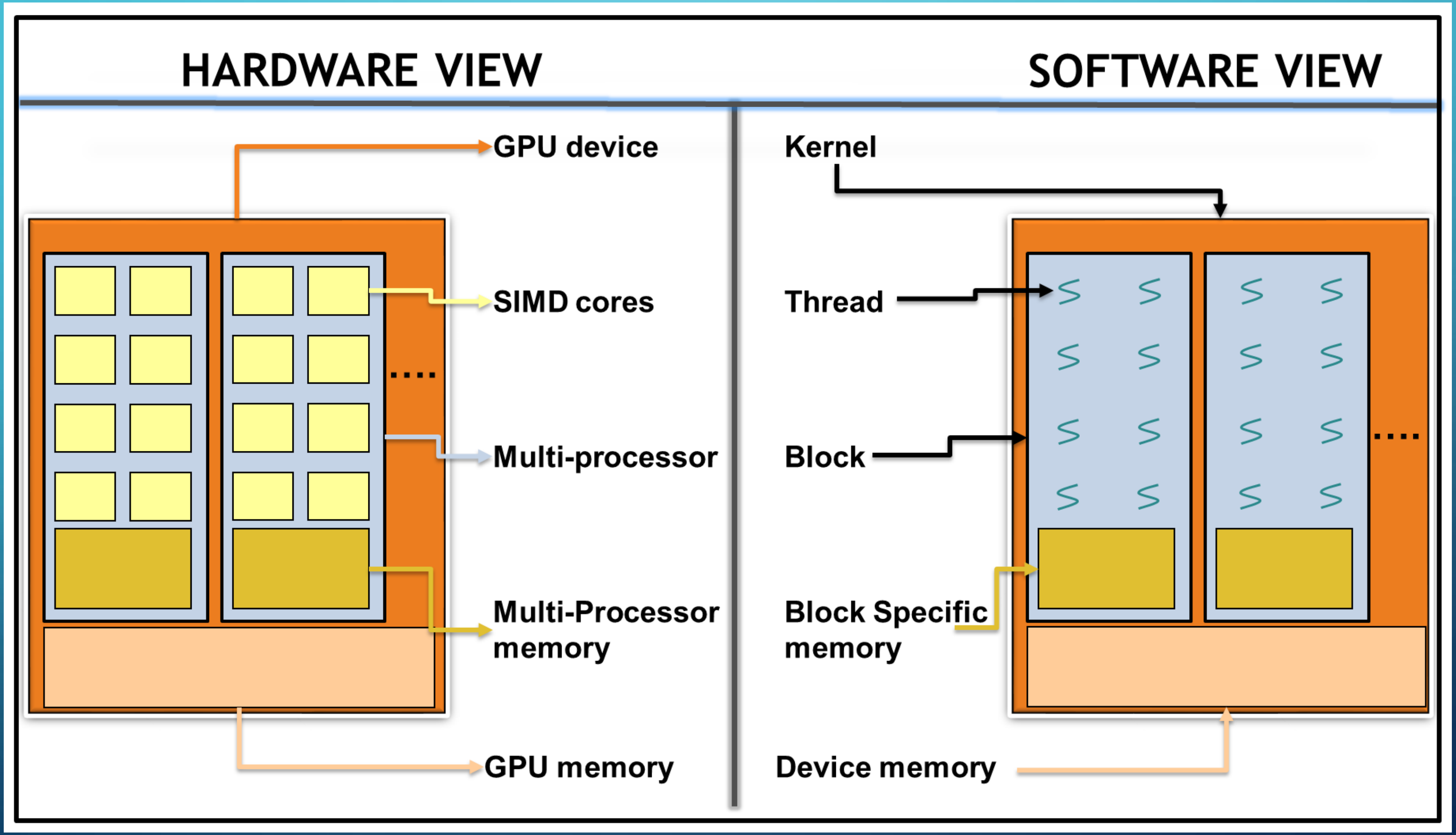
HELLO WORLD

- __global__: This keyword, when added before the function, tells the compiler that this is a function that will run on the device and not on the host. However, note that it is called by the host. Another important thing to note here is that the return type of the device function is always "void". Data-parallel portions of an algorithm are executed on the device as kernels.
- <<<,>>>: This keyword tells the compiler that this is a call to the device function and not the host function. Additionally, the 1,1 parameter basically dictates the number of threads to launch in the kernel.
- threadIdx.x, blockIdx.x: This is a unique ID that's given to all threads.

