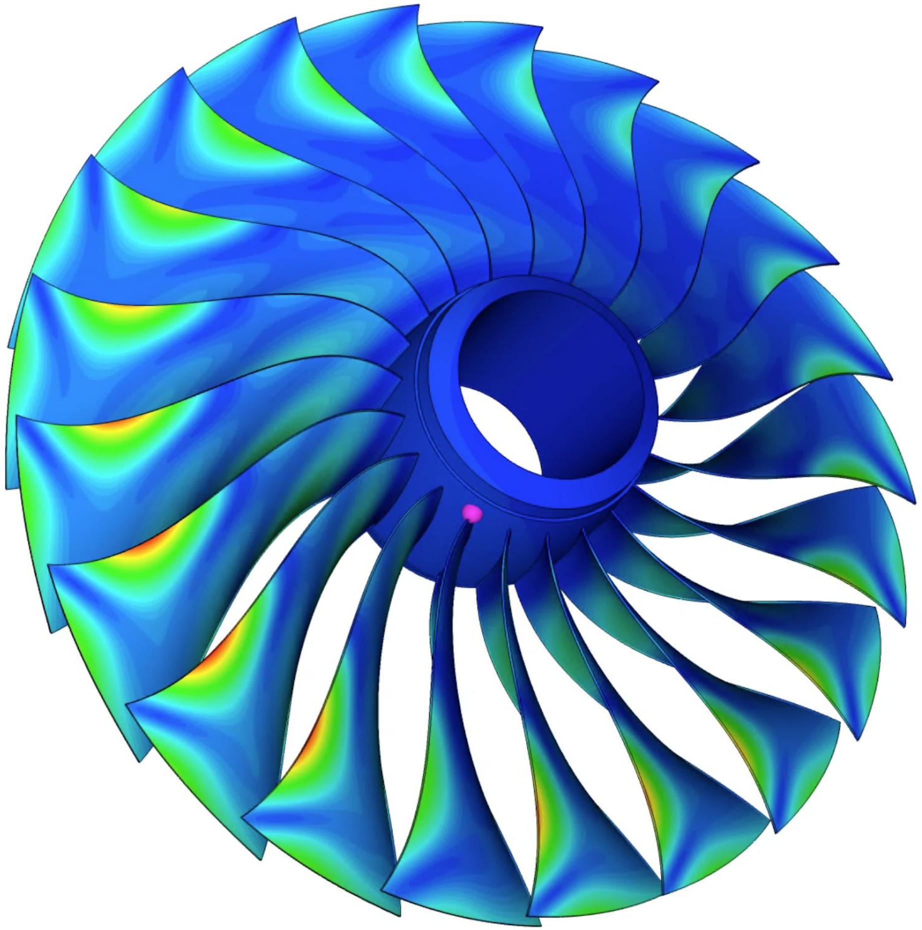
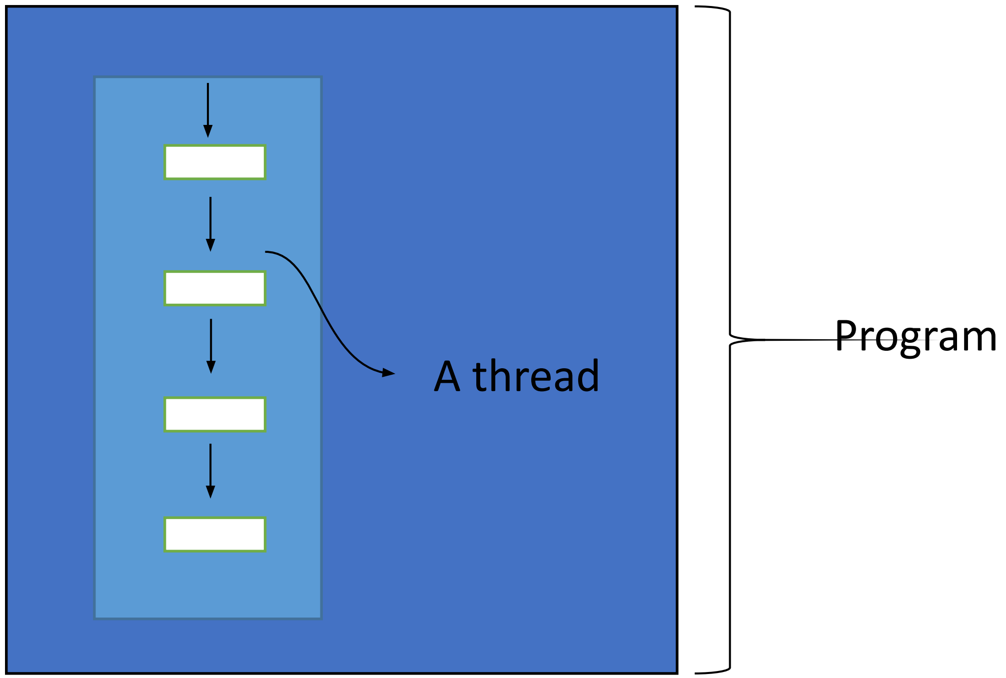
GPU accelerated computing for Finite Element Method



GPU programming - Basics

Thread

- A thread is a single sequential flow of control within a program.
- A sequential code in one processor has one thread.



Introduction to GPU

GPU programming - C library functions

Pointer

- A variable that points to the storage/memory address of another variable.
  - A variable of type certain type will store a value

```
int v = 0;
```

  - This variable has its address (where it is located the memory). This address can be obtained by using ‘&’

```
&v
```

  - A pointer stores the address of the variable

```
int *y = &v;
```

  - The value of the variable can be accessed using the variable or the pointer

```
v
*y
```

Introduction to GPU

GPU programming - C library functions

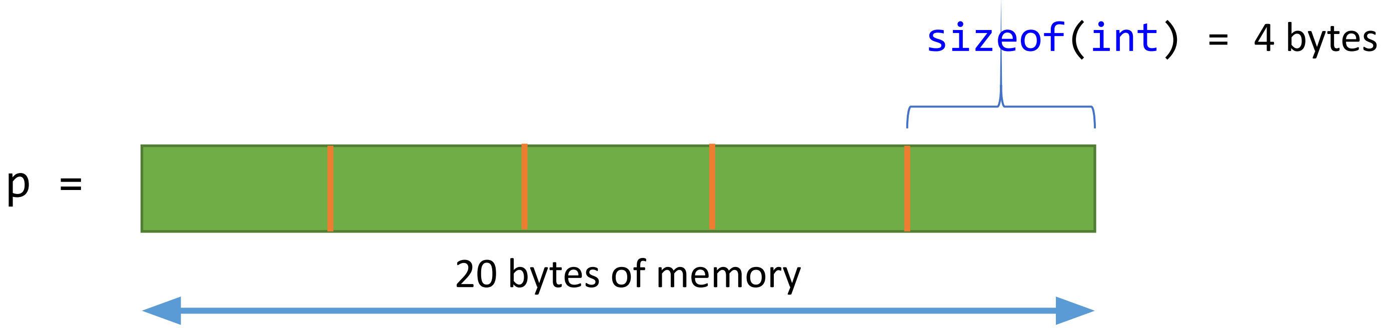
malloc()

