# Vector addition example

# Quick overview of CUDA implementation

- In general this is how a CUDA program works
  - Starts the process on the host (CPU)
  - Copy the data required for computation to device (GPU)
  - Performs the computation on device
  - Copy the results back from device to host

# Vector addition

- A simple example to understand CUDA basics
- Add two vectors A and B to another vector Sum
    - $\mathrm{Sum}_i = A_i + B_i$
    - $i<n$
    - n=size of the vector

# Vector addition: Serial implementation

- Let us see how it can be implemented serially in C
- Here is the code

```
/*
 * Function: add_host
 * ----------------------------
 *   Serially adds the values in vector a and b to sum
 *
 *   a: vector a
 *   b: vector b
 *   n: size of the vectors
 *   sum: vector to store results
 */
void add_host(int* a, int* b, int* sum, int n) {
        for (int i = 0; i < n; i++)
        {
                sum[i] = a[i] + b[i];
        }
}
```

# Vector addition: Serial implementation

- Here *a , *b, *sum are the pointers to the vectors defined in the main function of the program
- n represents the size of the vector (number of elements in the vector)
- Using a for loop, the program iterates from 0 to n-1,
    - In each iteration, corresponding to the index i, sums the element from vector a and b and saves in vector sum

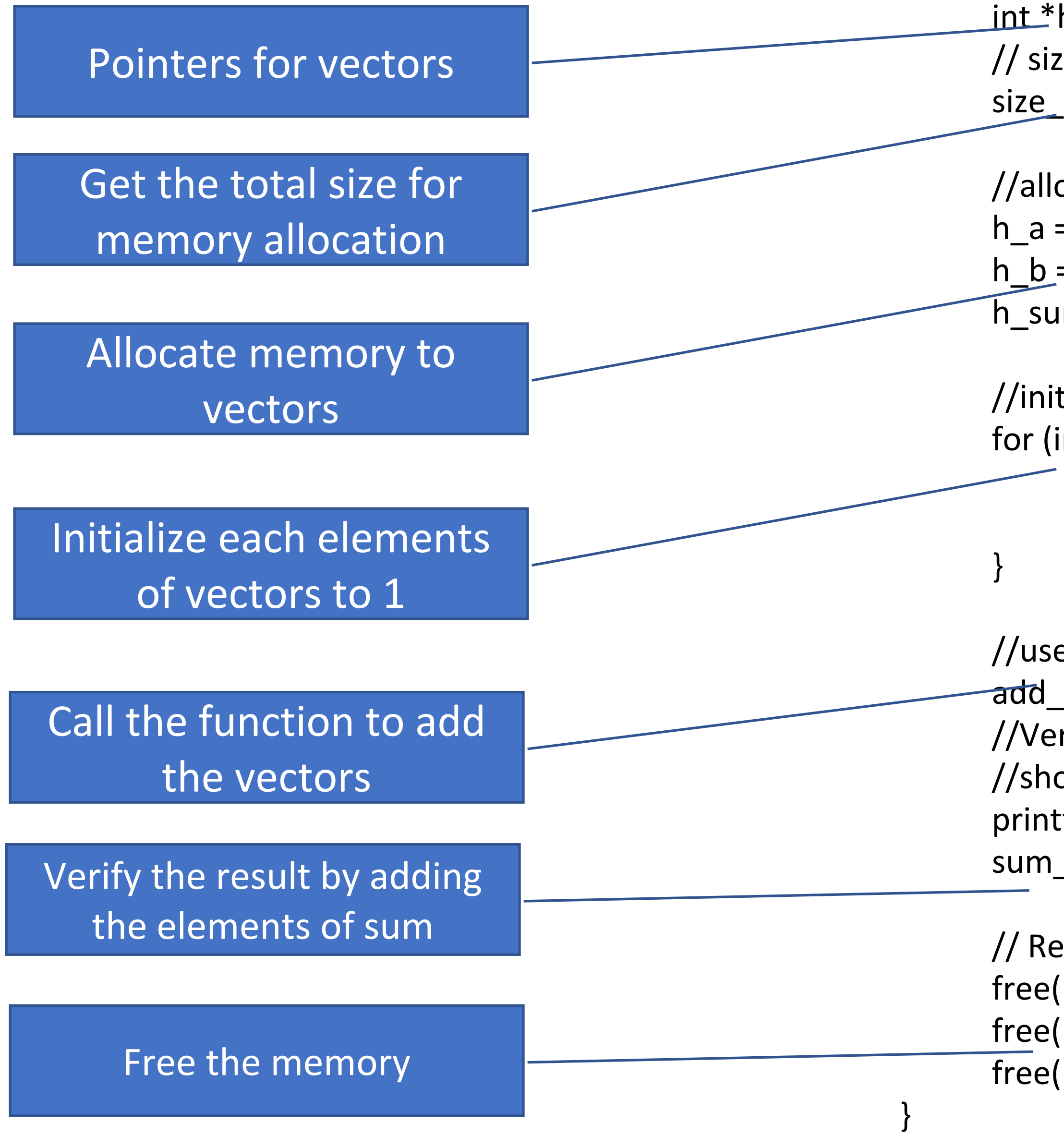
```
/*
 * Function: add_host
 * ----------------------------
 *   Serially adds the values in vector a and b
to sum
 *
 *   a: vector a
 *   b: vector b
 *   n: size of the vectors
 *   sum: vector to store results
 */
void add_host(int* a, int* b, int* sum, int n) {
        for (int i = 0; i < n; i++)
        {
                sum[i] = a[i] + b[i];
        }
}
```

