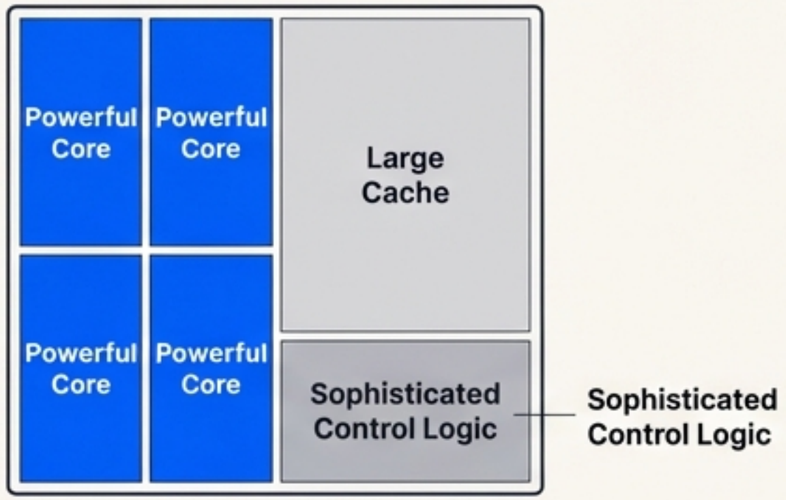
# The Era of Parallelism: Why Your CPU Is Not Enough

In 2005, the ‘Power Wall’ halted the **increase in single-processor clock speeds**. To gain more performance, the industry shifted to multi-core processors and massively parallel architectures like GPUs.

## CPU: Latency-Oriented

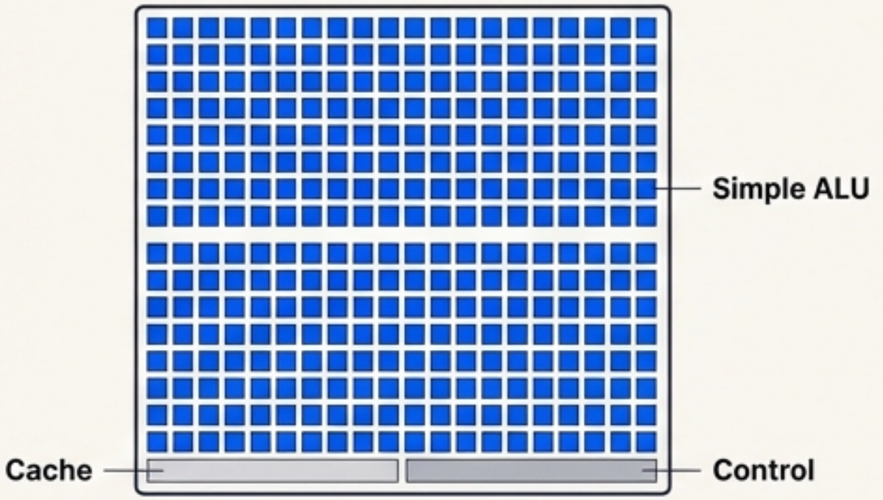
A sports car, designed to make a *single task* run as fast as possible.




- Few powerful cores
- Large caches
- Sophisticated control logic
- High clock frequency

## GPU: Throughput-Oriented

A bus, designed to execute a *massive number of tasks* simultaneously.

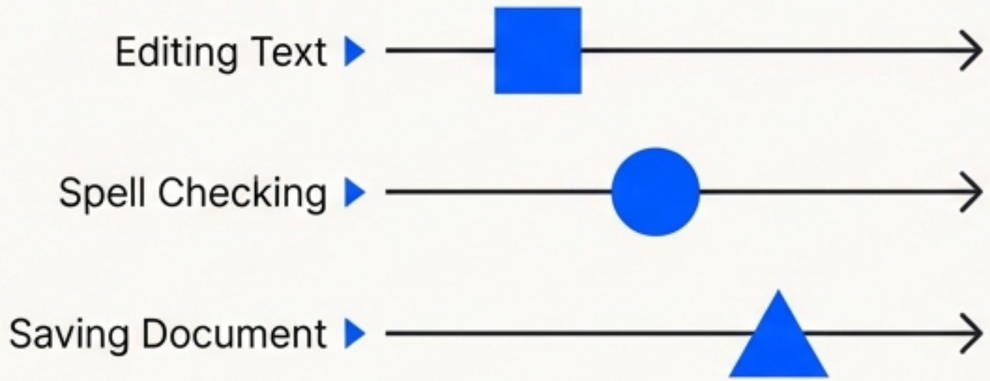



- Many simple cores (ALUs)
- Smaller caches
- Simple control logic
- Moderate clock frequency