- Dynamically allocates a single large block of memory
  - Syntax

```
pointer = (type*) malloc(byte size)
```

  - Example

```c
n = 5;
int *p;
p = (int*)malloc(n * sizeof(int));
```



GPU programming syntax

EXERCISE 0 : Vector Addition using C program

Source code
https://github.com/sivasanarul/FEMwithGPU/tree/master/EX1_vector_addition

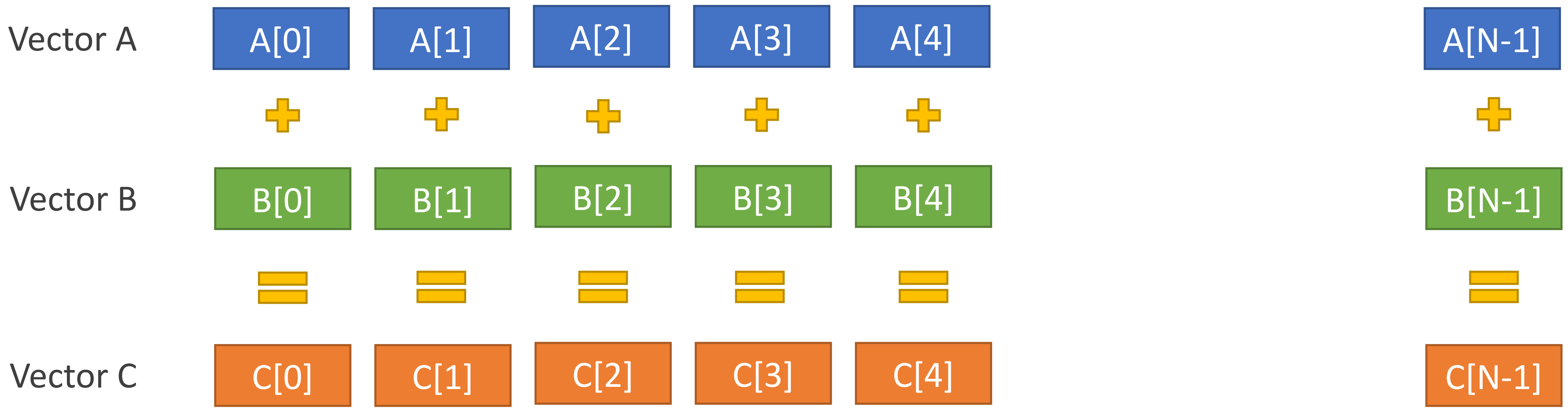
vector_add.c

Introduction to GPU

GPU programming

Vector Addition : vector_addition()

$$A + B = C$$



Introduction to GPU

GPU programming - C program for vector addition

C programming – Vector addition

```c
#include <stdio.h>
#include <stdlib.h>
#define N 100000000

int main(){
    float *a, *b, *c;

    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    c   = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f; b[i] = 2.0f;
    }



    clock_t t;
    t = clock();
    // Main function
    for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];}
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    printf("fun() took %f seconds to execute \n", time_taken);

    free(a);  free(b); free(c);

}
```


```c
float *a, *b, *c;

a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
c   = (float*)malloc(sizeof(float) * N);
```


Memory Allocation for the variables

→


```c
// Initialize array
for(int i = 0; i < N; i++){
    a[i] = 1.0f; b[i] = 2.0f;
}
```


