SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Programming Basics

Sivasankar Arul, IT4Innovations

June/2021

Univerza v Ljubljani

TECHNISCHE UNIVERSITÄT WIEN

CINECA consorzio interuniversitario

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

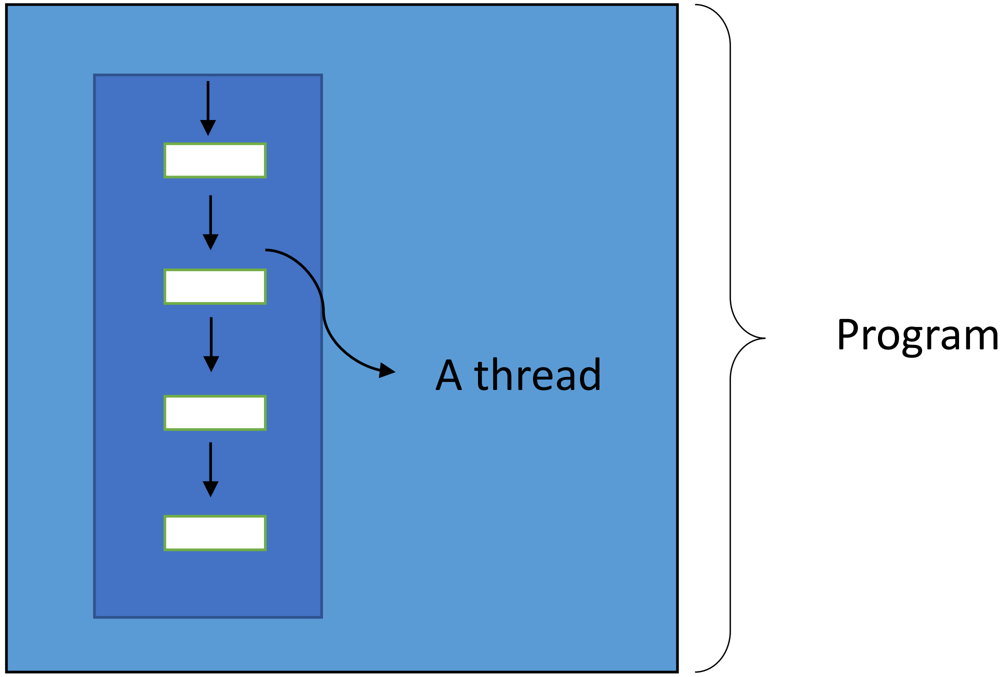
Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Thread

Thread

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



C - pointers

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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
```

Malloc()

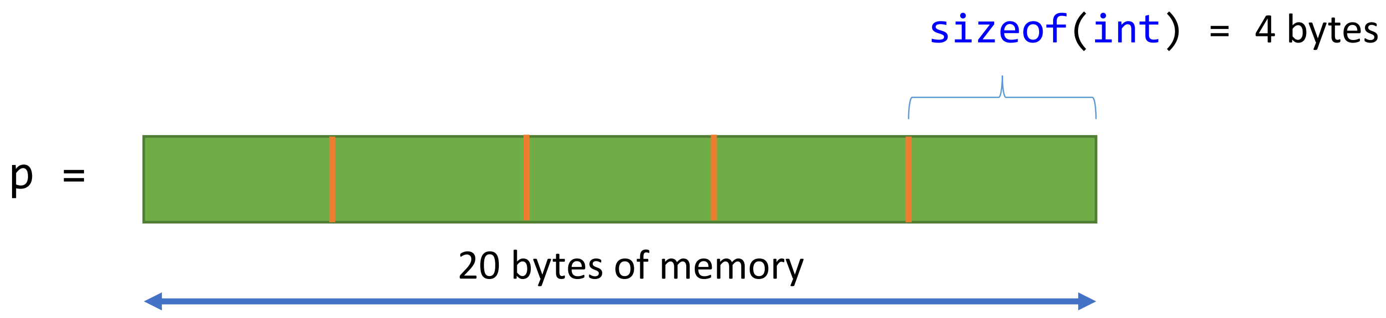
malloc()

- Dynamically allocates a single large block of memory
  - Syntax

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

  - Example

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



Vector Addition - C

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

EXERCISE 1 : Vector Addition using C program

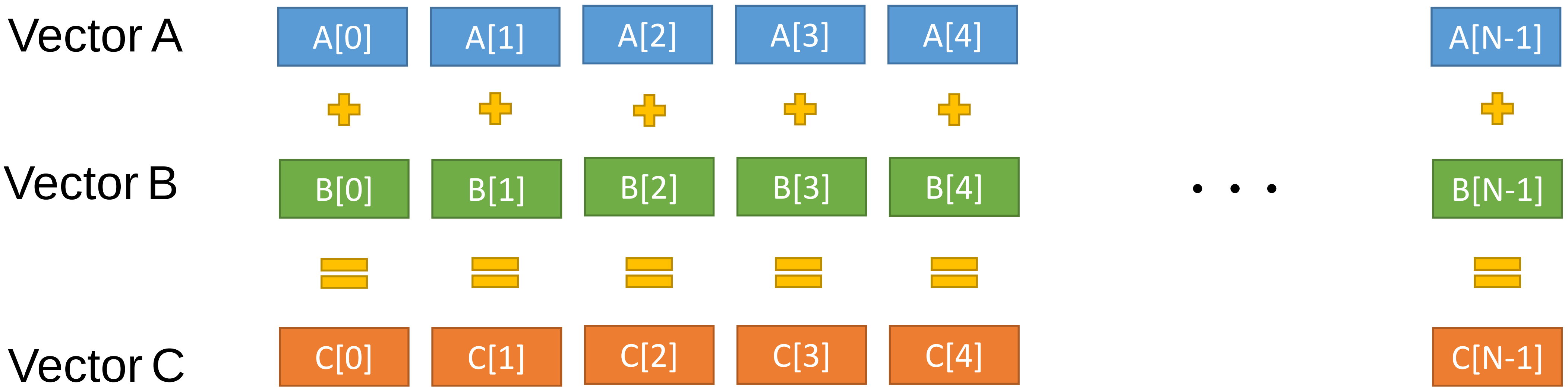
Source code

FOLDER: EX1_VECTOR_ADDITION

vector_add.c

Vector Addition

$$A + B = C$$



C program for Vector Addition

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

C programming – Vector addition

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

The “include” tells the pre-processor to include the content of the named header file.

⬇

```
#define array_size 10000000
```

Define size of the array as global variable

C program for Vector Addition

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
int main(){


}
```

The main body of the code.

⬇

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

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

Memory Allocation for the variables

C program for Vector Addition

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

Initializing the variables

⬇

```c
for(int i=0; i < array_size; i++){
    c[i] = a[i] + b[i];
    }
```

Addition of vectors

C program for Vector Addition

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

Deallocation of Memory

```
clock_t  t;
t = clock()
                   .
                   .
                   .
t = clock() – t;
double time_taken =
((double)t)/CLOCKS_PER_SEC;
```

Measuring time

Slurm - Running programs

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#!/bin/bash
#SBATCH --job-name=test
#SBATCH --output=res1.txt
#SBATCH --ntasks=1

