# Basic of CUDA and cuda_runtime API functions

## CUDA Thread Management

- **`threadIdx`, `blockIdx`, and `blockDim`:** These are **built-in variables** in CUDA that help manage **threads** and **blocks**.
    - **`threadIdx.x`**: ID of the thread within its block.
    - **`blockIdx.x`**: ID of the block within the grid.
    - **`blockDim.x`**: Total number of threads per block.

  **Example:**
  ```
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  ```

- This computes a **unique index** for each thread across all blocks.

## Choosing Block Size

1. **Block size (number of threads per block):**
    - A good starting point is **256 or 512 threads per block**. This is because most GPUs work efficiently with **warps** of 32 threads, and blocks are ideally a multiple of 32.
2. **Grid size (number of blocks):**

Grid size is calculated to **cover all data points**. For example:
cpp
Copy code
```
int gridSize = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
```

    -
    - This ensures that even if `N` isn't perfectly divisible by `BLOCK_SIZE`, all elements are processed.

---

## Block Size and Grid Size Example

- Suppose you have **1,000,000 elements** to process (`N = 1,000,000`).

- If you choose **256 threads per block** (`BLOCK_SIZE = 256`), the **grid size** will be:

Suppose you have **1,000,000 elements** to process (`N = 1,000,000`).

If you choose **256 threads per block** (`BLOCK_SIZE = 256`), the **grid size** will be:

$$\text{Grid Size} = \frac{N + \text{BLOCK\_SIZE} - 1}{\text{BLOCK\_SIZE}} = \frac{1,000,000 + 256 - 1}{256} = 3907 \text{ blocks}$$

### 1. `cudaMalloc()`

- **Purpose:** Allocates memory on the **GPU device**.

**Usage:**

```
CUDA_CHECK(cudaMalloc(&d_a, N * sizeof(float)));
```

- This allocates `N` elements of type `float` on the GPU and stores the address in `d_a`.
- **Explanation:** Similar to `malloc()` in C, but the allocated memory resides on the GPU, enabling **fast parallel access** by CUDA kernels.

### 2. `cudaMemcpy()`

- **Purpose:** Transfers data between the **host (CPU) and device (GPU)** memory.

**Usage:**

```
CUDA_CHECK(cudaMemcpy(d_a, h_a, N * sizeof(float),
cudaMemcpyHostToDevice));
```

- This copies `N` elements from the host array `h_a` to the device array `d_a`.
- **Modes:**
  - `cudaMemcpyHostToDevice`: Copies from CPU to GPU.
  - `cudaMemcpyDeviceToHost`: Copies from GPU to CPU.
  - `cudaMemcpyDeviceToDevice`: Copies between GPU memory regions.

### 3. `cudaMemset()`

- **Purpose:** Initializes **device memory** with a specific value (like `memset()` for GPU).

**Usage:**

```
CUDA_CHECK(cudaMemset(d_histogram, 0, 256 * sizeof(int)));
```

- This sets all values in the histogram array on the GPU to **0**.

- **Explanation:**
  Useful for initializing arrays before running kernels to prevent random or uninitialized values.

### 4. `cudaDeviceSynchronize()`

- **Purpose:** Blocks the CPU until all **previously launched GPU kernels** complete.

**Usage:**
```
CUDA_CHECK(cudaDeviceSynchronize());
```

- Ensures that **timing measurements** or **data transfers** are accurate by waiting for all GPU operations to finish.
- **Explanation:**
  CUDA kernels run asynchronously. This function ensures that all GPU work is complete before proceeding on the CPU side.

### 5. `cudaFree()`

- **Purpose:** Frees **GPU memory** allocated with `cudaMalloc()`.

**Usage:**
```
CUDA_CHECK(cudaFree(d_a));
```

- Frees the memory associated with `d_a` on the GPU.

### 6. CUDA Kernels (`__global__` Functions)

Kernels are **functions that run on the GPU** and are defined with the `__global__` keyword. Each kernel is executed by **multiple GPU threads** in parallel.

**Example Kernel Usage:**
cpp
Copy code
```
vectorAdd<<<(N + BLOCK_SIZE - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(d_a, d_b, d_c, N);
```

- 
  - **Launch Configuration:** `<<<gridSize, blockSize>>>` specifies how many **threads and blocks** to use.
    - `blockSize`: Number of threads per block (256 in this case).
    - `gridSize`: Number of blocks to launch (computed based on input size).

**7. `atomicAdd()`**

- **Purpose:** Ensures **atomic (thread-safe) addition** on shared memory (e.g., when multiple threads need to increment the same value).

**Usage:**

```
atomicAdd(result, input[idx]);
```

- Ensures that multiple threads updating the histogram do not overwrite each other's results.

**8. `atomicMin()`**

- **Purpose:** Ensures **atomic (thread-safe) minimum** operation.

**Usage:**

```
atomicMin(reinterpret_cast<int*>(result), __float_as_int(diff));
```

- Converts a float value to an integer and finds the minimum value atomically.