HELLO WORLD

- cudaDeviceSynchronize(): All of the kernel calls in CUDA are asynchronous in nature.
- The host becomes free after calling the kernel and starts executing the next instruction afterward.
- This should come as no big surprise since this is a heterogeneous environment and hence both the host and device can run in parallel to make use of the types of processors that are available.
- In case the host needs to wait for the device to finish, APIs have been provided as part of CUDA programming that make the host code wait for the device function to finish.
- One such API is cudaDeviceSynchronize, which waits until all of the previous calls to the device have finished.

GPU ARCHITECTURE



HELLO WORLD

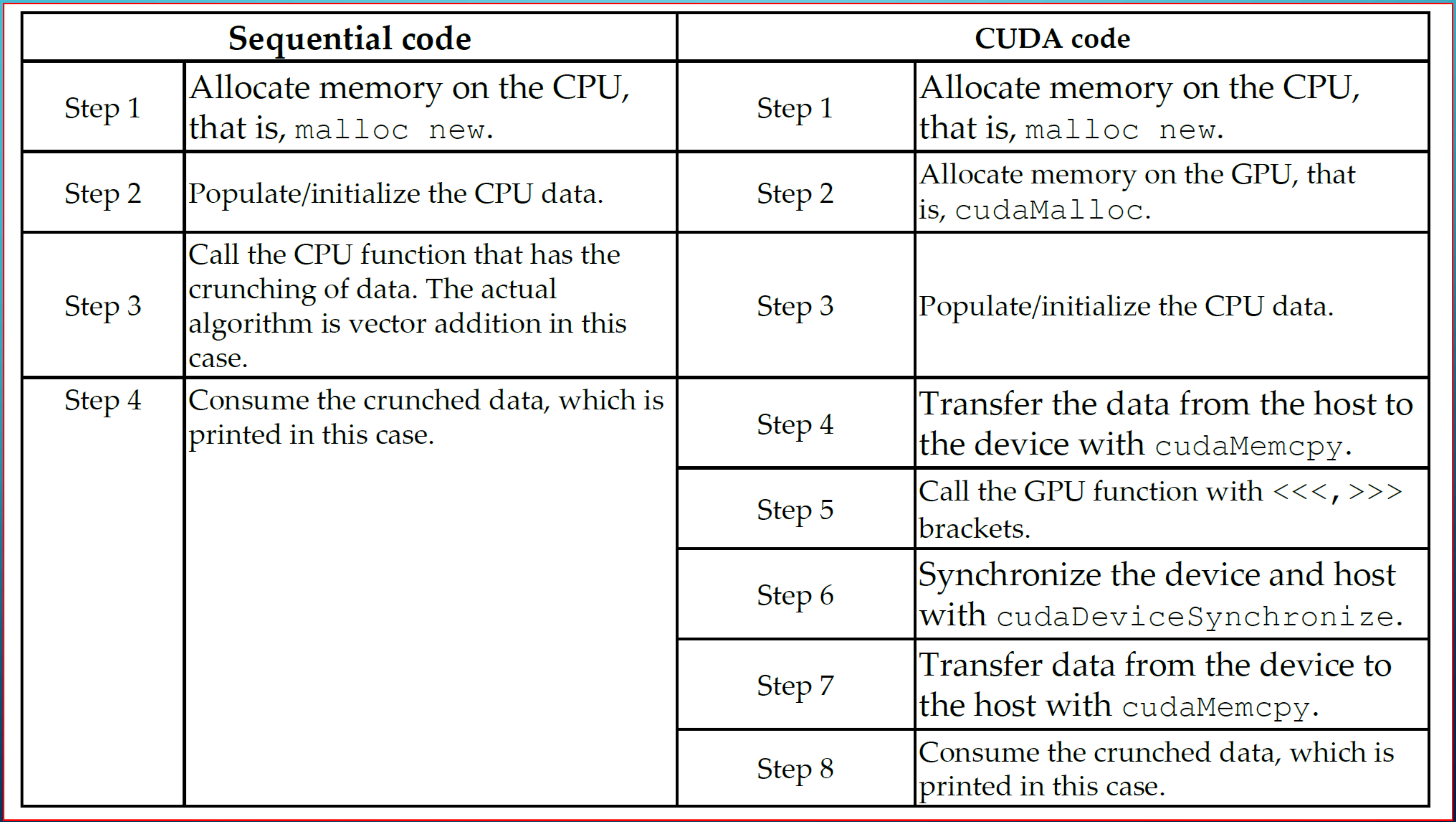
- CUDA Threads: CUDA threads execute on a CUDA core. CUDA threads are different from CPU threads. CUDA threads are extremely lightweight and provide fast context switching. The reason for fast context switching is due to the availability of a large register size in a GPU and hardware-based scheduler. Each CUDA thread must execute the same kernel and work independently on different data (SIMT).
- CUDA blocks: CUDA threads are grouped together into a logical entity called a CUDA block. CUDA blocks execute on a single Streaming Multiprocessor (SM). One block runs on a single SM, that is, all of the threads within one block can only execute on cores in one SM and do not execute on the cores of other SMs. Each GPU may have one or more SM and hence to effectively make use of the whole GPU; the user needs to divide the parallel computation into blocks and threads.
- GRID/kernel: CUDA blocks are grouped together into a logical entity called a CUDA GRID. A CUDA GRID is then executed on the device.