#SBATCH --time=03:00
#SBATCH --partition=gpu
#SBATCH --nodelist=gpu01

module purge
module load icc
module load CUDA

# Operations
echo "Job start“
./matvec_onethread
# Operations
echo "Job end"
```

- To run the compiled code “matvec_onethread”

Create the file by the name: submit.sh

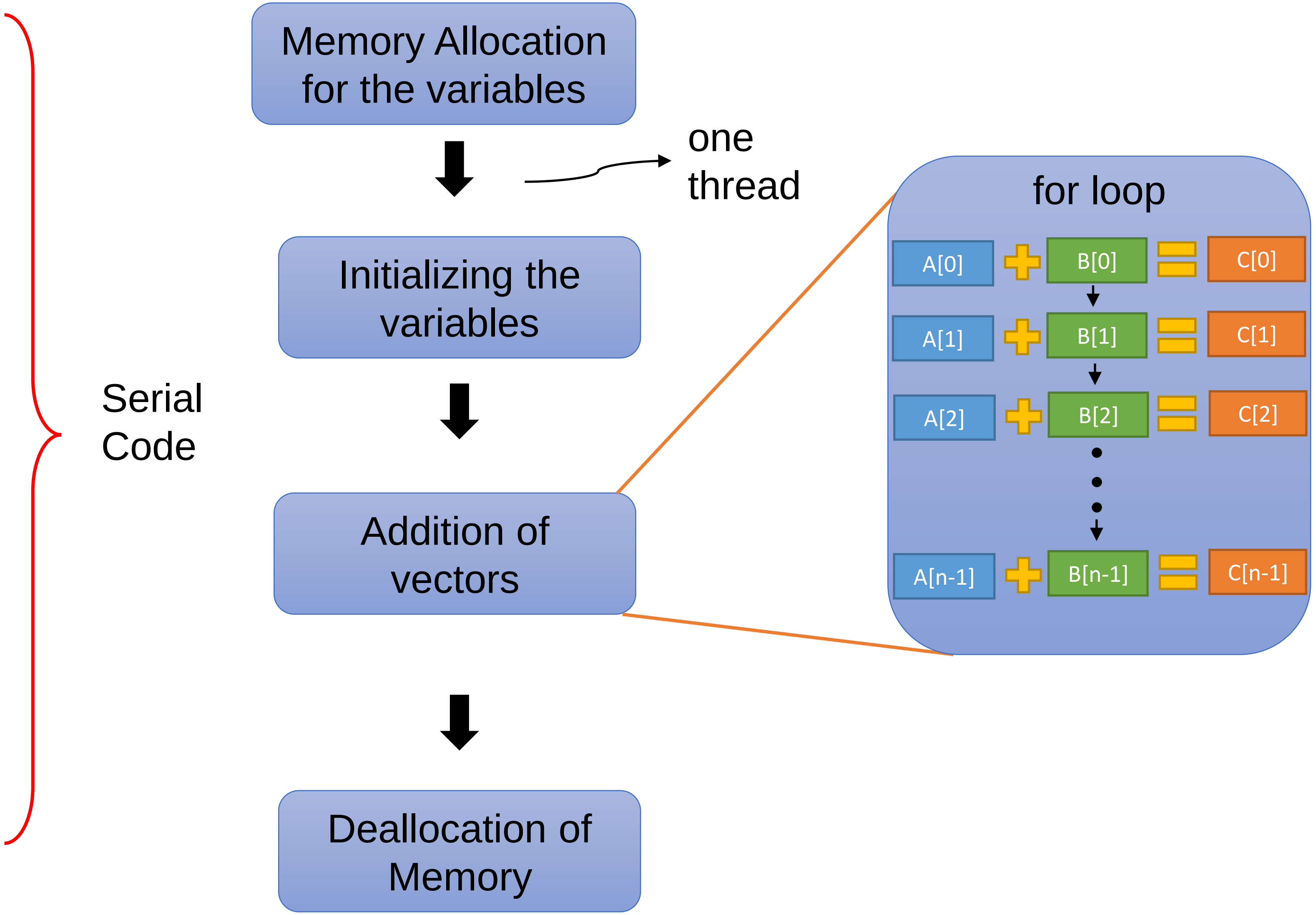
Command to launch: sbatch submit.sh

The output from the file is stored in “res1.txt”

This file launches a slot for 3 minutes in the core with gpu.

C program for Vector Addition


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

int main(){

    float *a, *b, *c;

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

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

    clock_t t;
    t = clock();

    // vector addition
    for(int i = 0;i < array_size; 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);

}
```



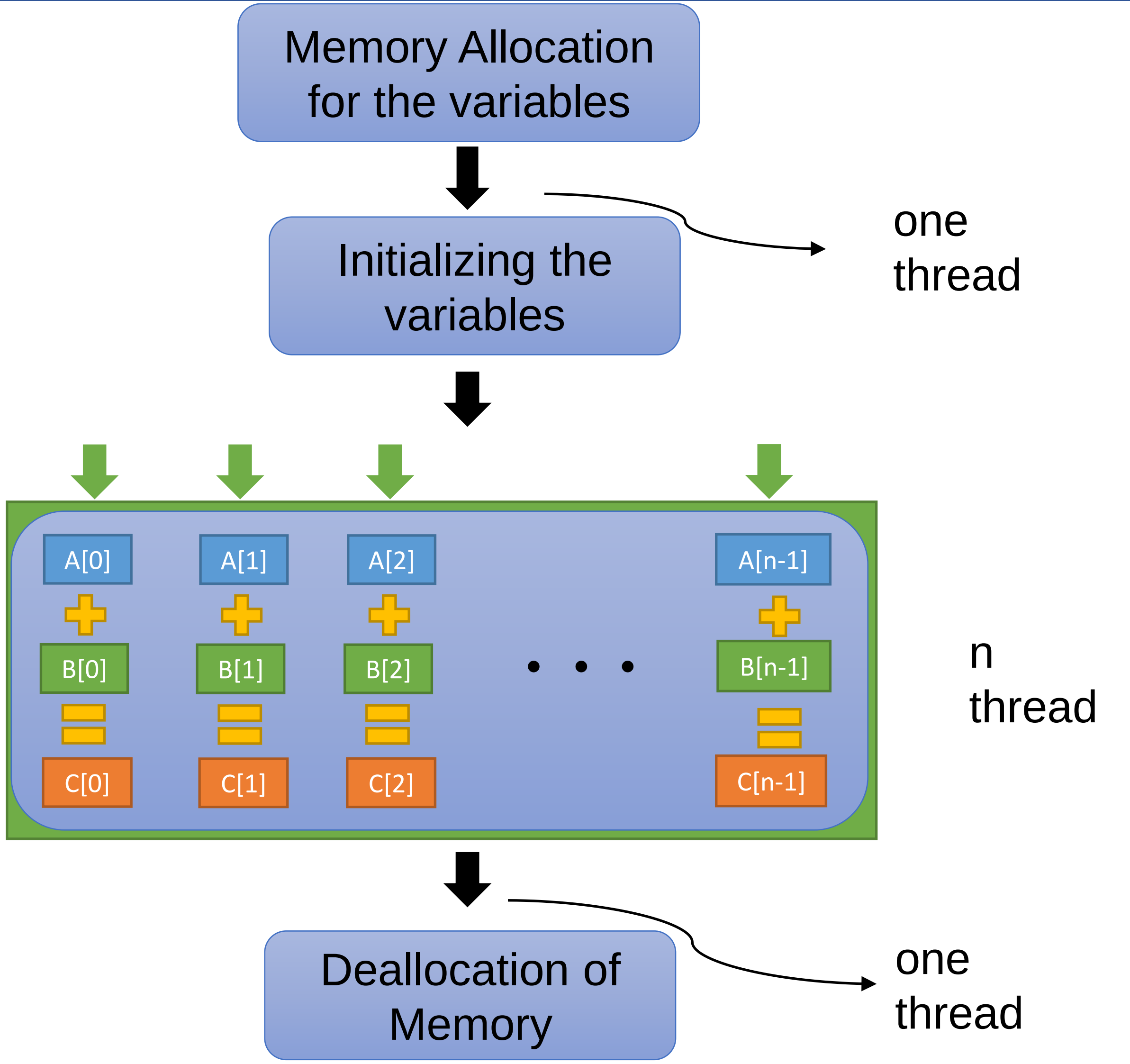
Heterogenous Program


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

    // Allocate host memory
    a   = (float*)malloc(sizeof(float) * array_size);
    b   = (float*)malloc(sizeof(float) * array_size);
    out = (float*)malloc(sizeof(float) * array_size);

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

    // Allocate device memory
    cudaMalloc((void**)&d_a, sizeof(float)*array_size);
    cudaMalloc((void**)&d_b, sizeof(float)*array_size);
    cudaMalloc((void**)&d_out, sizeof(float)*array_size);

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

    int block_size = 256;
    int grid_size  = (array_size + block_size) / block_size;
    // Vector addition
    vector_add<<<grid_size, block_size>>>(d_out, d_a, d_b, array_size);

    // Transfer data from device to host memory
    cudaMemcpy(out, d_out, sizeof(float)*array_size, cudaMemcpyDeviceToHost);