Initializing the variables

↓


```c
// Main function
for(int i=0;i<N;i++){
    c[i] = a[i] + b[i];}
```


Addition of vectors

←


```c
free(a);  free(b); free(c);
```

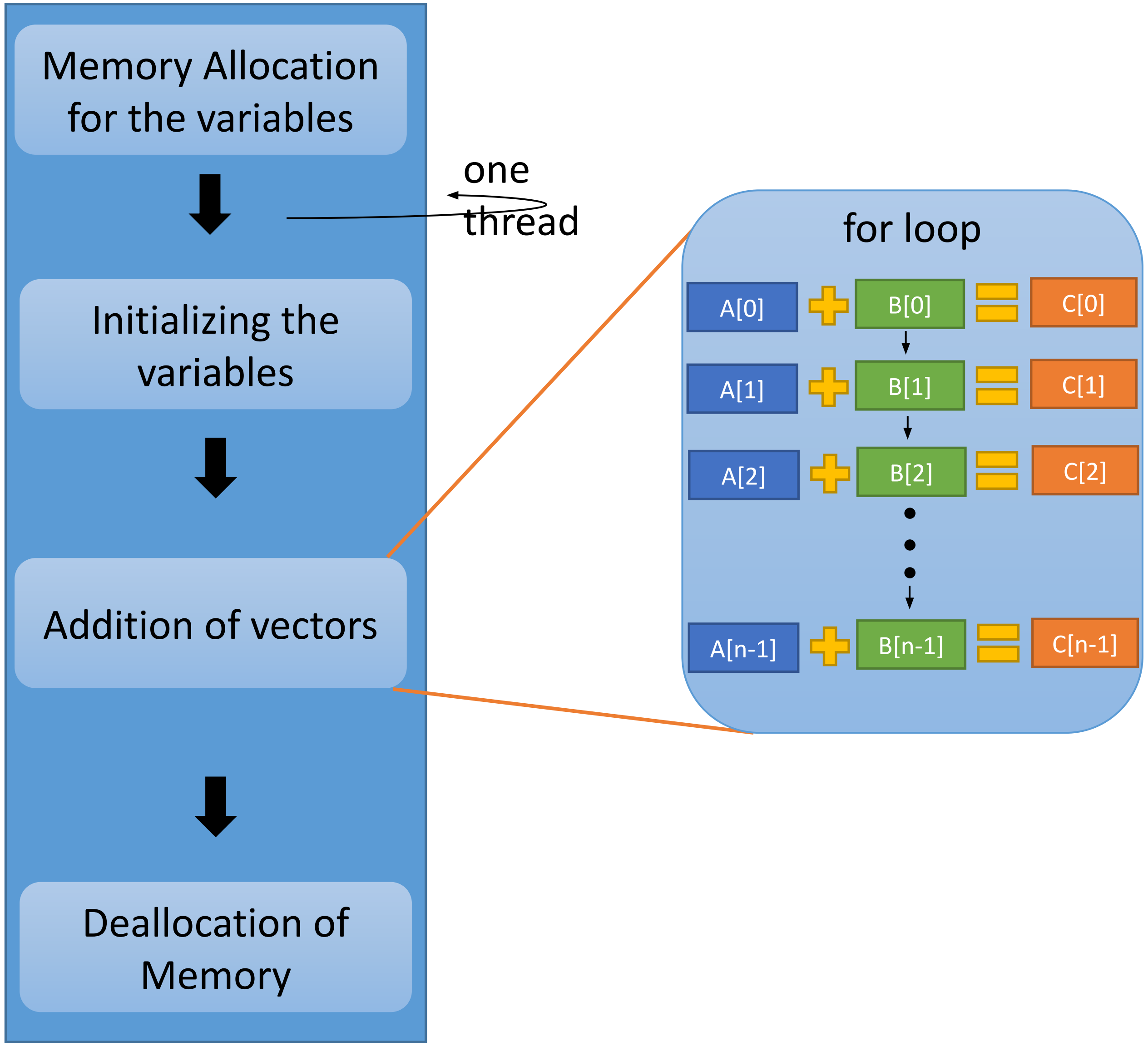

Deallocation of Memory

GPU programming - C program for vector addition

C programming – Vector addition

```c
#include <stdio.h>
#include <stdlib.h>
#define N 100000000

int main(){
    float *a, *b, *c;

    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    c   = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f; b[i] = 2.0f;
    }



    clock_t t;
    t = clock();
    // Main function
    for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];}
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    printf("fun() took %f seconds to execute \n", time_taken);

    free(a);  free(b); free(c);

}
```

Serial Code

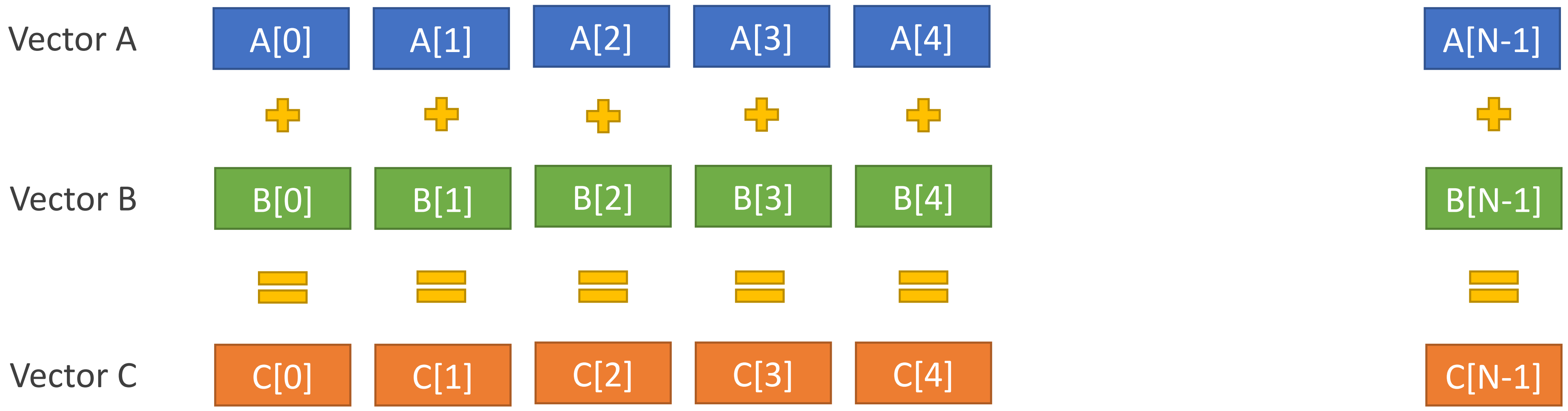


Introduction to GPU

GPU programming

Vector Addition : vector_addition()

$$A + B = C$$



- Divide array into blocks cuda
- threads into multiple blocks

Introduction to GPU

GPU programming - C program for vector addition, heterogenous program

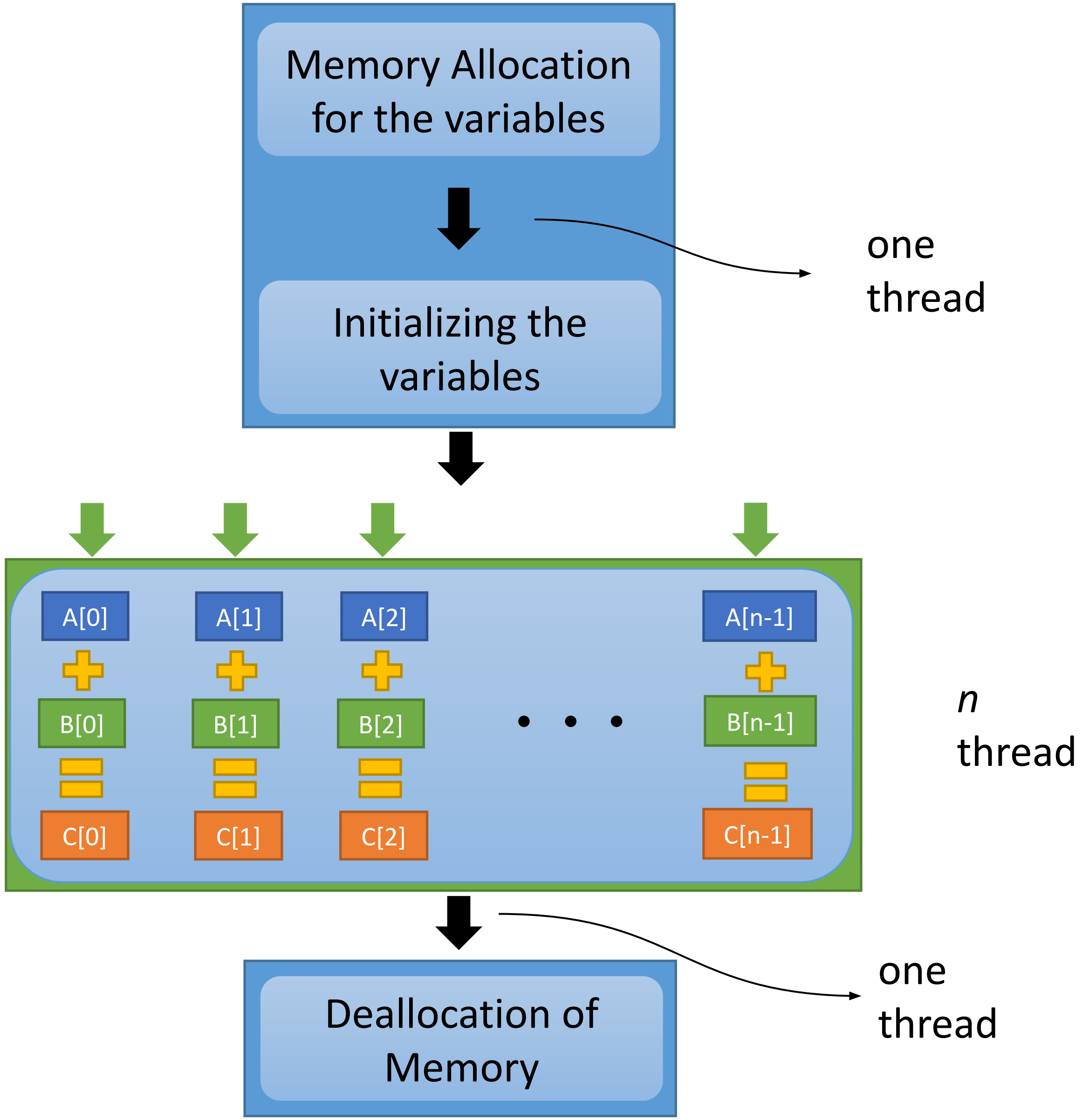
CUDA programming Model – Heterogenous Computing

```c
int main(){
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;

    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    // Allocate device memore for a
    cudaMalloc((void**)&d_a,sizeof(float)*N);
    cudaMalloc((void**)&d_b,sizeof(float)*N);
    cudaMalloc((void**)&d_out,sizeof(float)*N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a,a, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b, sizeof(float)*N, cudaMemcpyHostToDevice);

    // Main function
    int block_size = 256;
    int grid_size  = (N+block_size)/block_size;
    vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);

    cudaMemcpy(out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);

    // Deallocate device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);

    // Deallocate host memory
    free(a);
    free(b);
    free(out);

}
```