# Finding the Right Kind of Parallelism

## Task Parallelism

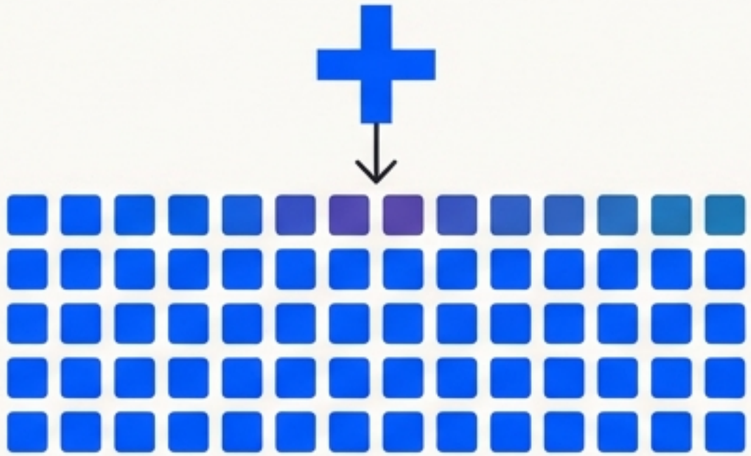
Performing *different operations* in parallel.




Offers modest parallelism. An application only has so many distinct tasks to run at once.

## Data Parallelism

Performing the *same operation* on many different pieces of data in parallel.



Calculating the color value for every pixel on a high-resolution screen.

**Key Insight:** Data parallelism offers a path to *massive* parallelism. To increase it, you don't write more code; you just run the same code on a larger dataset. **This is the sweet spot for GPUs.**

# Our ‘Hello, World’: Vector Addition

## The Problem

We want to add two vectors, X and Y, to produce a third vector, Z.
Z[i] = X[i] + Y[i] for every element i.

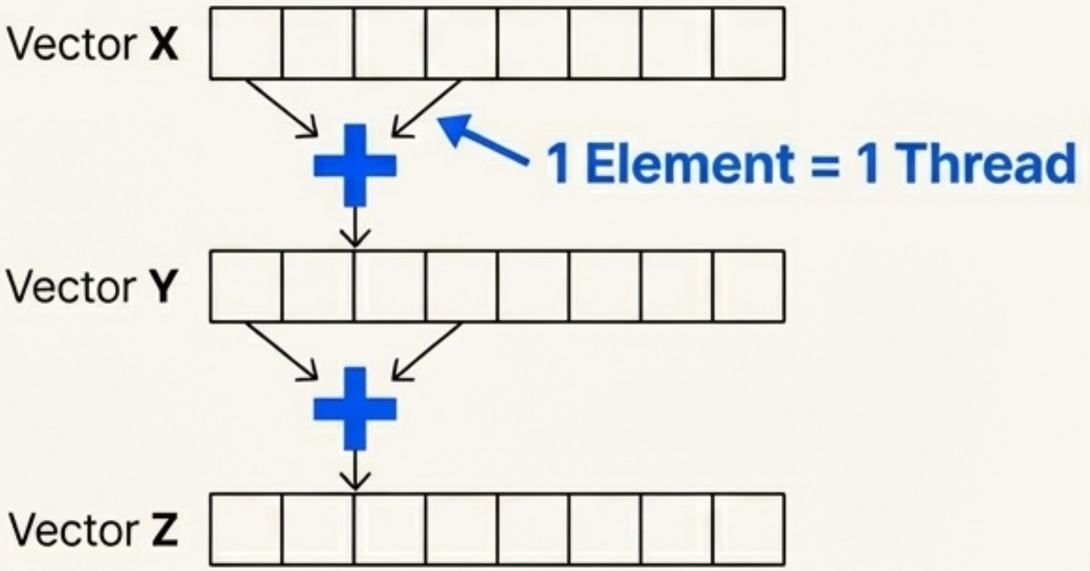
## The Sequential CPU Approach

A simple for loop iterates through the elements one by one.

```cpp
// vec_add_cpu.cpp
void vec_add_cpu(float* z,
                 const float* x,
                 const float* y, int n) {
  for (int i = 0; i < n; ++i) {
     z[i] = x[i] + y[i];
  }
}
```