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

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

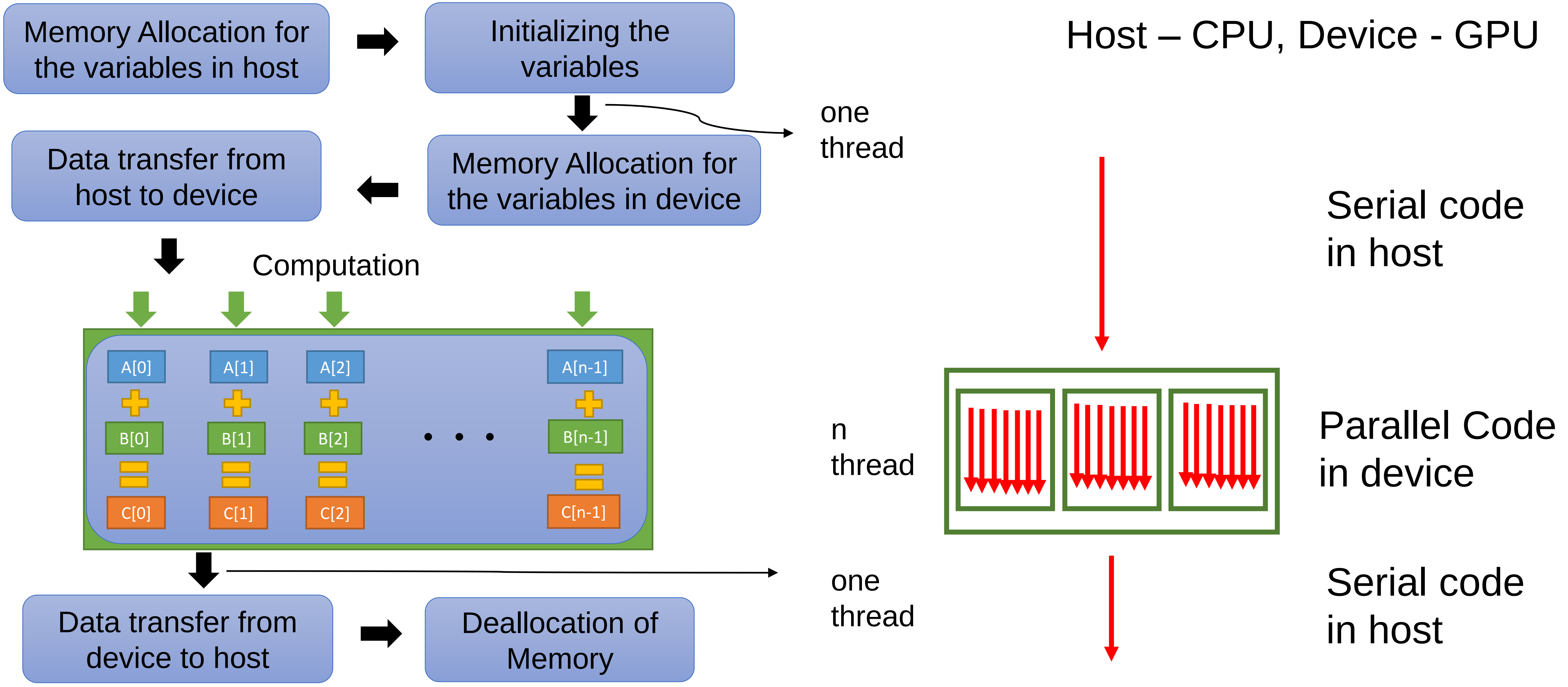
Serial Code

Parallel Code

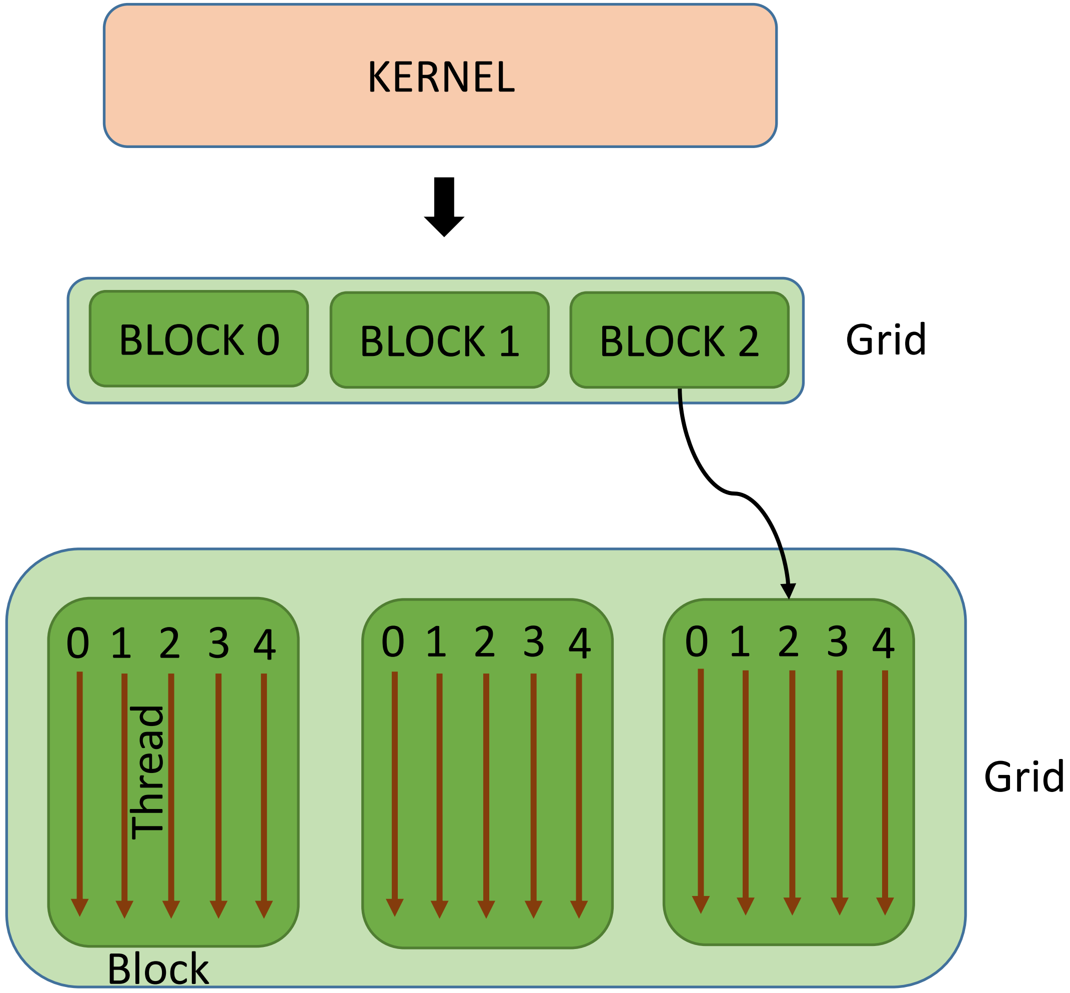
Serial Code



Heterogenous Program



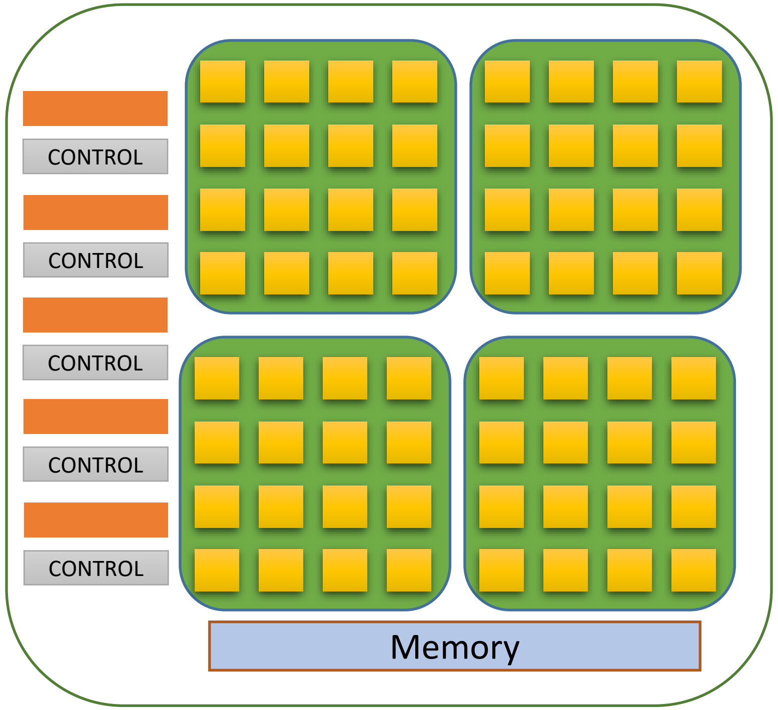
GPU Architecture

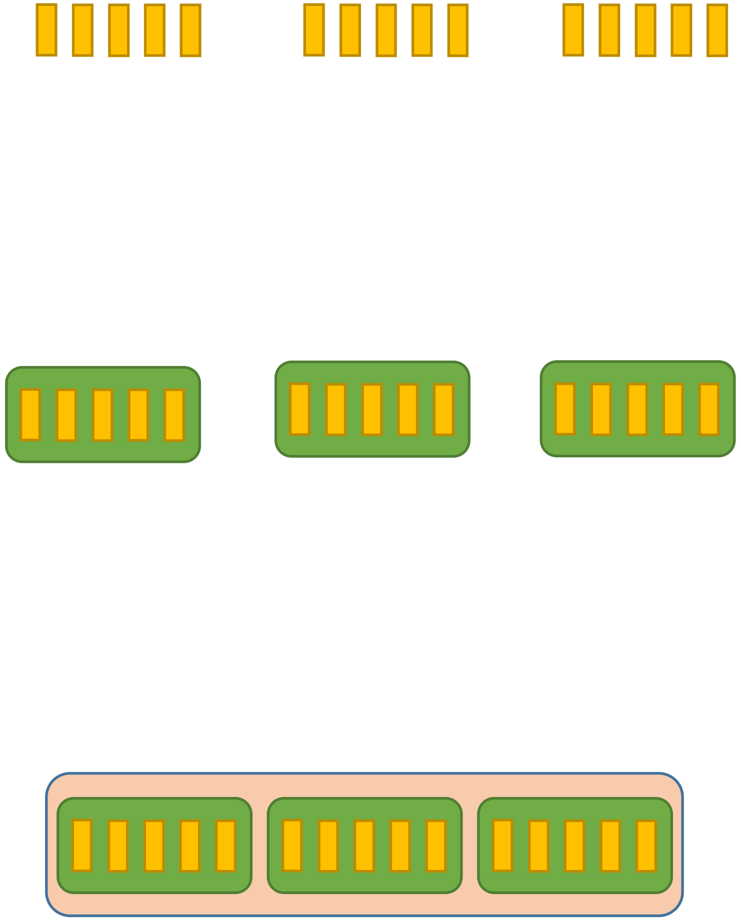


- A kernel is executed as a grid
- A grid is broken into blocks
- Each block is broken into threads

GPU Architecture

GPU device



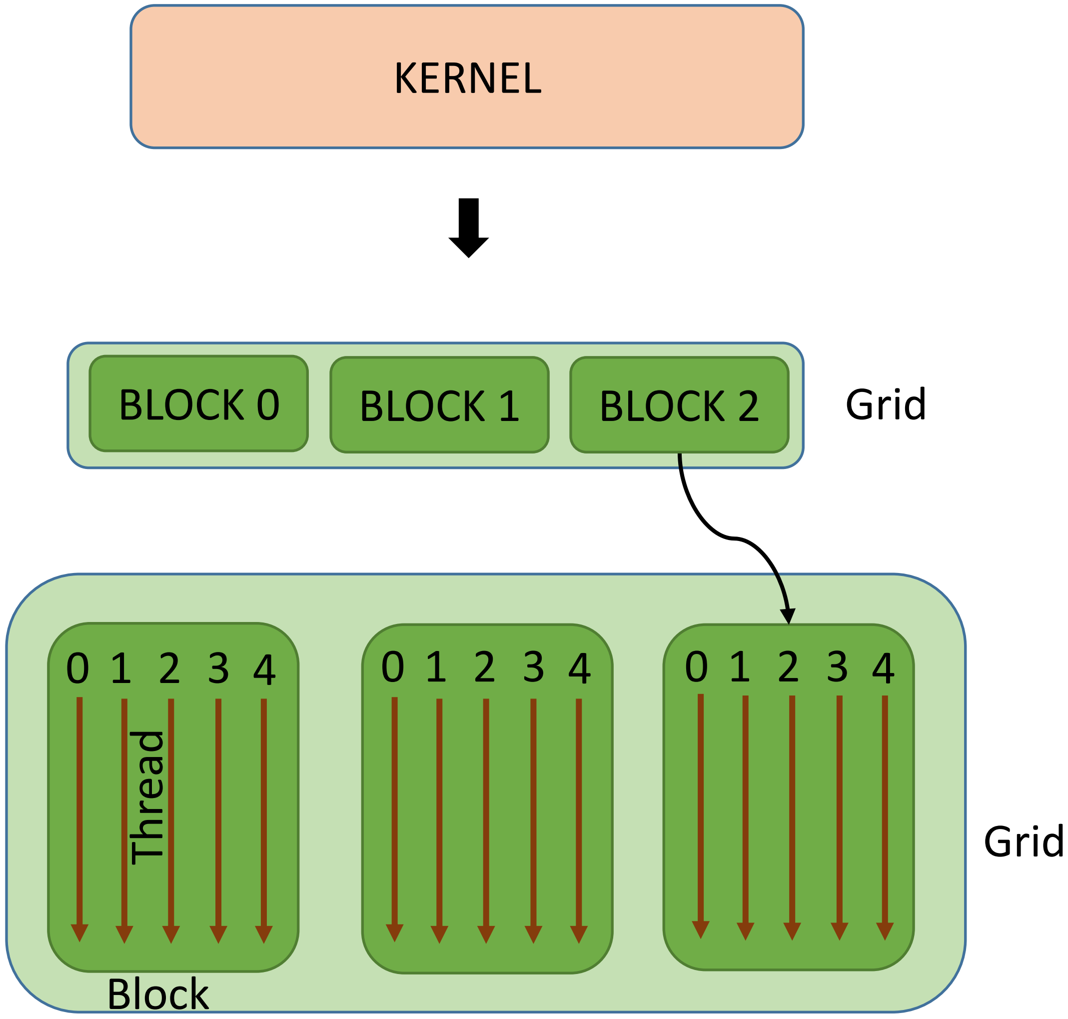