Serial Code

Parallel Code

Serial Code



Introduction to GPU

GPU programming - Heterogenous program

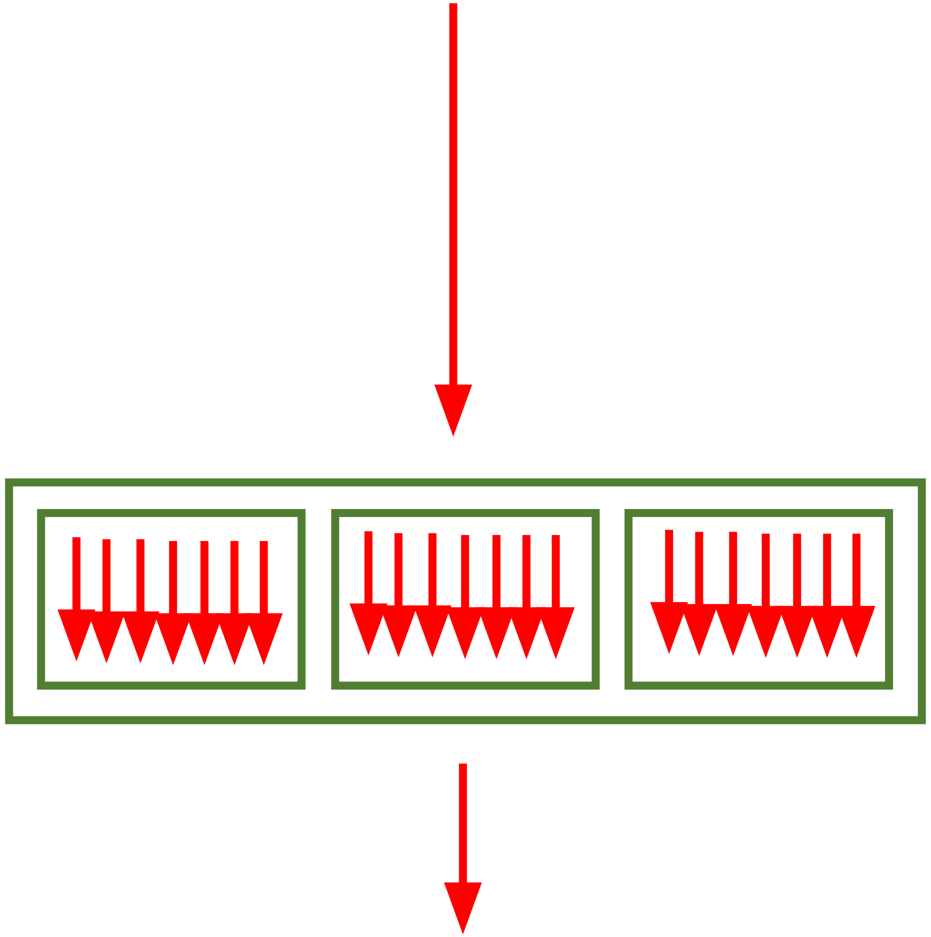
CUDA programming Model – Heterogenous Computing

```
int main(){
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;

    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    // Allocate device memore for a
    cudaMalloc((void**)&d_a,sizeof(float)*N);
    cudaMalloc((void**)&d_b,sizeof(float)*N);
    cudaMalloc((void**)&d_out,sizeof(float)*N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a,a, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b, sizeof(float)*N, cudaMemcpyHostToDevice);

    // Main function
    int block_size = 256;
    int grid_size  = (N+block_size)/block_size;
    vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);

    cudaMemcpy(out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);

    // Deallocate device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);

    // Deallocate host memory
    free(a);
    free(b);
    free(out);

}
```