VECTOR ADDITION – SEQUANTIAL CODE

```
#include<stdio.h>
#include<stdlib.h>
#define N 512
void host_add(int *a, int *b, int *c) {
        for(int idx=0;idx<N;idx++)
                c[idx] = a[idx] + b[idx];
}
//basically just fills the array with index.
void fill_array(int *data) {
        for(int idx=0;idx<N;idx++)
                data[idx] = idx;
}
void print_output(int *a, int *b, int*c) {
        for(int idx=0;idx<N;idx++)
                printf("\n %d + %d  = %d",  a[idx] , b[idx], c[idx]);
}
```

```
int main(void) {
        int *a, *b, *c;
        int size = N * sizeof(int);
        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); fill_array(a);
        b = (int *)malloc(size); fill_array(b);
        c = (int *)malloc(size);

        host_add(a,b,c);
        print_output(a,b,c);
        free(a); free(b); free(c);
        return 0;
}
```

VECTOR ADDITION



Sequential code		CUDA code	
Step 1	Allocate memory on the CPU, that is, `malloc new`.	Step 1	Allocate memory on the CPU, that is, `malloc new`.
Step 2	Populate/initialize the CPU data.	Step 2	Allocate memory on the GPU, that is, `cudaMalloc`.
Step 3	Call the CPU function that has the crunching of data. The actual algorithm is vector addition in this case.	Step 3	Populate/initialize the CPU data.
Step 4	Consume the crunched data, which is printed in this case.	Step 4	Transfer the data from the host to the device with `cudaMemcpy`.
		Step 5	Call the GPU function with <<<, >>> brackets.
		Step 6	Synchronize the device and host with `cudaDeviceSynchronize`.
		Step 7	Transfer data from the device to the host with `cudaMemcpy`.
		Step 8	Consume the crunched data, which is printed in this case.

VECTOR ADDITION – CUDA CODE

```
#include<stdio.h>
#include<stdlib.h>
#define N 512

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

//basically just fills the array with index.
void fill_array(int *data) {
        for(int idx=0;idx<N;idx++)
                data[idx] = idx;
}
void print_output(int *a, int *b, int*c) {
        for(int idx=0;idx<N;idx++)
                printf("\n %d + %d  = %d",  a[idx] , b[idx], c[idx]);
}
```