cores

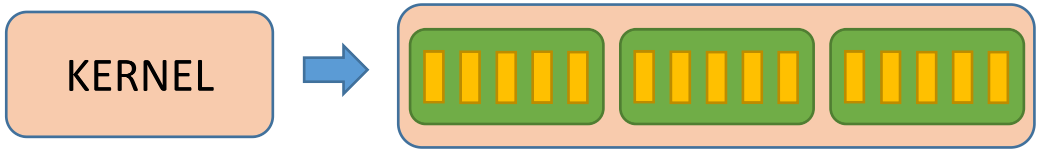
Streaming Multiprocessor – collection of cores

GPU – Collection of Streaming Multiprocessor

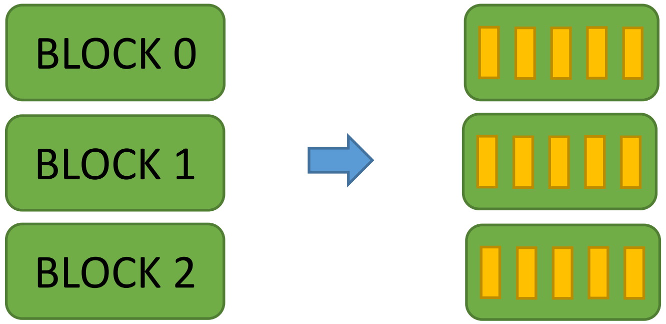
GPU Architecture



A kernel is executed in a CUDA-enabled GPU

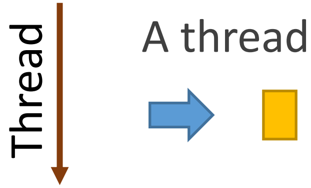


One block is executed in one Streaming Multiprocessor.
The three blocks are executed in parallel



- Depending on the number of SM, blocks are distributed and executed in parallel
- More SM a device has, faster is the execution

