ChatGPT

CUDA Vector Addition: Step-by-Step Explanation

The CUDA code adds two arrays **a** and **b** element-wise into **c** using the GPU. In general, this involves three main phases: preparing memory, running a GPU kernel, and retrieving results. Below we explain each part in plain terms.

1. Host vs. Device Memory

- **Host (CPU) memory:** The code first allocates ordinary C arrays for **a**, **b**, and **c** on the CPU (e.g. with `malloc` or `new`). These live in the *host* memory.
- **Device (GPU) memory:** It then allocates space on the GPU for each array using `cudaMalloc`. For example, `cudaMalloc(&dev_a, N*sizeof(int))` reserves memory on the GPU. CUDA requires this because CPU and GPU have separate memories [1].
- **Copying data to GPU:** After allocation, the code copies the input data from the CPU to the GPU. This is done with `cudaMemcpy`, which takes (destination, source, size, direction). For instance,

  ```
  cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
  ```

  copies the contents of host array **a** into GPU array **dev_a** [2] [3]. The flag `cudaMemcpyHostToDevice` indicates "CPU → GPU". Similarly, **b** is copied to `dev_b`.
- **Copying results back:** After the GPU computation, the result array **c** on the device is copied back to the host:

  ```
  cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
  ```

  Here `cudaMemcpyDeviceToHost` means "GPU → CPU" [2] [4].
- **Why two memories?** Think of the GPU as a separate computer: we must explicitly send data in and out. A CUDA guide notes that programs "manage device memory through calls to the CUDA runtime" and that we allocate GPU space so we can copy inputs from the host and later copy results back [1] [2].

2. The CUDA Kernel Function (`__global__`)

- **Definition:** The core computation is done in a special function marked with `__global__`, for example:

  ```
  __global__ void add(int *a, int *b, int *c, int N) {
      int idx = threadIdx.x;
      if (idx < N)
  ```

```
        c[idx] = a[idx] + b[idx];
}
```

The `__global__` keyword tells the compiler this is a **GPU kernel**: it runs on the GPU and can only be launched from host (CPU) code [5] [6]. NVIDIA explains that "every CUDA kernel starts with a `__global__` declaration" [5], and StackOverflow notes "**global** function is executed on GPU" [6]. In simple terms, this `add` function runs on the GPU.

- **Logic:** Inside the kernel, each thread does one addition. The snippet above shows each thread reading elements `a[idx]` and `b[idx]` and writing `c[idx] = a[idx] + b[idx]`. Because of the `if (idx < N)` guard, extra threads (if any) do nothing beyond array bounds. In effect, **each thread is responsible for one array element**. As one tutorial puts it, "each thread performs a single addition operation and computes a single new value" [7]. So with N threads, N pairs get added in parallel.
- **Thread vs. Block indexing:** CUDA provides built-in variables to identify each thread's work:
- `threadIdx.x` is the index **within its block**,
- `blockIdx.x` is the index of the **block within the grid**,
- `blockDim.x` is the number of threads in the block.

In code, we often compute a global index like:

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
c[index] = a[index] + b[index];
```

This formula combines block and thread indices to get a unique position [8] [9]. In our simple case with one block, `blockIdx.x` is 0 and `blockDim.x = N`, so `index = threadIdx.x`. In the kernel above we effectively set `idx = threadIdx.x`. Comments in example code emphasize this: "`blockIdx.x` is the index of the block... `threadIdx.x` is the index of the thread... Each thread can perform 1 addition" [8].

- **Analogy:** You can think of each thread like a worker on an assembly line. Each worker (thread) takes the two numbers at position *i*, adds them, and stores the sum. Many workers operate simultaneously on different positions, so the whole array gets added quickly.

3. Launch Configuration and Parallelism

- **Kernel launch syntax:** The host code invokes the GPU kernel with the triple-chevron syntax `<<<...>>>`. For example, `add<<<1, N>>>(dev_a, dev_b, dev_c, N);` launches **1 block** of **N threads** [10]. In general, the first number is how many blocks, the second is how many threads per block. NVIDIA notes that `<<<...>>>` "mark[s] a call from host code to device code" – i.e., a kernel launch [10].
- **Parallel execution:** When you launch a kernel, CUDA runs it across all threads **simultaneously**. A good way to see this is: if you launch 1 block of 5 threads (`<<<1,5>>>`), the GPU runs 5 parallel executions of the kernel code (one per thread) [11]. NVIDIA explains that the "parallel portion of your application is executed K times in parallel by K different CUDA threads, as opposed to only one time like regular C/C++ functions" [12]. In our case, with 1×N threads, the addition is done N times in parallel. For each thread `i`, the GPU runs `c[i] = a[i] + b[i]` at the same time.

- **Blocks and Threads:** More generally, CUDA grids can have many blocks and threads (up to 1024 threads per block [13]). Each block runs on a streaming multiprocessor, but all threads across the grid can work together. In the simple code, we use one block, so we just rely on `threadIdx.x`. If there were multiple blocks, `blockIdx.x` and `blockDim.x` would shift the index for threads in block 1, block 2, etc.

4. Retrieving Results and Cleanup

- **Copying back and printing:** After the kernel finishes, the code copies the result array from GPU to CPU (as mentioned above) and then typically loops over **c** on the CPU to print each sum (e.g. `printf("%d + %d = %d\n", a[i], b[i], c[i])`). This printing is just standard C code on the host (the GPU work is already done).
- **Freeing memory:** Finally, the code frees all allocated memory. On the GPU, it calls `cudaFree(dev_a)`, `cudaFree(dev_b)`, `cudaFree(dev_c)`, and on the CPU it frees or deletes **a**, **b**, **c**. This cleanup avoids memory leaks. The example guide shows these steps clearly [14].
- **Summary of steps:** In fact, a CUDA tutorial summarizes the whole process in order:
- Allocate host arrays and initialize them.
- Allocate device arrays with `cudaMalloc`.
- Copy data from host to device with `cudaMemcpy`.
- Launch the GPU kernel to do the addition.
- Copy the result array back to the host.
- Free GPU and CPU memory [15].

By following these steps, the code leverages GPU parallelism: **each thread on the GPU adds one pair of elements**, and all threads work together to compute the vector sum much faster than a simple CPU loop.

Sources: Authoritative CUDA docs and tutorials were used to explain these steps and concepts [1] [16] [5] [8] [10] [7].

---

[1] [8] **"CUDA Tutorial"**
https://jhui.github.io/2017/03/06/CUDA/

[2] [3] [4] [14] [16] **Vector Addition "Hello World!" Example with CUDA on Mac OSX | QuantStart**
https://www.quantstart.com/articles/Vector-Addition-Hello-World-Example-with-CUDA-on-Mac-OSX/

[5] [10] [12] [13] **CUDA Refresher: The CUDA Programming Model | NVIDIA Technical Blog**
https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[6] **cuda - what does __global__ mean in the context of a function declaration in c? - Stack Overflow**
https://stackoverflow.com/questions/69409395/what-does-global-mean-in-the-context-of-a-function-declaration-in-c

[7] [15] **4.4 Example: Vector Addition — Parallel Computing for Beginners**
https://www.learnpdc.org/PDCBeginners/4-cuda/4-VectorAdd.html

[9] **harmanani.github.io**
https://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf

[11] **Understanding the basics of CUDA thread hierarchies - EximiaCo**
https://eximia.co/understanding-the-basics-of-cuda-thread-hierarchies/