# Vector addition: main function

- In the main function, we create the vectors
- SIZE is the number of elements in vector
- Initialize them
- Use the add function
- To check the program is working, we also use a function sum_vect that simply sums the result
    - Since we initialize vector a and b to all 1s, if it is correct, the sum of result should be 2 * size of vectors

```
/********************************************
 * main
 ********************************************/
int main(void) {
    //host vectors
    int *h_a, *h_b, *h_sum;
    // size of the total vectors necessary to allocate memory
    size_t size_vect = SIZE*sizeof(int);

    //allocate memory for the vectors on host (cpu)
    h_a = (int*)malloc(size_vect);
    h_b = (int*)malloc(size_vect);
    h_sum = (int*)malloc(size_vect);

    //initialize the vectors each with value 1
    for (int i = 0; i < SIZE; i++) {
        h_a[i] = 1;
        h_b[i] = 1;
    }

    //use serial function for vector addition
    add_host(h_a, h_b, h_sum,SIZE);
    //Verify the result by adding all the sum,
    //should be 2 * SIZE
    printf("Host sum:\n");
    sum_vect(h_sum);

    // Release all host memory
    free(h_a);
    free(h_b);
    free(h_sum);
}
```

# Vector addition: main function



Pointers for vectors

Get the total size for memory allocation

Allocate memory to vectors

Initialize each elements of vectors to 1

Call the function to add the vectors

Verify the result by adding the elements of sum

Free the memory

```
int main(void) {
        //host vectors
        int *h_a, *h_b, *h_sum;
        // size of the total vectors necessary to allocate memory
        size_t size_vect = SIZE*sizeof(int);

        //allocate memory for the vectors on host (cpu)
        h_a = (int*)malloc(size_vect);
        h_b = (int*)malloc(size_vect);
        h_sum = (int*)malloc(size_vect);

        //initialize the vectors each with value 1
        for (int i = 0; i < SIZE; i++) {
                h_a[i] = 1;
                h_b[i] = 1;
        }

        //use serial function for vector addition
        add_host(h_a, h_b, h_sum);
        //Verify the result by adding all the sum,
        //should be 2 * SIZE
        printf("Host sum:\n");
        sum_vect(h_sum);

        // Release all host memory
        free(h_a);
        free(h_b);
        free(h_sum);
}
```

# Function to sum vector elements and print

```
/*
 * Function: sum_vect
 * ----------------------------
 *   Adds and prints all the elements in vector vect for validation
 *
 *   vect: vector
 */
void sum_vect(int* vect)
{
        int total = 0;
        //sum
        for (int i = 0; i < SIZE; i++)
        {
                total += vect[i];
        }
        //print the result
        printf("%d \n", total);
}
```

# Vector addition: CUDA implementation

- Now let us see the kernel for vector addition in CUDA

```
*
 * Kernel - Add vectors
 * ----------------------------
 *   Each thread adds the values from vector a and b to sum
 *      corresponding to the thread index
 *
 *   a: vector a
 *   b: vector b
 *   sum: vector to store results
 */
__global__ void add_device(int* a, int* b, int* sum, int n) {
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if (thread_id < n)
                sum[thread_id] = a[thread_id] + b[thread_id];
}
```