A thread is executed in a core



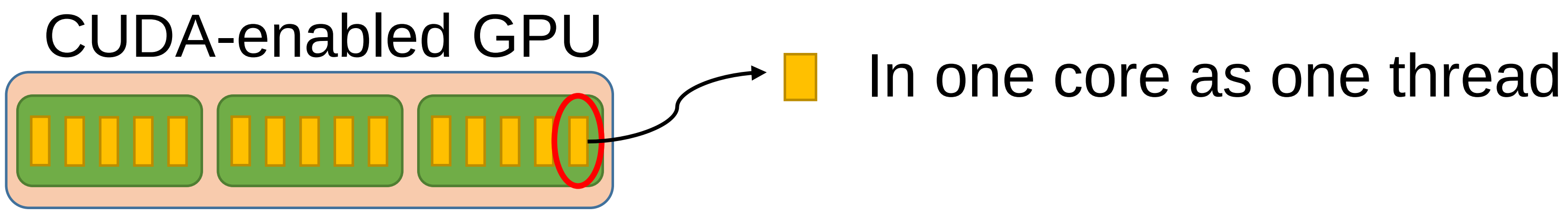
Vector Addition - GPU

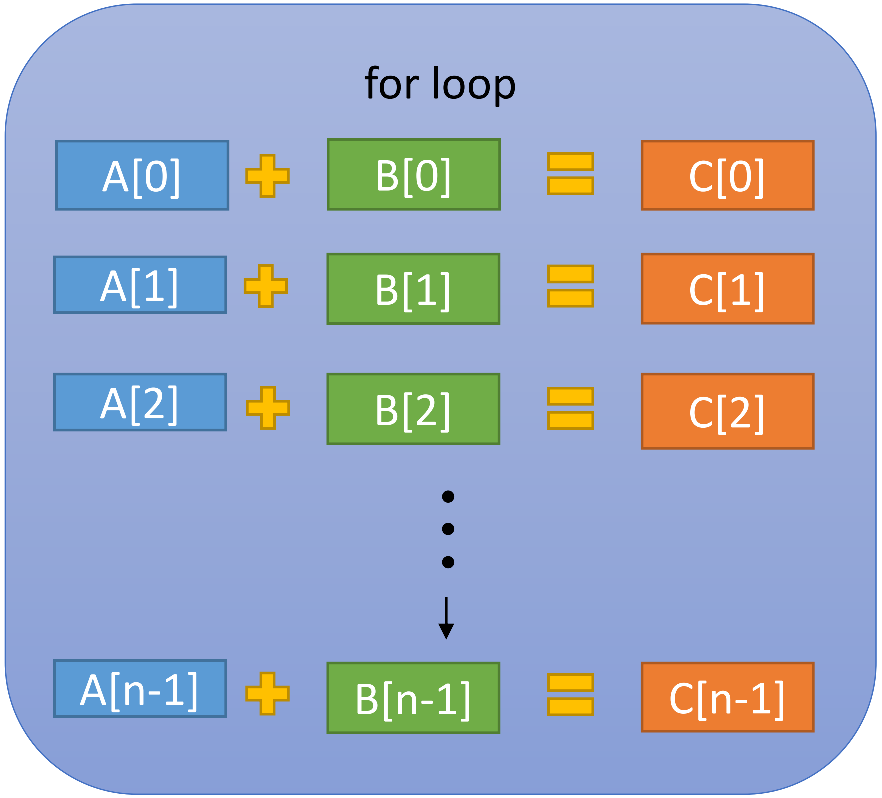
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Vector Addition using GPU

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

Vector Addition – One core

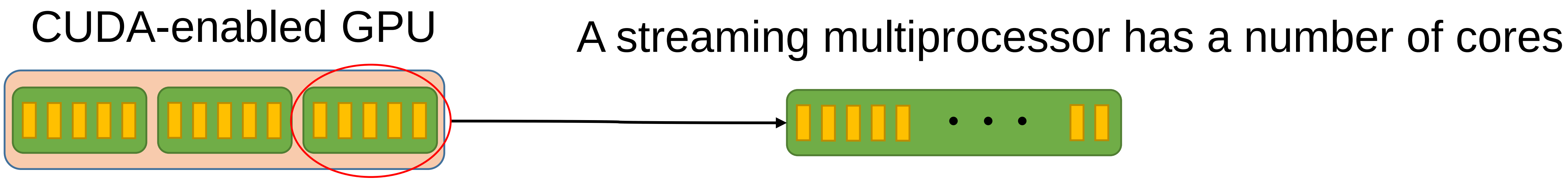





```
__global__ void vector_add(float *out, float *a, float *b, int n){
for(int i = 0; i < n; i++){
    out[i] = a[i] + b[i];}
}
```

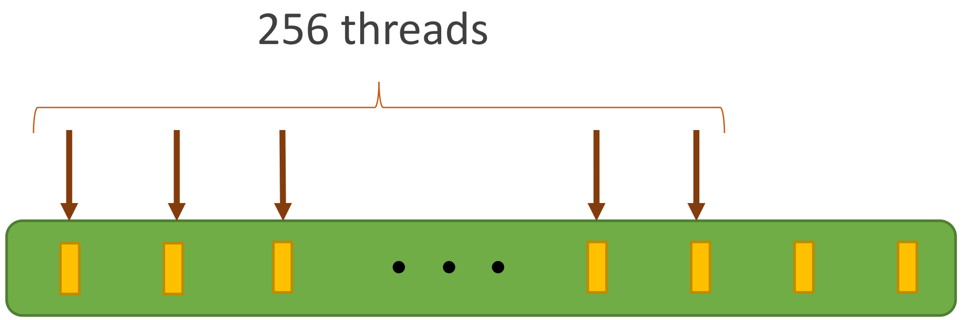
Vector Addition – One SM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP



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)
```




```
__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];}
}
```

Vector Addition – One SM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- Each thread performs the vector addition on a certain chunk of the array.

- **Strategy for distributing the array between the threads**
  - **Requisite**
    1. The threads should not communicate with each other.
    2. The array should be equally split between the threads.
  - **Constraint**
    1. Each thread will run the same function.

- The array is in the device memory.
- It is available for all the threads.

What's available?
Each thread can have its local variables.
We define two local variables:

1. It has an unique id : threadIdx.x
2. The number of threads : blockDim.x

Vector Addition – One SM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
__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];}
}
```

`thread 0`

Local variables for this thread:

```
index  = threadIdx.x = 0
stride = blockDim.x  = 256
```

For loop

first loop:

```
i = index = 0
out[i=0] = a[i=0] + b[i=0]
```

second loop:

```
i = i + stride = 256
out[i=256] = a[i=256] + b[i=256]
```

third loop:

```
i = i + stride = 512
out[i=512] = a[i=512] + b[i = 512]
```

until: `i < n`

`thread 1`

Local variables for this thread:

```
index  = threadIdx.x = 1
stride = blockDim.x  = 256
```

For loop

first loop:

```
i = index = 1
out[i=1] = a[i=1] + b[i=1]
```

second loop:

```
i = i + stride = 257
out[i=257] = a[i=257] + b[i=257]
```

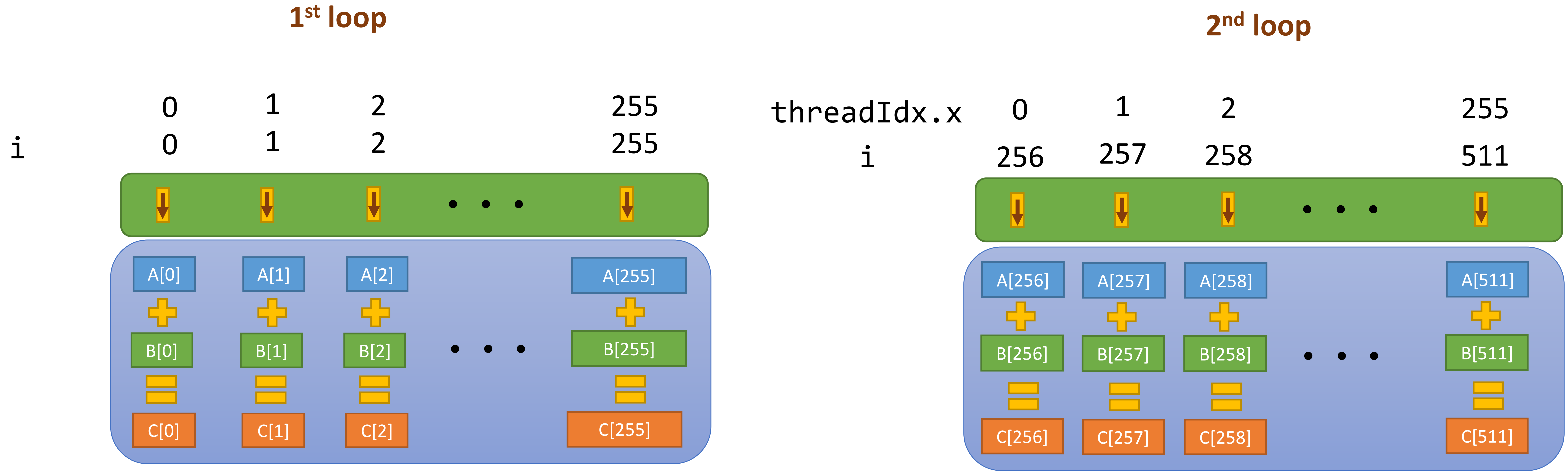
third loop:

```
i = i + stride = 513
out[i=513] = a[i=513] + b[i = 513]
```