VECTOR ADDITION – CUDA CODE

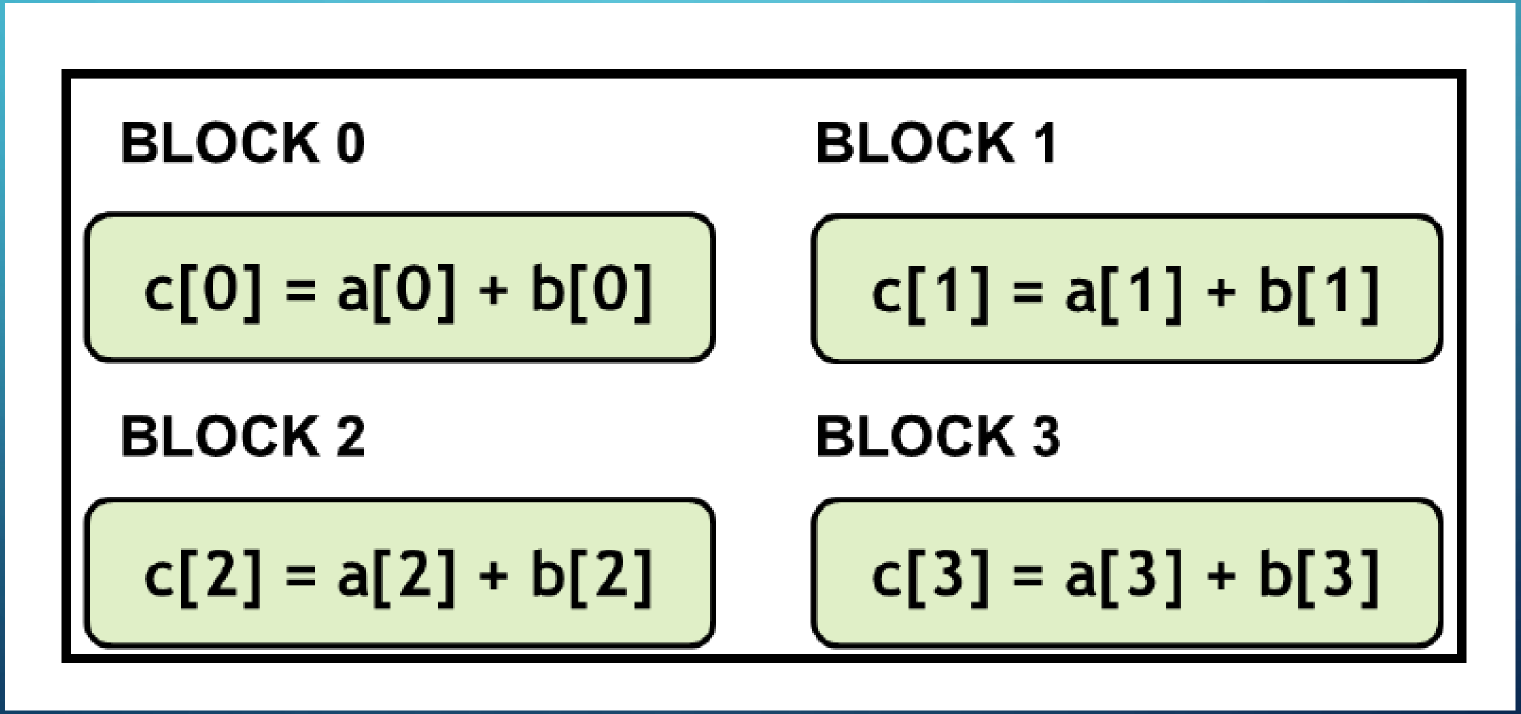
```
int main(void) {
        int *a, *b, *c;
        int *d_a, *d_b, *d_c; // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for host copies of a, b, c and setup input values
        a = (int *)malloc(size); fill_array(a);
        b = (int *)malloc(size); fill_array(b);
        c = (int *)malloc(size);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
         cudaMalloc((void **)&d_c, size);
```

```
        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        device_add<<<N,1>>>(d_a,d_b,d_c);

        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        print_output(a,b,c);

        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

EXPERIMENT 1 – CREATING MULTIPLE BLOCKS

- //changing from device_add<<<1,1>>> to device_add<<<N,1>>>

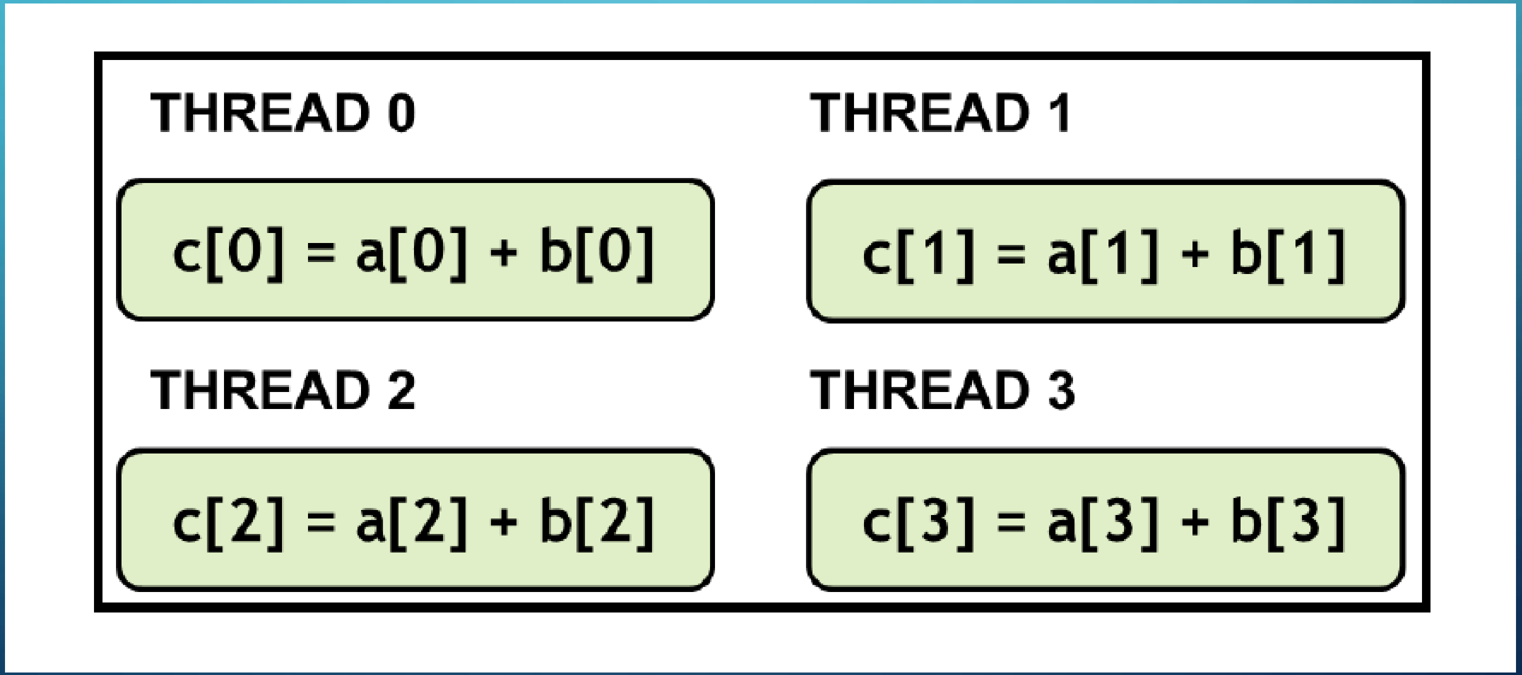
```
__global__ void device_add(int *a, int *b, int *c) {
 c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```