Serial Code

Parallel Code

Serial Code



Serial Code in host

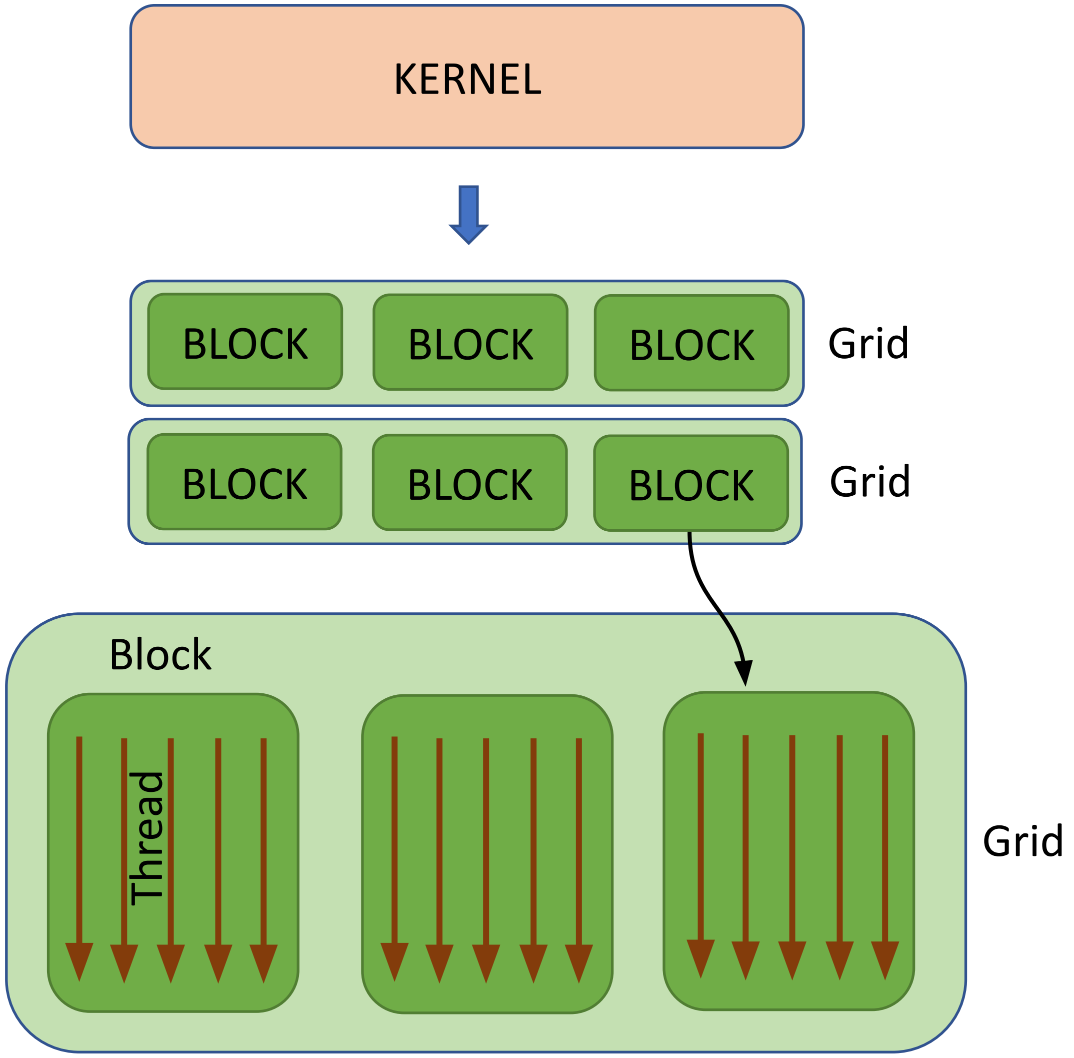
Parallel Code in device

Serial Code in host

Introduction to GPU

Introduction of GPU

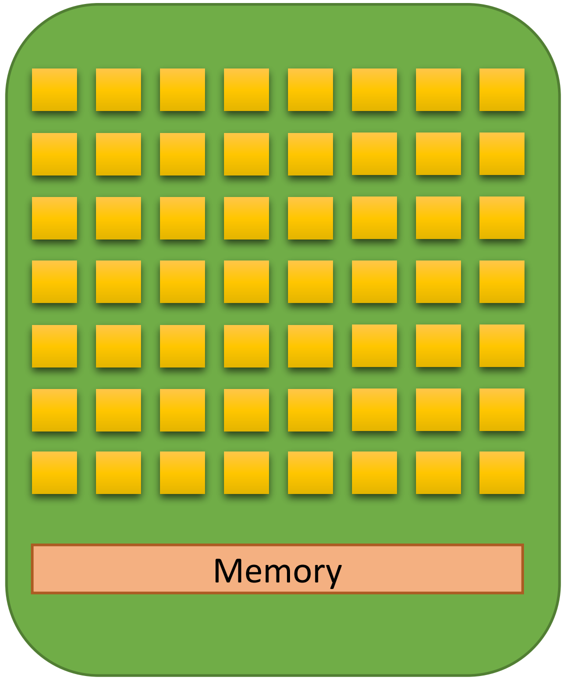
A kernel is a function that is run in a GPU



- A kernel is split into grids
- A grid is split into blocks
- Each block is collection of threads