until: `i < n`

Vector Addition – One SM

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

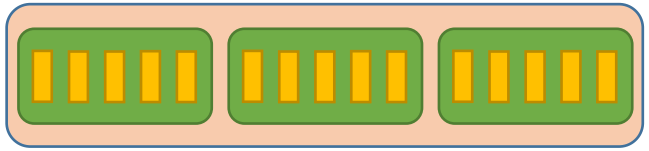



```
__global__ void vector_add(float *out, float *a, float *b, int n){
int index  = threadIdx.x;
int stride = blockDim.x;

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

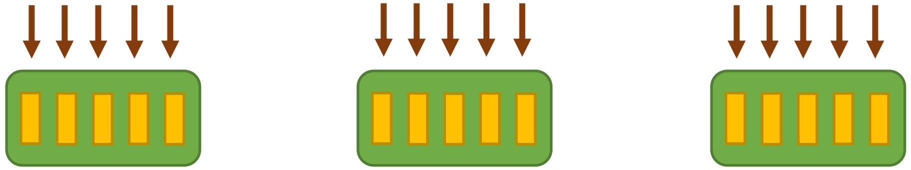
Vector Addition – Multiple SMs

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

CUDA-enabled GPU



Many streaming multiprocessors can be used



- Each thread accesses one element in the array. We predefine the number of threads in a block.
- The number of blocks is calculated based on the array size and the number of threads in a block.

$$\text{number of blocks } n = \text{array size} / \text{number of threads in each block}$$

Vector Addition – Multiple SMs

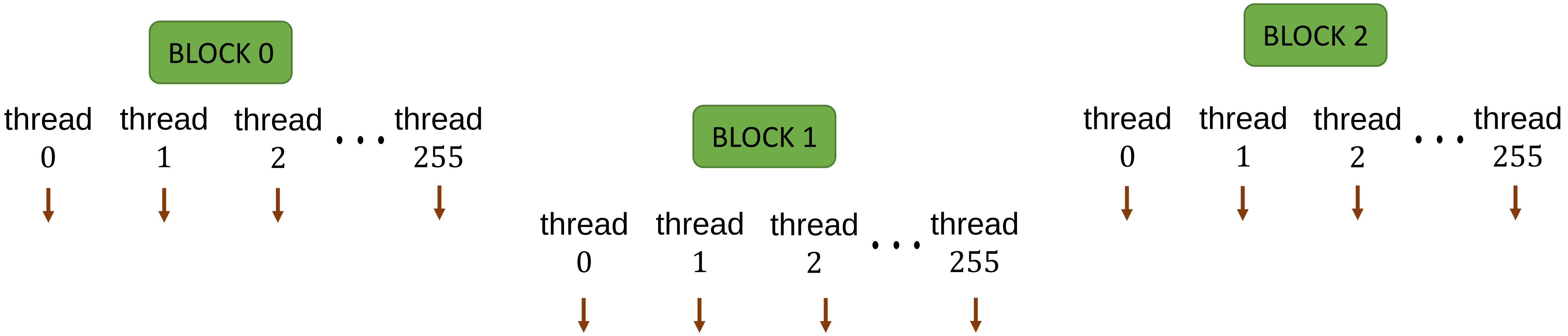
SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

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

The above command instantiates n blocks with 256 threads in each block.

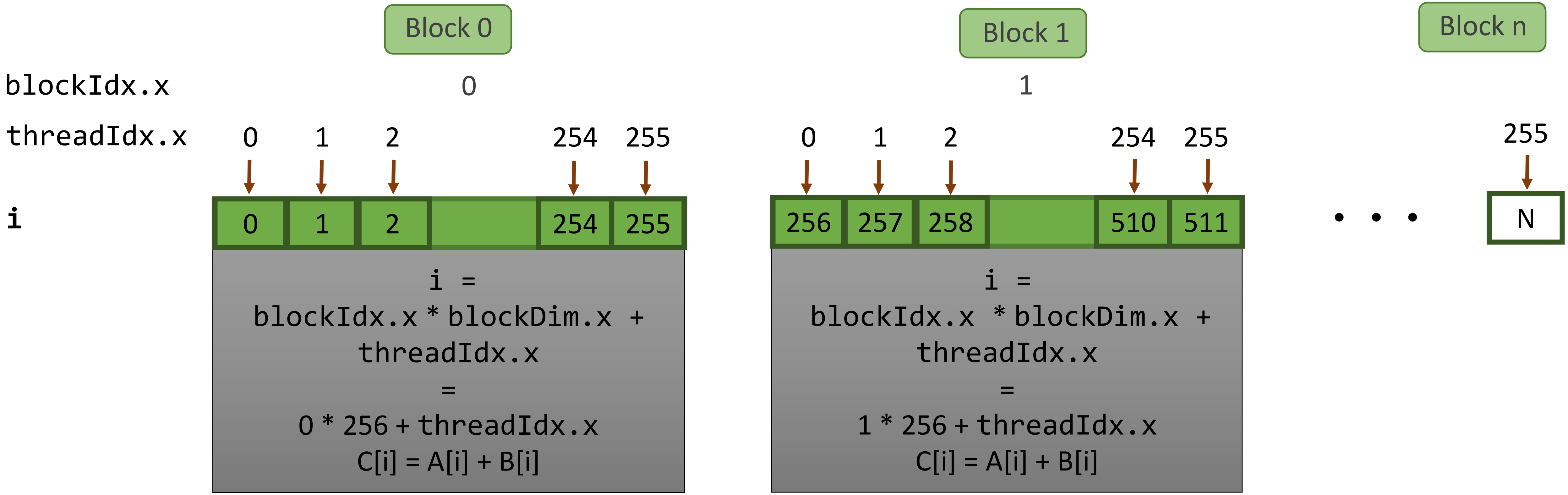
Each thread in a block has an unique id starting from 0.
Each block has an unique id starting from 0.



Vector Addition – Multiple SMs

Each thread can have its local variables. We define three local variables:

1. The id of the thread : `threadIdx.x`
2. The number of threads in the block : `blockDim.x = 256`
3. The block to which the thread belongs to : `blockIdx.x`

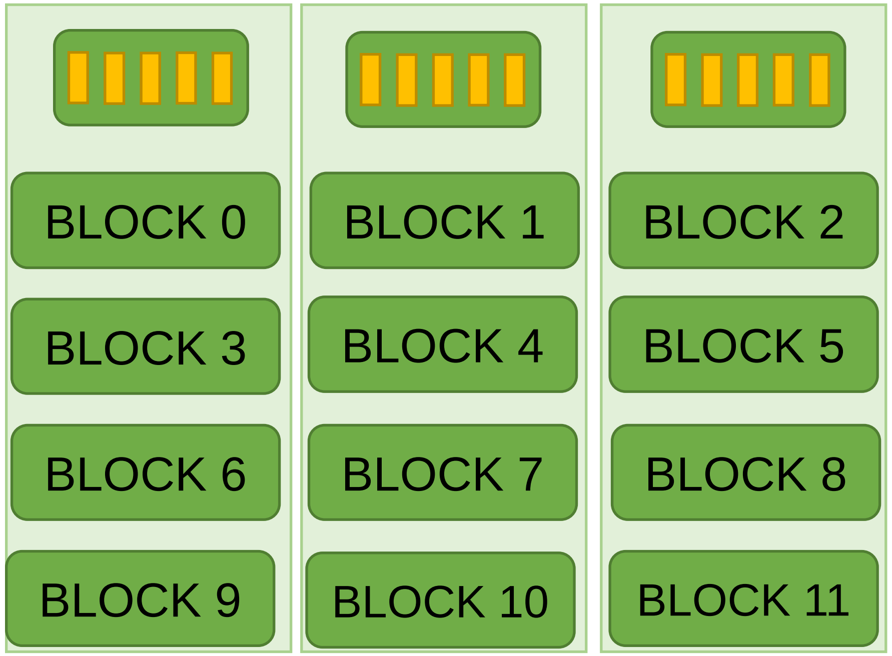


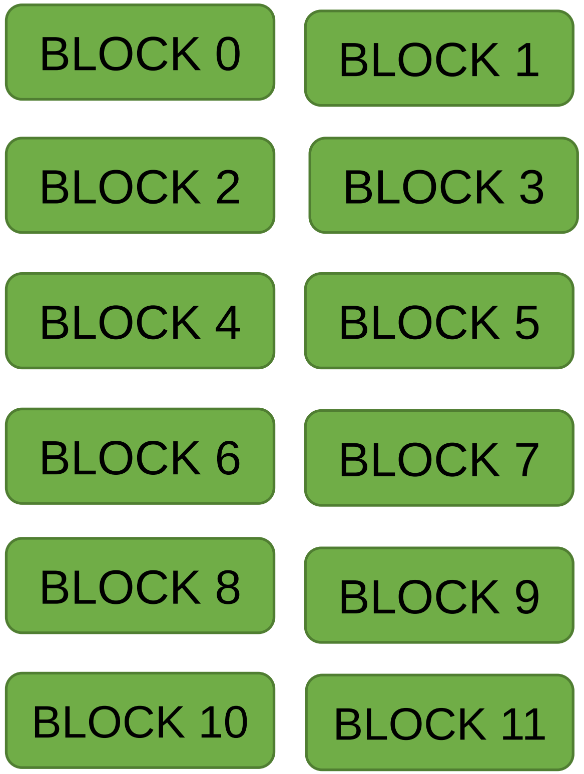
Vector Addition – Multiple SMs

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

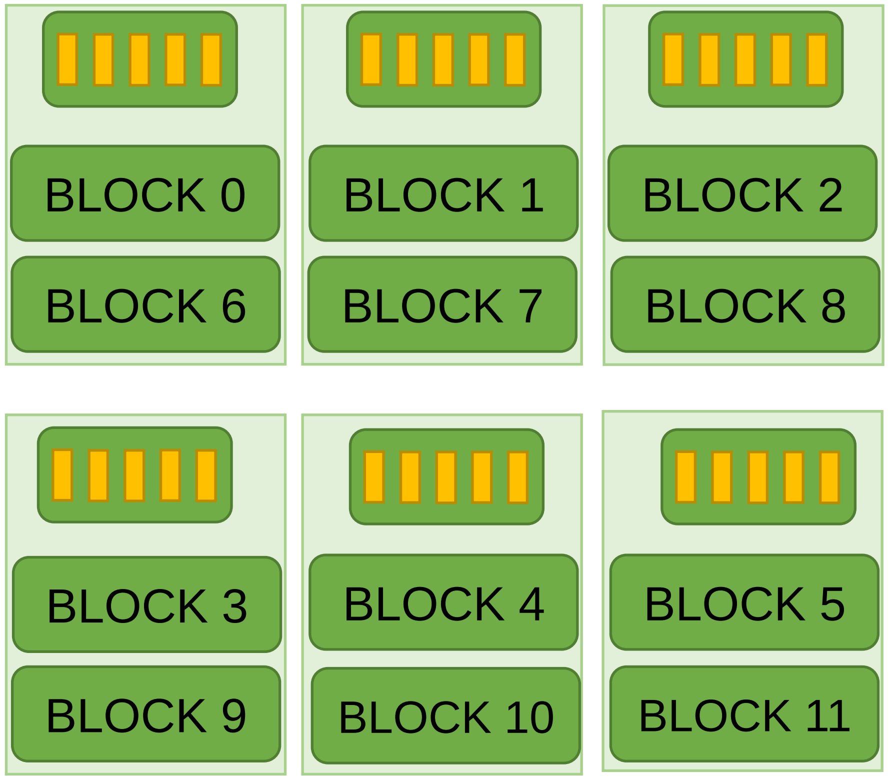
The blocks are distributed among the streaming multiprocessors

When 3 streaming multiprocessors are available





When 6 streaming multiprocessors are available



GPU Programming Syntax

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

- Functions that run on GPU are usually enclosed in “<<<   >>>”.
- The file has extension “.cu”.
- It is complied using nvcc compiler driver.

Heterogenous Program