EXPERIMENT 2 – CREATING MULTIPLE THREADS

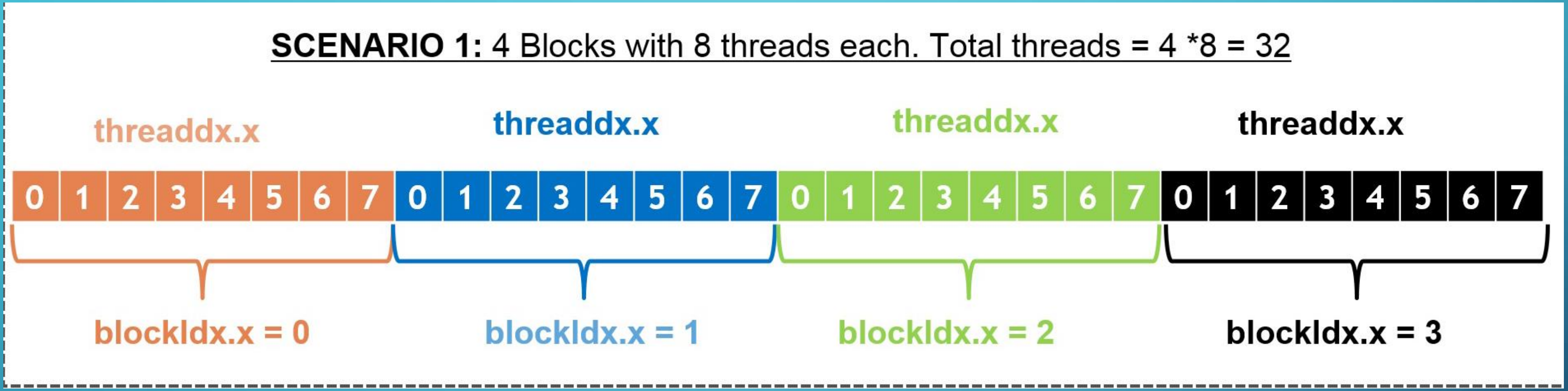
- //changing from device_add<<<1,1>>> to device_add<<<1,N>>>