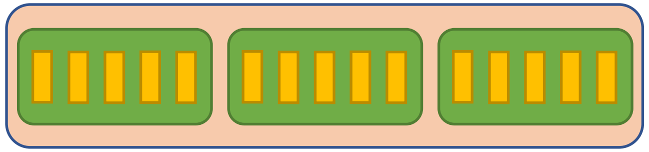
Introduction to GPU

Introduction of GPU



cores

Streaming Multiprocessor – collection of cores

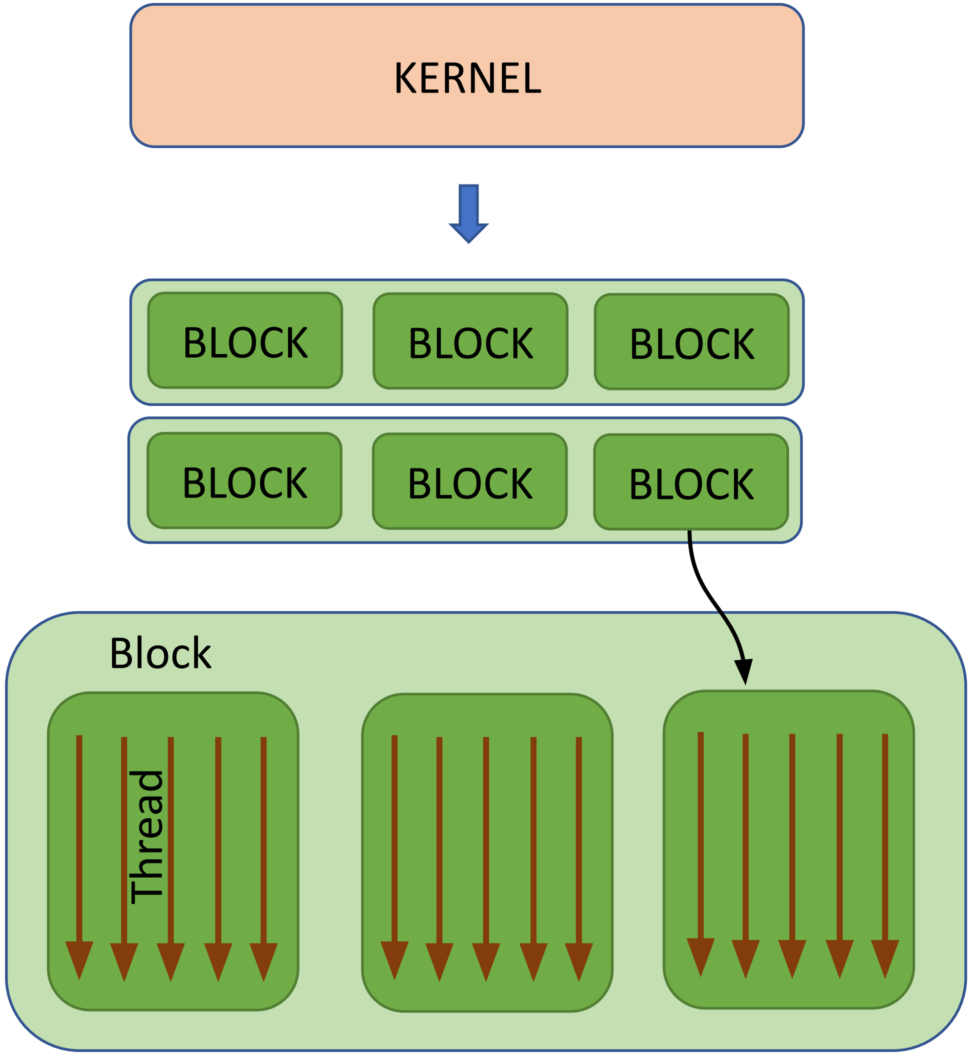


CUDA-enabled GPU – Collection of Streaming Multiprocessor

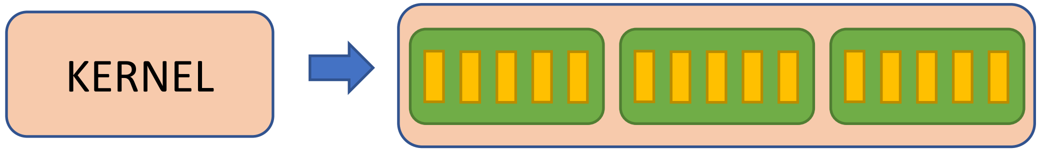
Introduction to GPU

Introduction of GPU

A kernel is a function that is run in a GPU

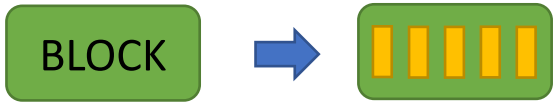


A kernel is executed in a CUDA-enabled GPU



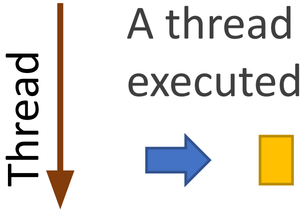
Multiple kernels can be executed in one CUDA-enabled GPU

A block is executed in a Streaming Multiprocessor



Multiple blocks can be executed in one Streaming multiprocessor

A thread is executed in a core



Introduction of GPU programming

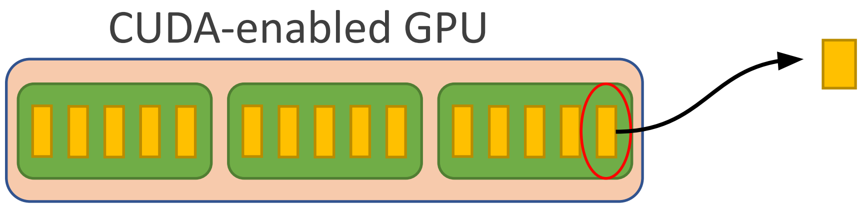
Vector Addition : vector_addition()

- In one core as one thread
- In one streaming multiprocessor as one block
- In the entire GPU device as multiple blocks

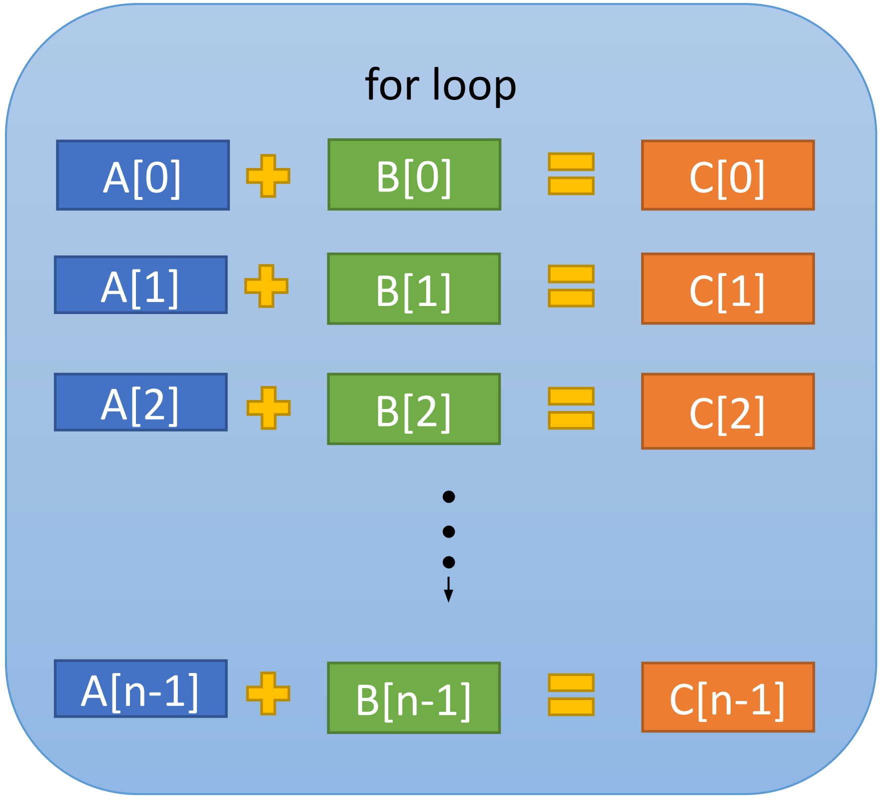
Introduction to GPU

Introduction of GPU programming - vector addition - one thread

Vector Addition : vector_addition()



In one core as one thread



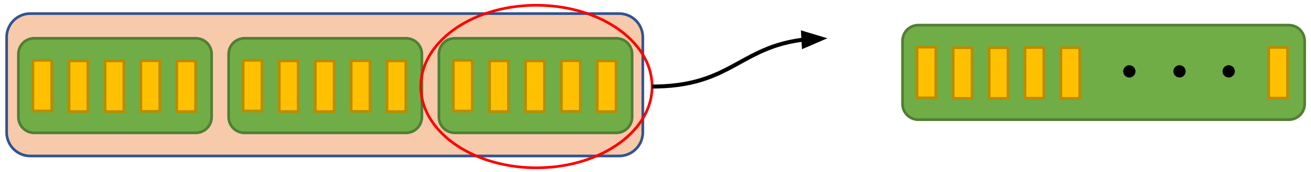

```
__global__ void vector_add(float *out, float *a, float *b, int n){
    for(int i=0;i<n;i++){
        out[i] = a[i] + b[i];}
}
```

Introduction of GPU programming - vector addition - one streaming multiprocessor

Vector Addition : vector_addition()

CUDA-enabled GPU

In one streaming multiprocessor (SM) as one block

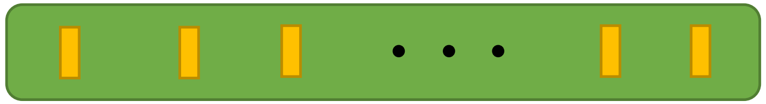


- When the kernel is called, the number of blocks and the number of threads in each block is specified

```
vector_add <<<1,256>>> (d_out, d_a, d_b, N)
```

  - The block runs in a stream multiprocessor (SM)
  - The SM will have 256 thread.
  - Each thread runs in a core.
  - Each thread will have an unique id: `threadIdx.x`