```
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

GPU Programming Functions

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

The CPU manages both device and host memory

- Allocate the memory in the CPU

  (type*) malloc(byte – size)

- Allocate the memory in the GPU

  cudaMalloc((void**) pointer,malloc(byte - size)

- 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

  cudaMemcpy(host_variable, device_variable, size of variable, CudaMemcpyDevicetoHost)

- The memory in GPU is deallocated

  cudaFree(pointer)

- Finally, the memory in CPU is deallocated

  free(pointer)

EXERCISE 1 : Vector Addition using GPU program

Source code

FOLDER: EX1_VECTOR_ADDITION

Vector Addition – GPU program

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Host – CPU, Device - GPU

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <cuda.h>
#include <sys/time.h>
```

The “include” statement to tell the pre-processor to include the content of the named header file.

⬇

```
#define array_size 268435456
```

Define size of the array as global variable

Vector Addition – GPU program

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Host – CPU, Device - GPU

```
float *a, *b, *out;
float *d_a, *d_b, *d_out;

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

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

Memory Allocation in Host and initialization of data

Vector Addition – GPU program

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Host – CPU, Device - GPU

```
// Allocate device memory for variables
cudaMalloc((void**)&d_a,   sizeof(float) * array_size);
cudaMalloc((void**)&d_b,   sizeof(float) * array_size);
cudaMalloc((void**)&d_out, sizeof(float) * array_size);
```

Memory Allocation in Device

Vector Addition – GPU program

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Host – CPU, Device - GPU

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

Data transfer from Host to Device

Vector Addition – GPU program

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

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

Data transfer from Device to Host

Host – CPU, Device - GPU

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

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

Deallocation of Memory

One thread

Host – CPU, Device - GPU

Computation in Device

Kernel

```
__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];}
}
```

```
// Main function
int block_size = 1;
int grid_size  = 1;
vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);
cudaDeviceSynchronize();
```

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

One block

Host – CPU, Device - GPU

Computation in Device

Kernel

```
__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];}
}
```

```
// Main function
int block_size = 256;
int grid_size  = 1;
vector_add<<<grid_size,block_size>>>(d_out, d_a, d_b, N);
cudaDeviceSynchronize();
```

Vector Addition – GPU program

Multiple block

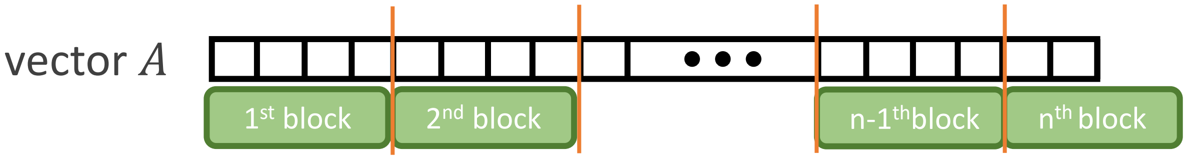
Computation in Device

Host – CPU, Device - GPU

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

```
// 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);
cudaDeviceSynchronize();
```

Remark : We ensure the tail of the array is processed by launching one extra block.



Slurm - Running programs

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
#!/bin/bash
#SBATCH --job-name=test
#SBATCH --output=res1.txt
#SBATCH --ntasks=1