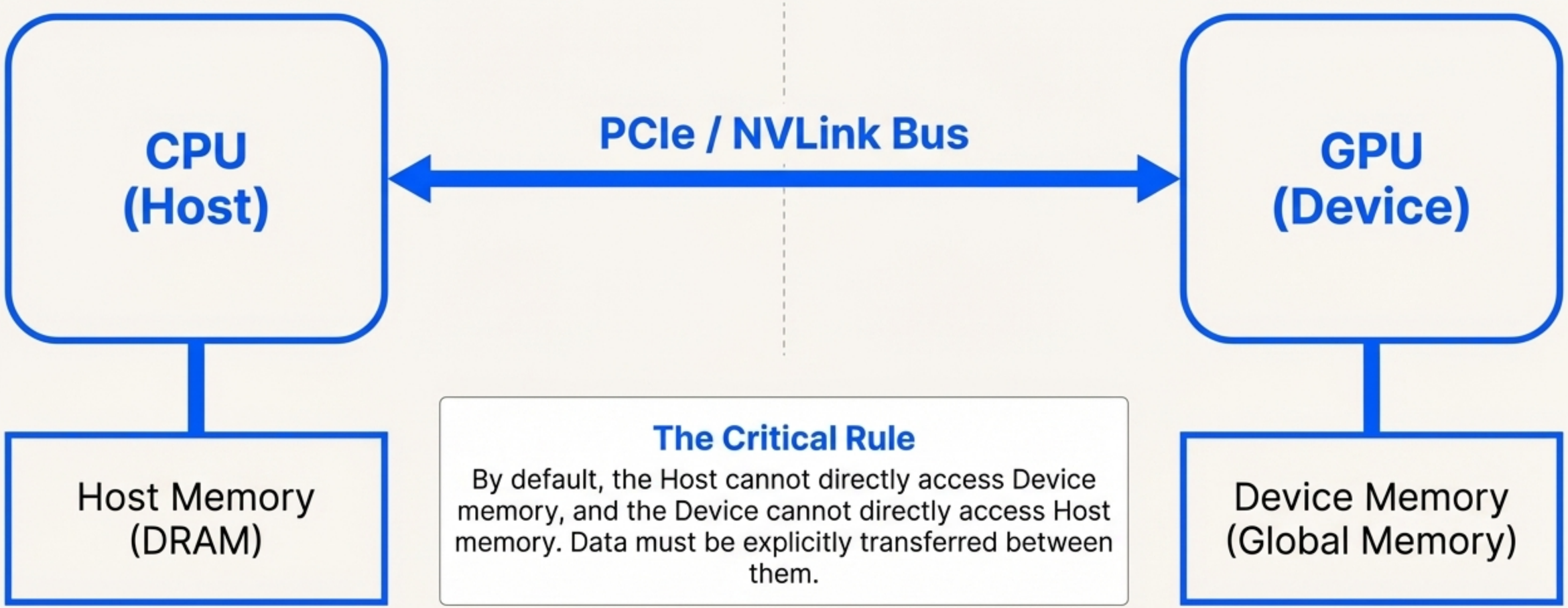
## The Data Parallel GPU Approach

Assign one thread to each element. Every thread performs the same addition operation, but on its unique pair of elements (X[i], Y[i]).



# The Host and The Device: A Tale of Two Memories

A typical GPU-accelerated system has two distinct processing units with their own dedicated memory.

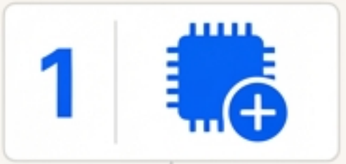



### The Critical Rule

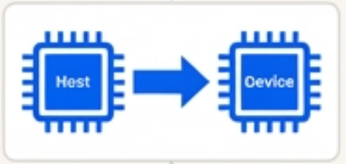
By default, the Host cannot directly access Device memory, and the Device cannot directly access Host memory. Data must be explicitly transferred between them.

# The 5-Step CUDA Workflow

Offloading computation to the GPU follows a standard, repeatable pattern. Mastering this sequence is the key to writing your first CUDA program.

1 

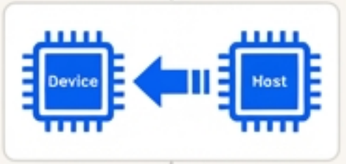
**Allocate** memory on the Device (GPU) for inputs and outputs.