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

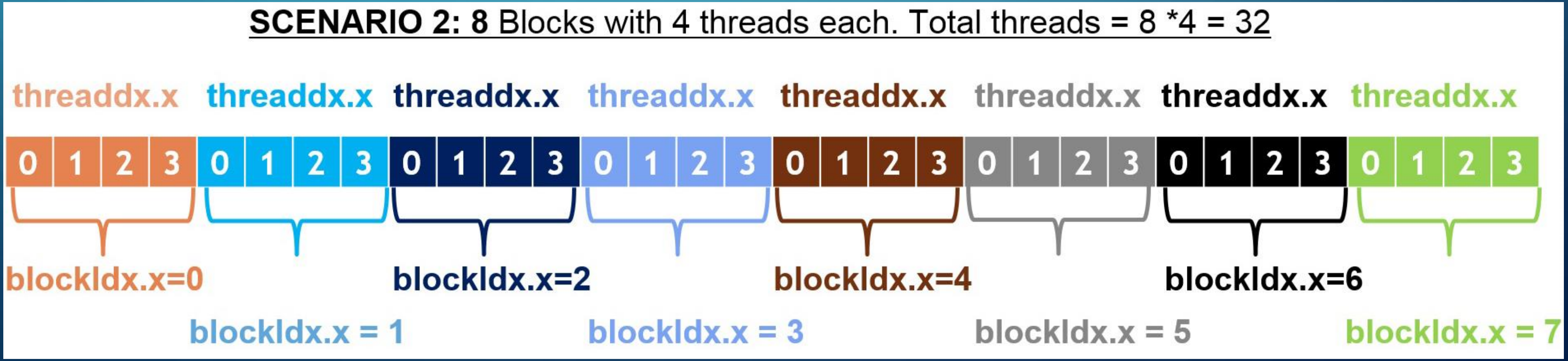


EXPERIMENT 3 – COMBINING BLOCKS AND THREADS

- //changing from device_add<<<1,1>>> to device_add<<<4, 8>>>

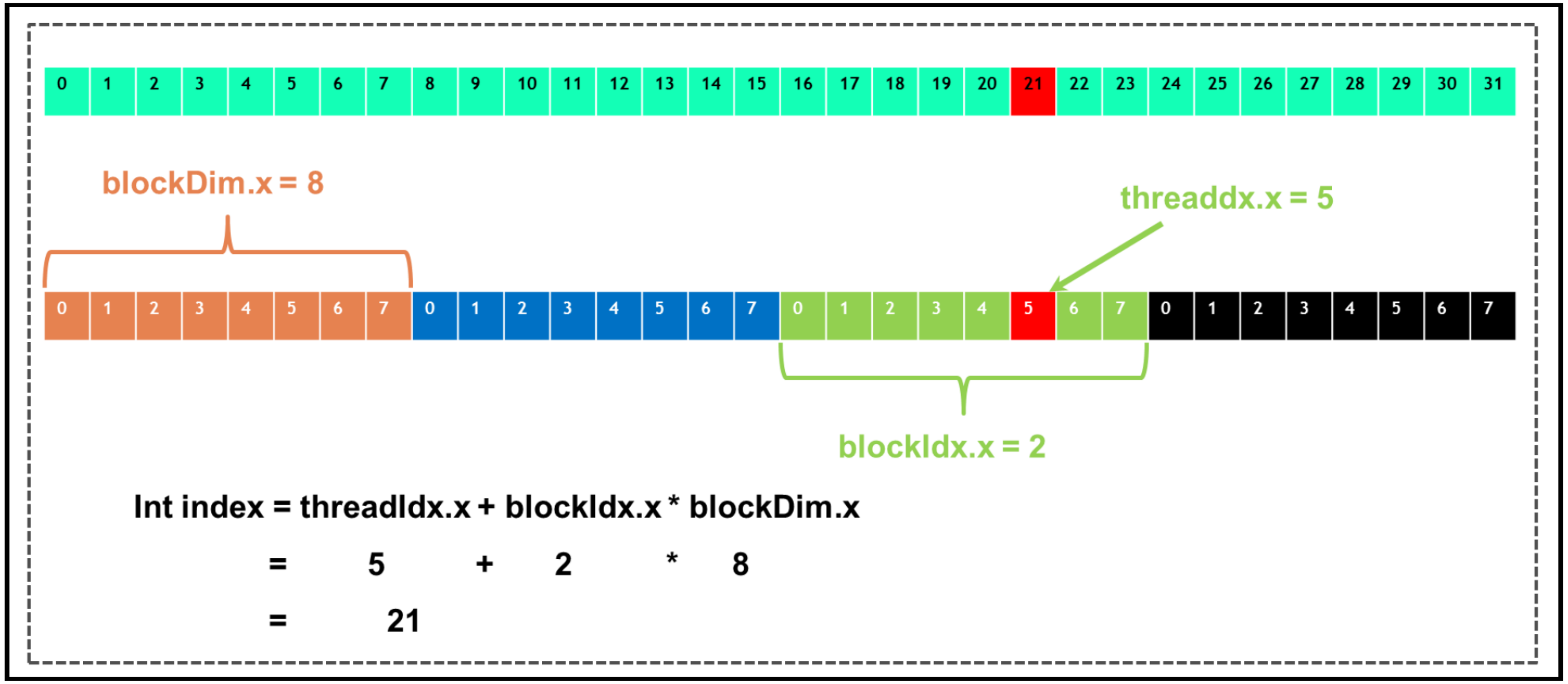


- //changing from device_add<<<1,1>>> to device_add<<<8, 4>>>



EXPERIMENT 3 – COMBINING BLOCKS AND THREADS

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



Int index = threadIdx.x + blockIdx.x * blockDim.x

= 5 + 2 * 8

= 21

ERROR REPORTING IN CUDA

- In CUDA, the host code manages errors. Most CUDA functions call cudaError_t, which is basically an enumeration type.
- cudaSuccess (value 0) indicates a 0 error.