#SBATCH --time=03:00
#SBATCH --partition=gpu
#SBATCH --nodelist=gpu01

module purge
module load icc
module load CUDA

# Operations
echo "Job start“
./matvec_onethread
# Operations
echo "Job end"
```

- To run the compiled code “matvec_onethread”

Create the file by the name: submit.sh

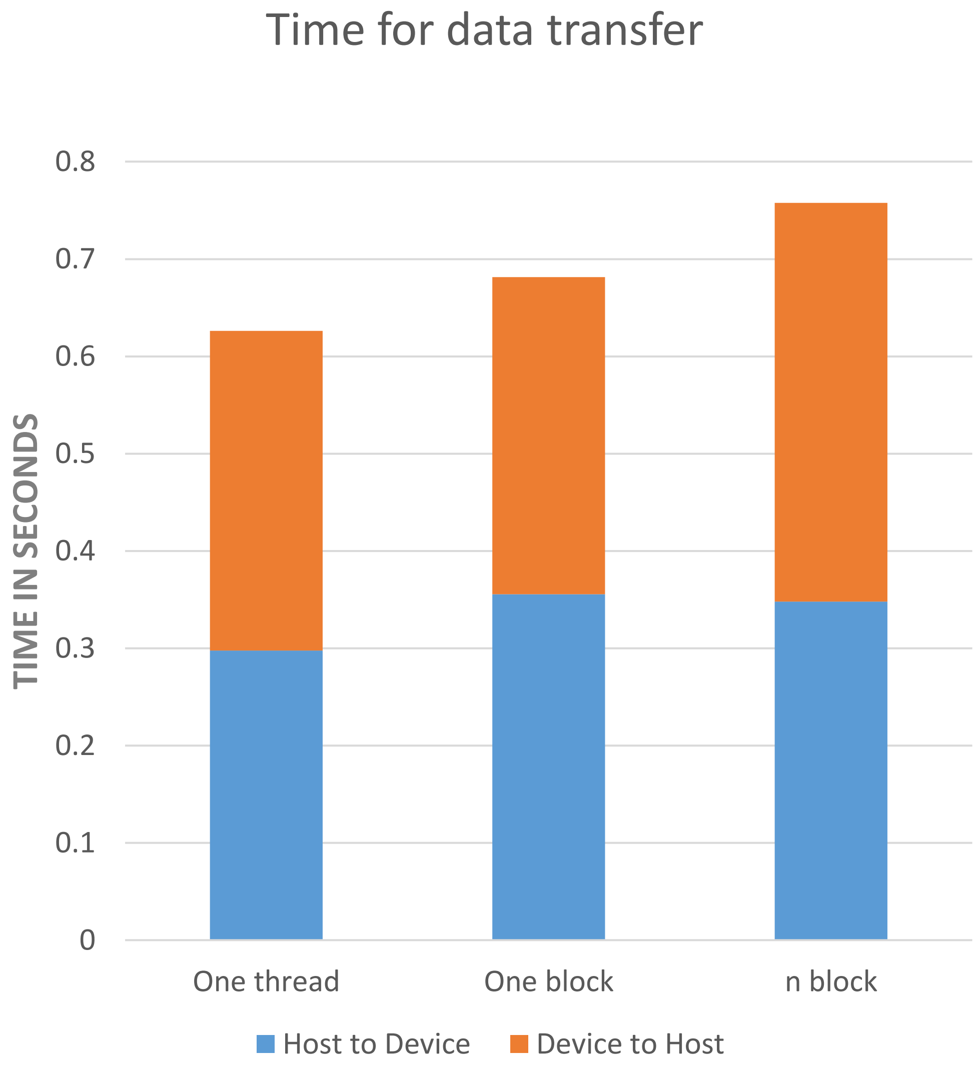
Command to launch: sbatch submit.sh

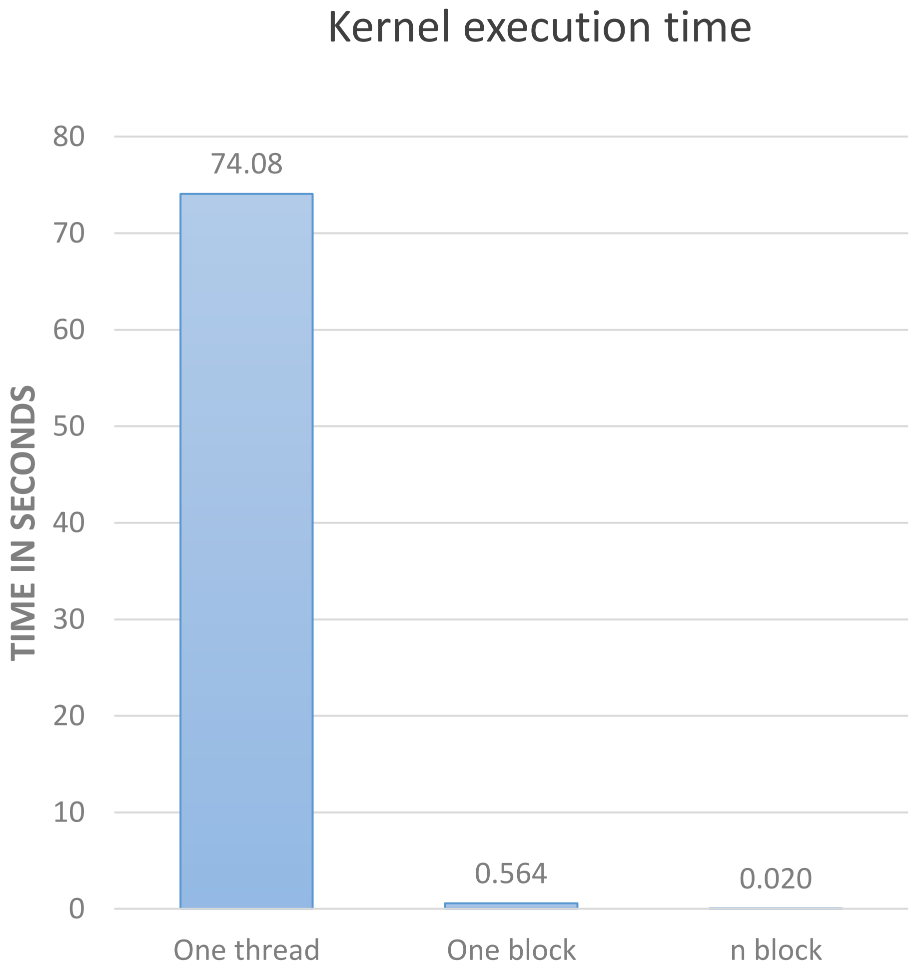
The output from the file is stored in “res1.txt”

This file launches a slot for 3 minutes in the core with gpu.

Time comparison

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP





Code Profiling

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

Code profiling – nvprof ./####

1 thread

```
==8304== Profiling application: ./vector_add_onethread
==8304== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.45%  111.230s         1  111.230s  111.230s  111.230s  vector_add(float*, float*, float*, int)
                    0.30%  335.78ms         1  335.78ms  335.78ms  335.78ms  [CUDA memcpy DtoH]
                    0.25%  283.87ms         2  141.93ms  141.51ms  142.35ms  [CUDA memcpy HtoD]
```

1 block with 256 threads

```
==8354== Profiling application: ./vector_add_oneblock
==8354== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   46.94%  695.40ms         1  695.40ms  695.40ms  695.40ms  vector_add(float*, float*, float*, int)
                   30.61%  453.48ms         2  226.74ms  226.60ms  226.87ms  [CUDA memcpy HtoD]
                   22.45%  332.56ms         1  332.56ms  332.56ms  332.56ms  [CUDA memcpy DtoH]
```

N blocks with all threads

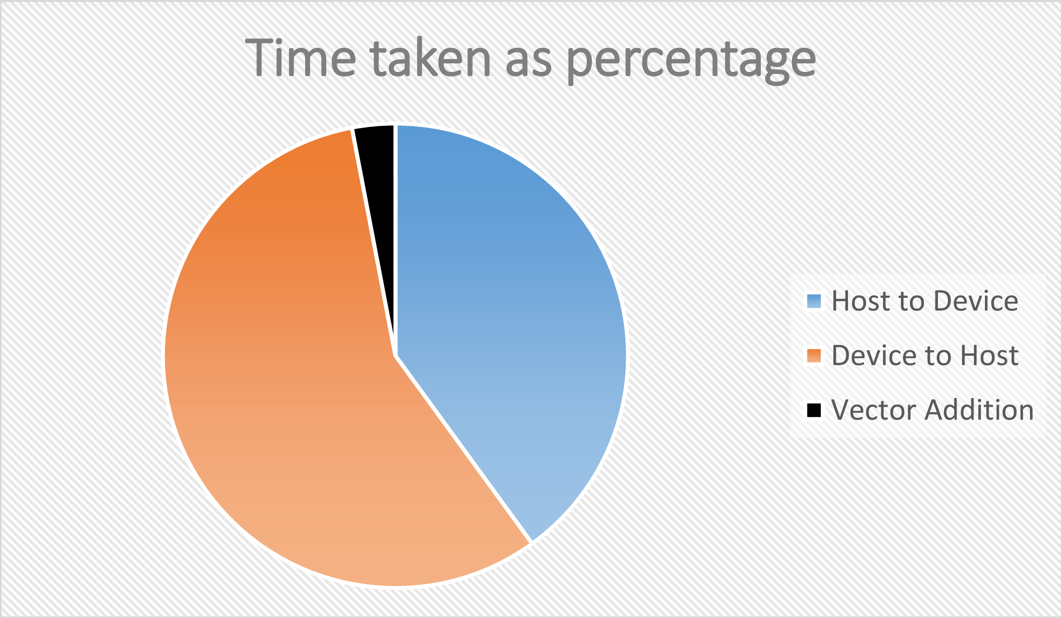
```
==11178== Profiling application: ./vector_add_nblock
==11178== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.10%  410.44ms         1  410.44ms  410.44ms  410.44ms  [CUDA memcpy DtoH]
                   39.65%  284.97ms         2  142.49ms  142.12ms  142.85ms  [CUDA memcpy HtoD]
                    3.25%  23.336ms         1  23.336ms  23.336ms  23.336ms  vector_add(float*, float*, float*, int)
```

Code Profiling

SCtrain | SUPERCOMPUTING KNOWLEDGE PARTNERSHIP

```
==11178== Profiling application: ./vector_add_nblock
==11178== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.10%  410.44ms         1  410.44ms  410.44ms  410.44ms  [CUDA memcpy DtoH]
                   39.65%  284.97ms         2  142.49ms  142.12ms  142.85ms  [CUDA memcpy HtoD]
                    3.25%  23.336ms         1  23.336ms  23.336ms  23.336ms  vector_add(float*, float*, float*, int)
```

Expensive step is the memory transfer



GPU computation

- For a task involving single computation on a data,
  - When a GPU is used most of the time will be spent on copying data between CPU and GPU memory.

- One way to circumvent this problem, if the task allows it, then:
  - ❑Perform simultaneous data transfer and computation
    - ➢Overlap computation and data transfer

- GPU is ideal when many computations needs to be done for a given data.

Thank you for your attention!

http://sctrain.eu/

Univerza v Ljubljani
TECHNISCHE UNIVERSITÄT WIEN
CINECA consorzio interuniversitario
VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

Co-funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission.
This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.