2 

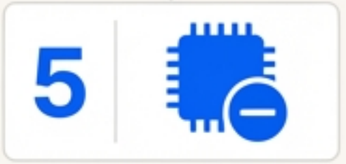

**Copy** input data from Host (CPU) memory to Device (GPU) memory.

3 

**Launch Kernel** to perform the computation on the Device.

4 


**Copy** results from Device (GPU) memory back to Host (CPU) memory.

5 

**Deallocate** memory on the Device.

# Step 1 & 5: Managing Device Memory

To manage memory on the GPU, we use CUDA's built-in functions.

### Step 1: Allocate with `cudaMalloc`

This function allocates a specified number of bytes in the GPU's global memory. Note the syntax: It takes a pointer to a pointer (`void**`) to return the allocated address, using the function's return value for an error code (`cudaError_t`).

### Step 5: Deallocate with `cudaFree`

Frees memory previously allocated with `cudaMalloc`. Good practice: Write your `cudaFree` call immediately after `cudaMalloc` to avoid memory leaks.

```
// Building vec_add_gpu.cu
void vec_add_gpu(...) {
  float *x_d, *y_d, *z_d; // _d is a
  convention for "device" pointers
  int size = n * sizeof(float);

  // Step 1: Allocate memory on the GPU
  cudaMalloc((void**)&x_d, size);
  cudaMalloc((void**)&y_d, size);
  cudaMalloc((void**)&z_d, size);

  // [Steps 2, 3, 4 will go here]

  // Step 5: Deallocate GPU memory
  cudaFree(x_d);
  cudaFree(y_d);
  cudaFree(z_d);
}
```

# Step 2 & 4: Moving Data Across the PCIe Bus

**cudaMemcpy** is the function used to copy data between Host and Device memory.

```
cudaMemcpy(destination, source, size, direction)
```

- **cudaMemcpyHostToDevice**: Copies from CPU to GPU (Step 2).
- **cudaMemcpyDeviceToHost**: Copies from GPU to CPU (Step 4).
- (Others exist, like **DeviceToDevice** and **HostToHost**).

```
// Building vec_add_gpu.cu (continued)
void vec_add_gpu(float* z, const float* x,
    const float* y, int n) {
  ... // Allocation code from previous slide (grayed

  // Step 2: Copy input vectors from host to device
  cudaMemcpy(x_d, x, size, cudaMemcpyHostToDevice);
  cudaMemcpy(y_d, y, size, cudaMemcpyHostToDevice);

  // [Step 3: Launch Kernel]
  // Step 4: Copy result from device to host
  cudaMemcpy(z, z_d, size, cudaMemcpyDeviceToHost);

  ... // Deallocation code from previous slide (graye
}
```