`threadIdx.x` 0 1 2 254 255

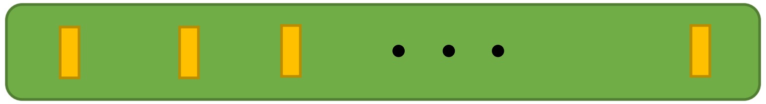


Introduction to GPU

Introduction of GPU programming - vector addition - one streaming multiprocessor

Vector Addition : vector_addition()

threadIdx.x 0 1 2 255



blockDim.x 256

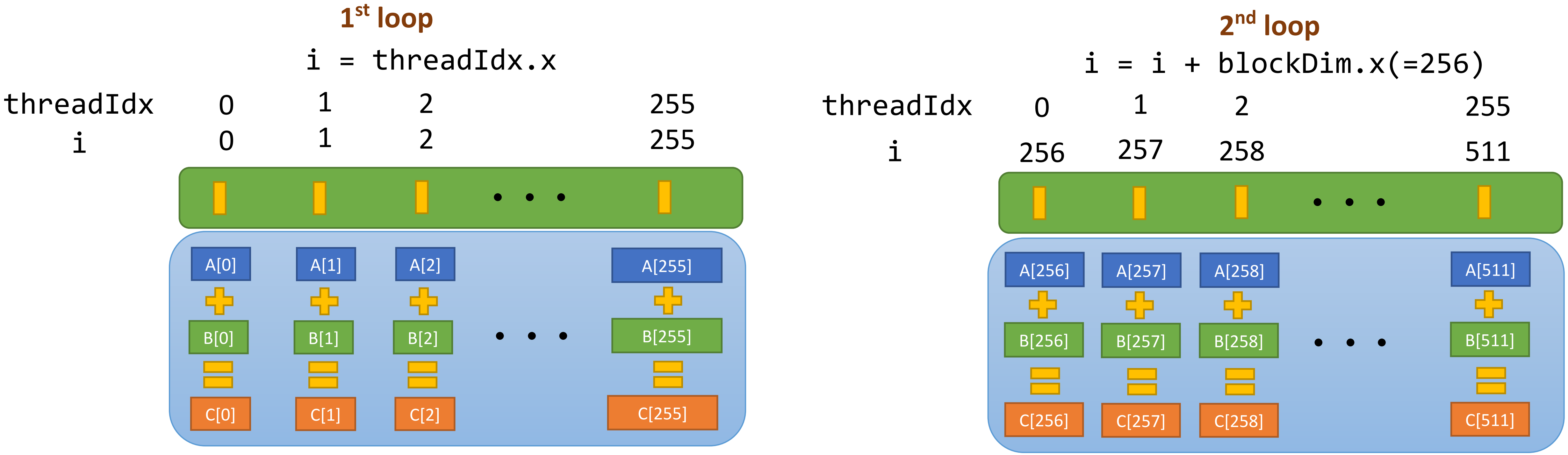
All the threads in the cores run the same function

```
__global__ void vector_add(float *out, float *a, float *b, int n){
      int index = threadIdx.x;
      int stride = blockDim.x;

      for(int i=index;i<n;i+=stride){
         out[i] = a[i] + b[i];}
}
```

Introduction to GPU

Introduction of GPU programming - vector addition - one streaming multiprocessor

Vector Addition : vector_addition()



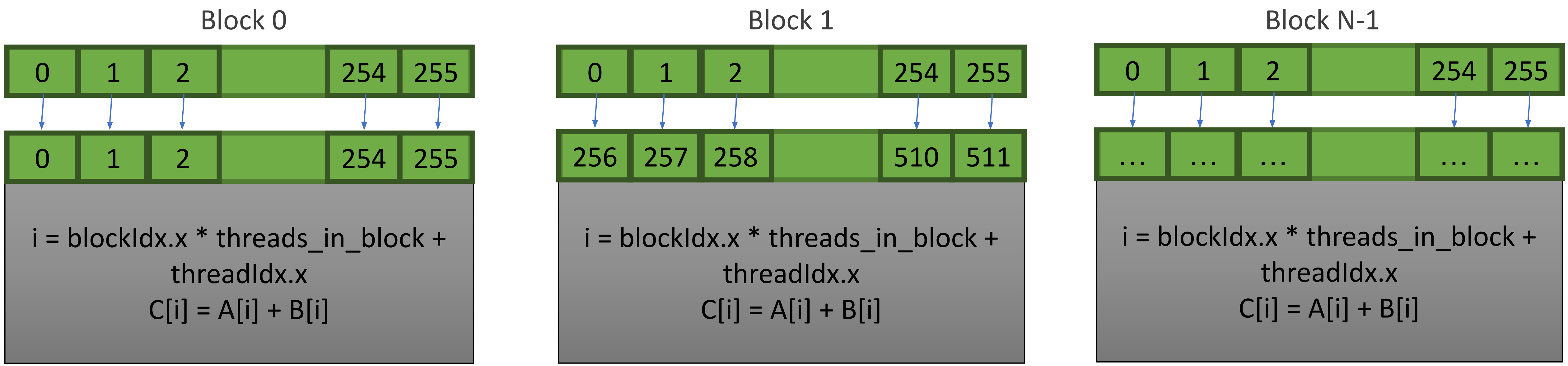

```
__global__ void vector_add(float *out, float *a, float *b, int n){
    int index = threadIdx.x;
    int stride = blockDim.x;

    for(int i=index;i<n;i+=stride){
      out[i] = a[i] + b[i];}
}
```

Introduction to GPU

Introduction of GPU programming - vector addition – GPU device

Vector Addition Example : vector_addition()

Number of blocks : N	blockIdx.x : 0, 1, 2, 3 … N – 1
Number of threads in each block : 256	threadIdx.x : 0, 1, 2, 3 … 255



Introduction to GPU

Introduction to CUDA

CUDA programming Model – Heterogenous Computing

Host – CPU, Device - GPU

```
int main(){
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;

    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    // Allocate device memore for a
    cudaMalloc((void**)&d_a,sizeof(float)*N);
    cudaMalloc((void**)&d_b,sizeof(float)*N);
    cudaMalloc((void**)&d_out,sizeof(float)*N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a,a, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b, sizeof(float)*N, cudaMemcpyHostToDevice);

    // Main function
    int block_size = 256;
    int grid_size  = (N+block_size)/block_size;
    vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);

    cudaMemcpy(out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);

    // Deallocate device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);

    // Deallocate host memory
    free(a);
    free(b);
    free(out);

}
```

```
int main(){
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;

    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f;
        b[i] = 2.0f;
    }
```

Memory Allocation in Host

```
    // Allocate device memore for a
    cudaMalloc((void**)&d_a,sizeof(float)*N);
    cudaMalloc((void**)&d_b,sizeof(float)*N);
    cudaMalloc((void**)&d_out,sizeof(float)*N);
```

Memory Allocation in Device

```
    // Transfer data from host to device memory
    cudaMemcpy(d_a,a, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b, sizeof(float)*N, cudaMemcpyHostToDevice);
```

Data transfer from Host to Device

```
    // Main function
    int block_size = 256;
    int grid_size  = (N+block_size)/block_size;
    vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);
```

Computation in Device

```
cudaMemcpy(out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);
```

Data transfer from Device to Host

```
    // Deallocate device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);

    // Deallocate host memory
    free(a);
    free(b);
    free(out);
```

Deallocation of Memory

Introduction to GPU

GPU programming syntax

- Functions that run on GPU are usually enclosed in “<<<   >>>”.
- It has extension ‘.cu’.
- Complied using nvcc compiler driver.