# Vector addition: CUDA implementation

- Here *a , *b, *sum are the pointers to the vectors defined in the main function of the program
- n represents the size of the vector (number of elements in the vector)
- __global__  represents that the function is run on the device (is called from the host)
- thread_id is the global id of the thread within the block
- Corresponding to their id, each of the thread will add the elements from vector a and b and store in vector sum
    - In case, there are more threads than the size of vector, we use a if condition

```
*
 * Kernel - Add vectors
 * ----------------------------
 *   Each thread adds the values from vector a and b to sum
 *          corresponding to the thread index
 *
 *   a: vector a
 *   b: vector b
 *   sum: vector to store results
 */
__global__ void add_device(int* a, int* b, int* sum, int n) {
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if (thread_id < n)
                sum[thread_id] = a[thread_id] + b[thread_id];
}
```

# Vector addition: CUDA implementation

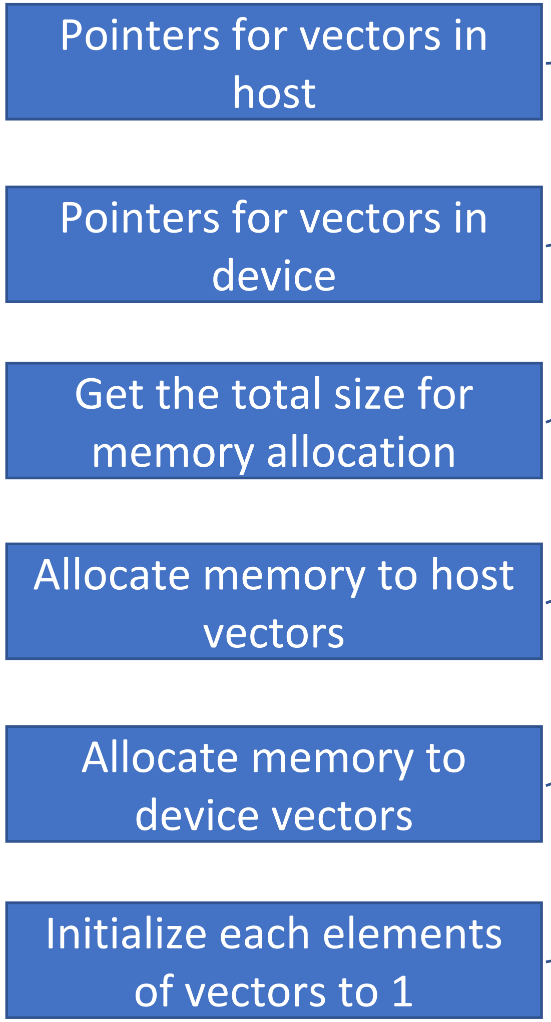
- Here *a , *b, *sum are the pointers to the vectors defined in the main function of the program
- n represents the size of the vector (number of elements in the vector)
- __global__  represents that the function is run on the device (is called from the host)
- thread_id is the global id of the thread within the block

```
*
 * Kernel - Add vectors
 * ----------------------------
 *   Each thread adds the values from vector a and b to sum
 *          corresponding to the thread index
 *
 *   a: vector a
 *   b: vector b
 *   sum: vector to store results
 */
__global__ void add_device(int* a, int* b, int* sum, int n) {
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if (thread_id < n)
                sum[thread_id] = a[thread_id] + b[thread_id];
}
```

# Vector addition: CUDA implementation

- In general this is how a CUDA program works for the vector addition
  - Starts the process on the host (CPU)
    - Create vectors for use in host
      - Allocate memory for the vectors using malloc
    - Create new vectors for use in device
      - Create memory for the vectors using cudaMalloc
    - Initialize the vectors
  - Copy the data required for computation to device (GPU) using cudaMemcpy
  - Execute the kernel with block size and number of threads in each block
  - Performs the computation on device
    - Each of the threads execute the kernel
      - Each of the thread adds the vector elements based on their thread id
  - Copy the results back from device to host
  - Complete other process in host
  - Free allocated memory