- The user can also make use of the cudaGetErrorString() function, which returns a string describing the error condition, as follows:

```
cudaError_t e;
e = cudaMemcpy(...);
if(e)
    printf("Error: %sn", cudaGetErrorString(err));
```

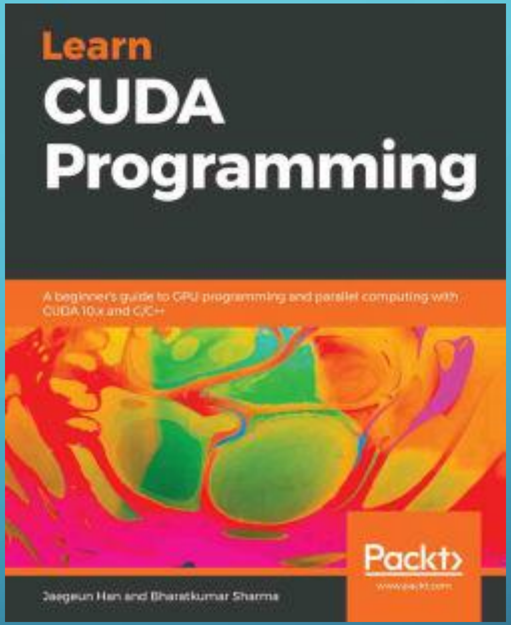
ERROR REPORTING IN CUDA

- Kernel launches have no return value. We can make use of a function such as cudaGetLastError() here, which returns the error code for the last CUDA function.

```
MyKernel<<< ... >>> (...);
cudaDeviceSynchronize();
e = cudaGetLastError();
```

REFERENCES

- Learn CUDA Programming, Jaegeun Han & Bharatkumar Sharma, Packt, 2019


Learn
CUDA
Programming
A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++
Packt›
www.packt.com
Jaegeun Han and Bharatkumar Sharma