Introduction to GPU

GPU programming syntax

The host device CPU manages memory

- It allocates the memory in the GPU
  - cudaMalloc (void **pointer, size_t nbytes)
  - cudaMemset (void *point, int value, size_t count)
- Data is transferred from host memory to device memory
  - cudaMemcpy(device_variable, host_variable, size_of_variable, CudaMemcpyHosttoDevice)
- After the kernel execution and data is transferred from device to host memory
  - cudaFree (void *pointer)
- Finally, the memory is deallocated
  - cudaMemcpy(host_variable, device_variable, size_of_variable, CudaMemcpyDevicetoHost)

GPU programming syntax

EXERCISE 1 : Vector Addition using GPU program

Source code:
https://github.com/sivasanarul/FEMwithGPU/tree/master/EX1_vector_addition

Introduction to GPU

Introduction of GPU

Code profiling – nvprof ./####

1 thread

```
==28571== Profiling application: ./vector_add_onethread
==28571== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.28%  14.4019s         1  14.4019s  14.4019s  14.4019s  vector_add(float*, float*, float*, int)
                    3.14%  479.91ms         2  239.95ms  239.17ms  240.74ms  [CUDA memcpy HtoD]
                    2.58%  394.51ms         1  394.51ms  394.51ms  394.51ms  [CUDA memcpy DtoH]
```

1 block with 256 threads

```
==28894== Profiling application: ./vector_add_oneblock
==28894== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   38.34%  478.81ms         2  239.40ms  239.19ms  239.62ms  [CUDA memcpy HtoD]
                   30.86%  385.37ms         1  385.37ms  385.37ms  385.37ms  [CUDA memcpy DtoH]
                   30.81%  384.72ms         1  384.72ms  384.72ms  384.72ms  vector_add(float*, float*, float*, int)
```

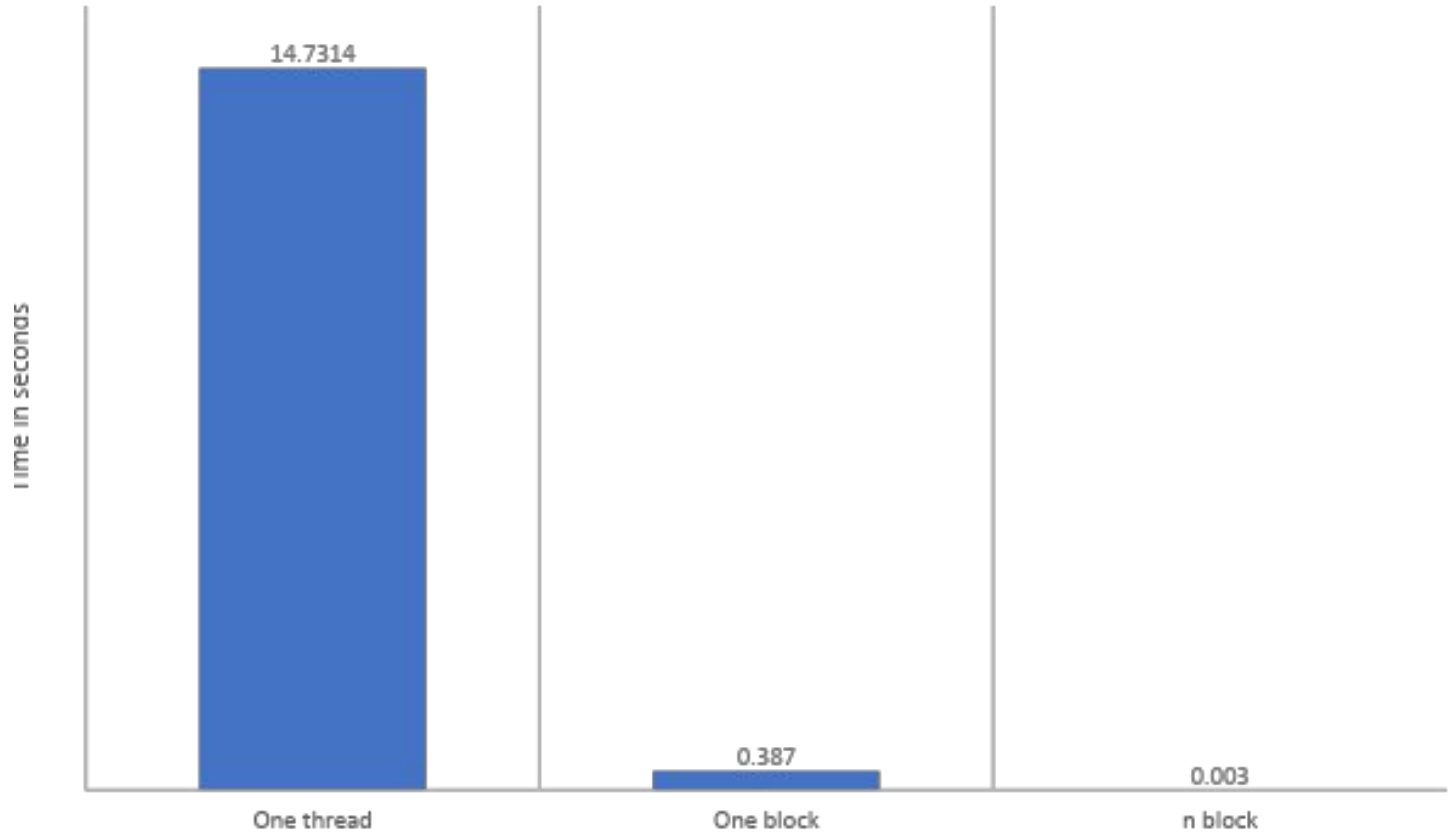
N blocks with all threads

```
==29270== Profiling application: ./vector_add_nblock
==29270== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.40%  474.87ms         2  237.43ms  237.09ms  237.77ms  [CUDA memcpy HtoD]
                   44.14%  378.35ms         1  378.35ms  378.35ms  378.35ms  [CUDA memcpy DtoH]
                    0.45%  3.8855ms         1  3.8855ms  3.8855ms  3.8855ms  vector_add(float*, float*, float*, int)
```

Introduction to GPU

Introduction of GPU programming

Vector Addition Example : vector_addition()



Introduction to GPU

Introduction of GPU

Code profiling

```
==29270== Profiling application: ./vector_add_nblock
==29270== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   55.40%  474.87ms         2  237.43ms  237.09ms  237.77ms  [CUDA memcpy HtoD]
                   44.14%  378.35ms         1  378.35ms  378.35ms  378.35ms  [CUDA memcpy DtoH]
                    0.45%  3.8855ms         1  3.8855ms  3.8855ms  3.8855ms  vector_add(float*, float*, float*, int)
```

Expensive step is the memory transfer

To circumvent:

- Perform simultaneous data transfer and computation
- Some streaming multiprocessors perform computation, some perform data transfer