# Step 3: Launching a Kernel

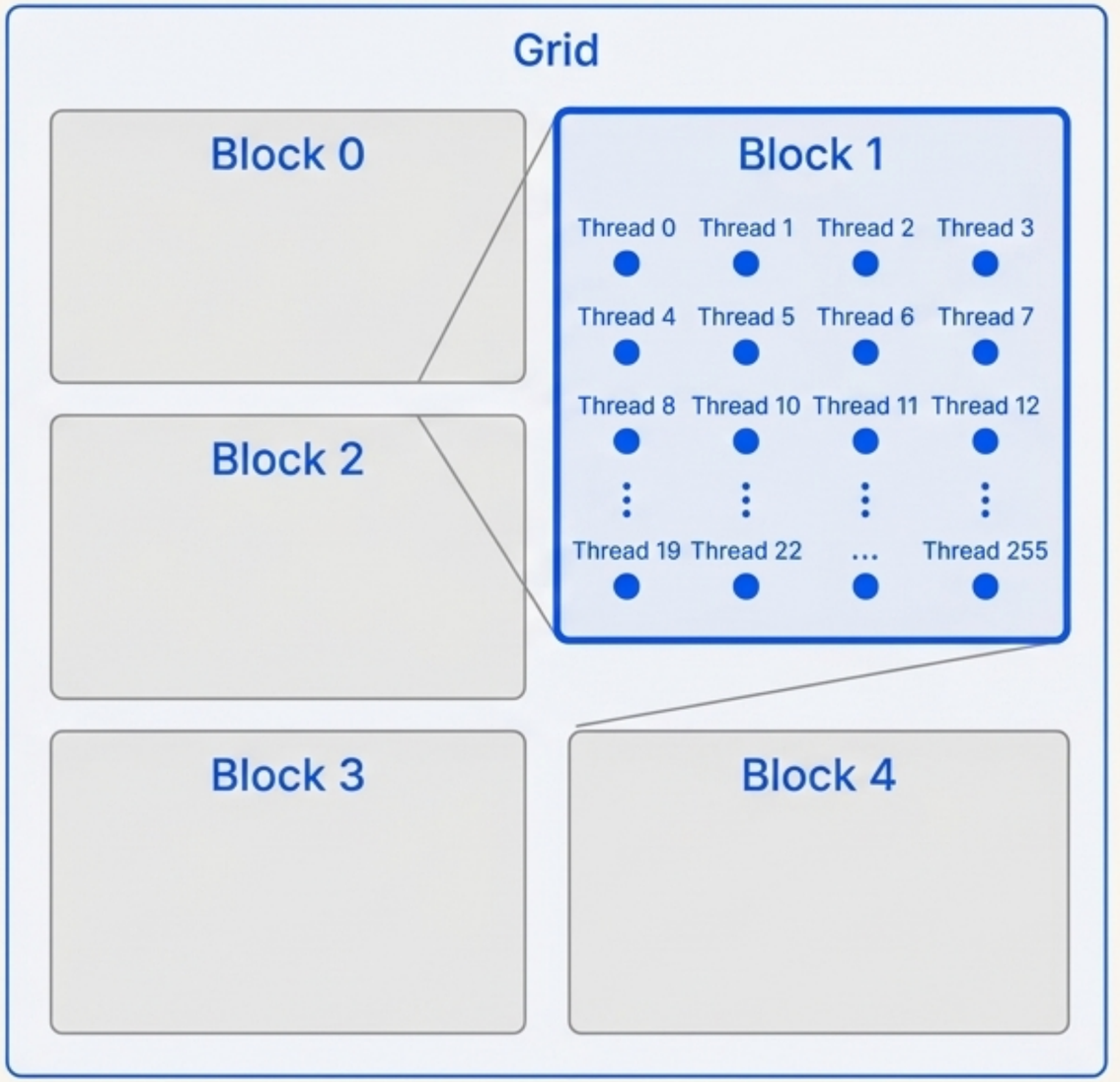
A **kernel** is a function that runs on the GPU. We launch a kernel to execute our code in parallel across many threads.

**The Thread Hierarchy:**

- A **Grid** is the entire collection of threads launched by a single kernel call.
- The Grid is divided into **Thread Blocks**.
- Threads within the same block can cooperate in special ways (more on this in future topics).

**Launch Syntax:**
CUDA C++ extends C++ with a special `<<<...>>>` syntax.

```
kernel_name
<lt;<l< Num_Blocks, Threads_Per_Block &gt;
(args...);
```



# Configuring the Grid for Vector Addition

**Goal:** Our goal is to launch one thread for each element in the vector (a total of N threads).

**Step 1: Choose a Block Size.**

This is the number of threads per block. It's often a power of two for performance reasons.
A typical value is 256 or 512.
Let's pick 512.

```
// In vec_add_gpu.cu
const int threadsPerBlock = 512;
const int numBlocks = n / threadsPerBlock; // NOTE: This has a bug!

// The kernel launch
vec_add_kernel<<<numBlocks, threadsPerBlock>>>(z_d, x_d, y_d, n);
```

**Step 2: Calculate the Number of Blocks.**

To get N total threads, we divide N by our block size.

```
Num_Blocks = N / Threads_Per_Block;
```

**Warning: Integer Division!**

What if n is not perfectly divisible by 512? The code above will launch *too few* threads. We'll fix this in a moment.

# Inside the Kernel: Every Thread Needs a Unique ID

## Kernel Definition

A **kernel** function is defined using the `__global__` specifier. It's executed by every thread in the grid.