# CUDA implementation: main function



Pointers for vectors in host

Pointers for vectors in device

Get the total size for memory allocation

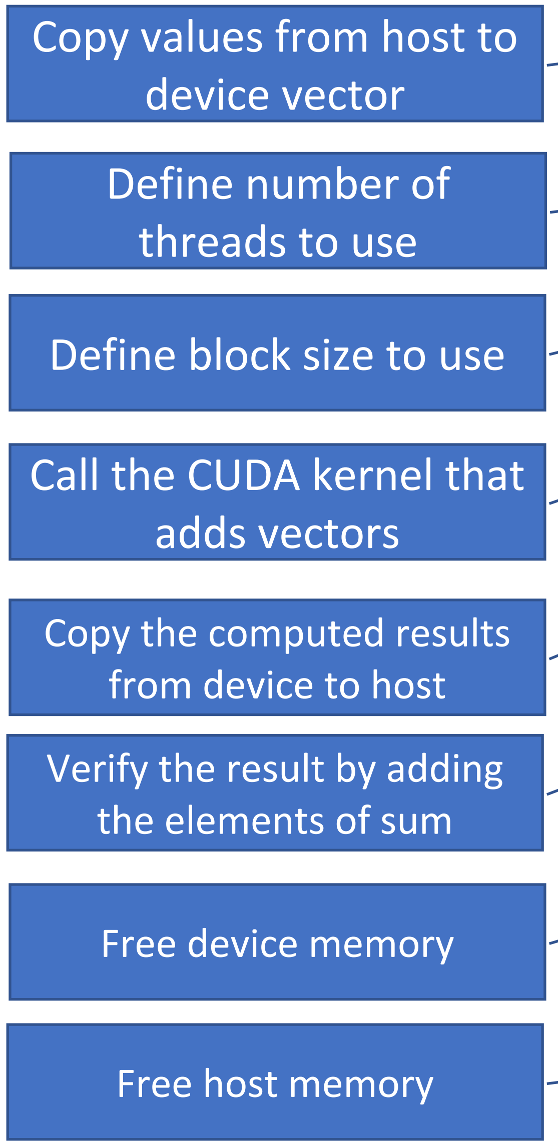
Allocate memory to host vectors

Allocate memory to device vectors

Initialize each elements of vectors to 1

```
int main(void) {
    //host vectors
    int *h_a, *h_b, *h_sum;
    //device vectors
    int* d_a, * d_b, * d_sum;
    // size of the total vectors necessary to allocate memory
    size_t size_vect = SIZE*sizeof(int);

    //allocate memory for the vectors on host (cpu)
    h_a = (int*)malloc(size_vect);
    h_b = (int*)malloc(size_vect);
    h_sum = (int*)malloc(size_vect);

    //allocate memory for the vectors on device (gpu)
    cudaMalloc((void **)&d_a, size_vect);
    cudaMalloc((void **)&d_b, size_vect);
    cudaMalloc((void **)&d_sum, size_vect);

    //initialize the vectors each with value 1
    for (int i = 0; i < SIZE; i++) {
        h_a[i] = 1;
        h_b[i] = 1;
    }

    //continued on next slide
```

# CUDA implementation: main function


Copy values from host to device vector
Define number of threads to use
Define block size to use
Call the CUDA kernel that adds vectors
Copy the computed results from device to host
Verify the result by adding the elements of sum
Free device memory
Free host memory

```
    //continued from previous slide
    //Start CUDA processing
    // Copy vector host values to device
    cudaMemcpy(d_a, h_a, size_vect, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size_vect, cudaMemcpyHostToDevice);

    //define number of threads
    int threads = 1024;
    //define block size in integer
    int block_size = (int)ceil((float)SIZE / threads);
    //execute the kernel with block size and number of threads
    add_device << <block_size, threads >>> (d_a, d_b, d_sum, SIZE);

    // Copy result back to host
    cudaMemcpy(h_sum, d_sum, size_vect, cudaMemcpyDeviceToHost);

    //Verify the result by adding all the sum, should be 2 * SIZE
    printf("Device sum:\n");
    sum_vect(h_sum);

    // Release all device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_sum);

    // Release all host memory
    free(h_a);
    free(h_b);
    free(h_sum);
}
```

# To compile and run

- Compile
    - nvcc –o vect_add vector_addition.cu
        - Syntax: nvcc <Filename.cu>
- Run
    - Windows:
        - vect_add
        - Syntax: <Output name>
    - Linux:
        - ./vect_add
            - Syntax: ./<Output name>
- Sample result (for #define SIZE 1000000)

```
Host sum:
2000000
Device sum:
2000000
```

# Exercise

- You are given the code for vector addition program in CUDA
- Write a similar program for multiplying three vectors.