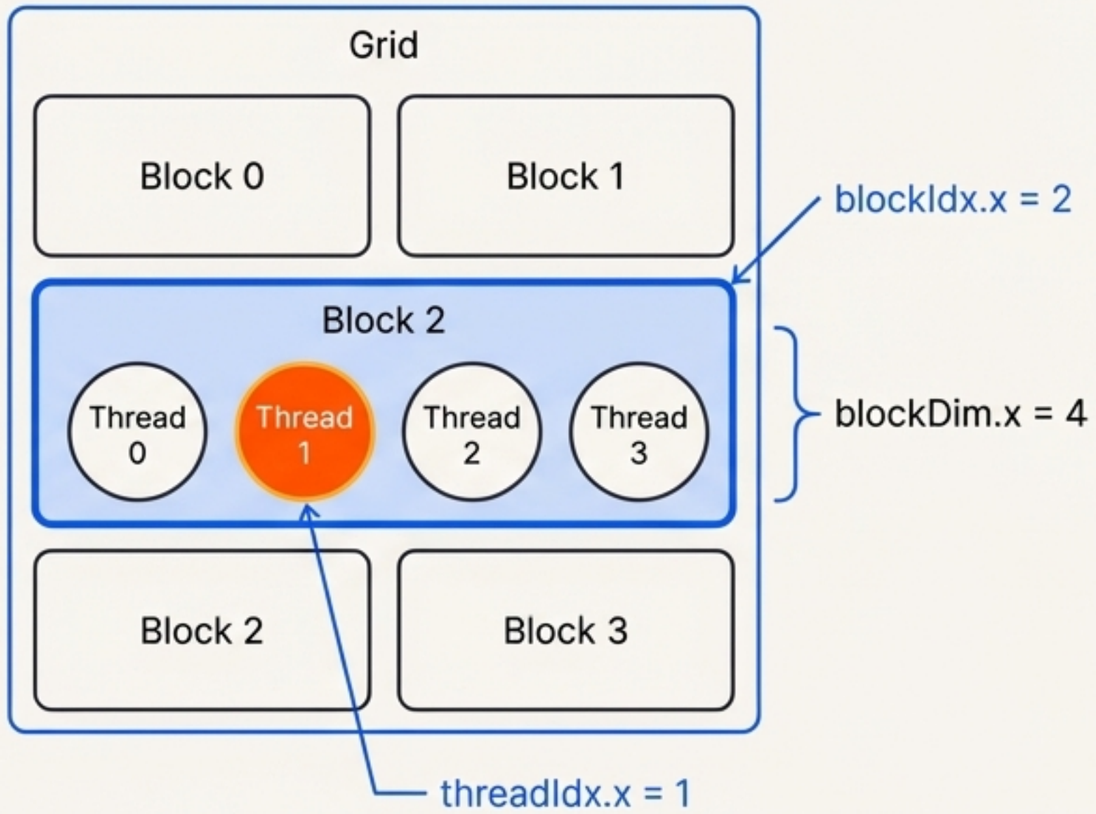
## CUDA Built-in Variables

Each thread has access to special variables that describe its position:

- `blockDim.x`: The number of threads in this block. (e.g., 512)
- `blockIdx.x`: The index of this thread's block within the grid.
- `threadIdx.x`: The index of this thread within its block.

## Calculating the Global Index

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```




Global Index **i** = (blockIdx.x * blockDim.x) + threadIdx.x

i = (2 * 4) + 1 = **9**

# The Vector Addition Kernel

Every thread launched will execute this function. Each thread calculates its unique index `i` and then performs the addition for that single element. This is an example of **SPMD (Single Program, Multiple Data)**: All threads run the same program, but operate on different data based on their unique ID.

```
// vec_add_kernel.cu
__global__
void vec_add_kernel(float* z, const float* x, const float* y, int n) {
    // Calculate the global thread index
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Each thread adds one pair of elements
    z[i] = x[i] + y[i]; // NOTE: Still has a bug!
}
```

**Warning: Out-of-Bounds Access!**

If we launch more threads than elements (which we need to do to handle non-divisible sizes), some threads will have an index `i >= n`. Accessing `x[i]` would be an error.

# Making It Robust: Handling Any Size N

We need to fix two issues to handle vector sizes that are not perfect multiples of our block size.

**Fix 1: Launch Enough Threads using Ceiling Division.** To ensure we have enough threads to cover all N elements, **we must round up when calculating** numBlocks.

`The Integer Math Trick for Ceiling: (n + threadsPerBlock - 1) / threadsPerBlock`

**Fix 2: Add a Boundary Check in the Kernel.** Since we might launch extra threads, each thread must first check if its calculated index `i` is within the valid range of the array before performing any work.

**Left Code Block:**

```
// In the host function:
const int threadsPerBlock = 512;
const int numBlocks = (n + threadsPerBlock - 1) /
    threadsPerBlock;
vec_add_kernel<<<numBlocks, threadsPerBlock>>>(...);
```

CEILING DIVISION

**Right Code Block:**

```
// In the kernel:
__global__ void vec_add_kernel(...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        z[i] = x[i] + y[i];
    }
}
```

← BOUNDARY CHECK

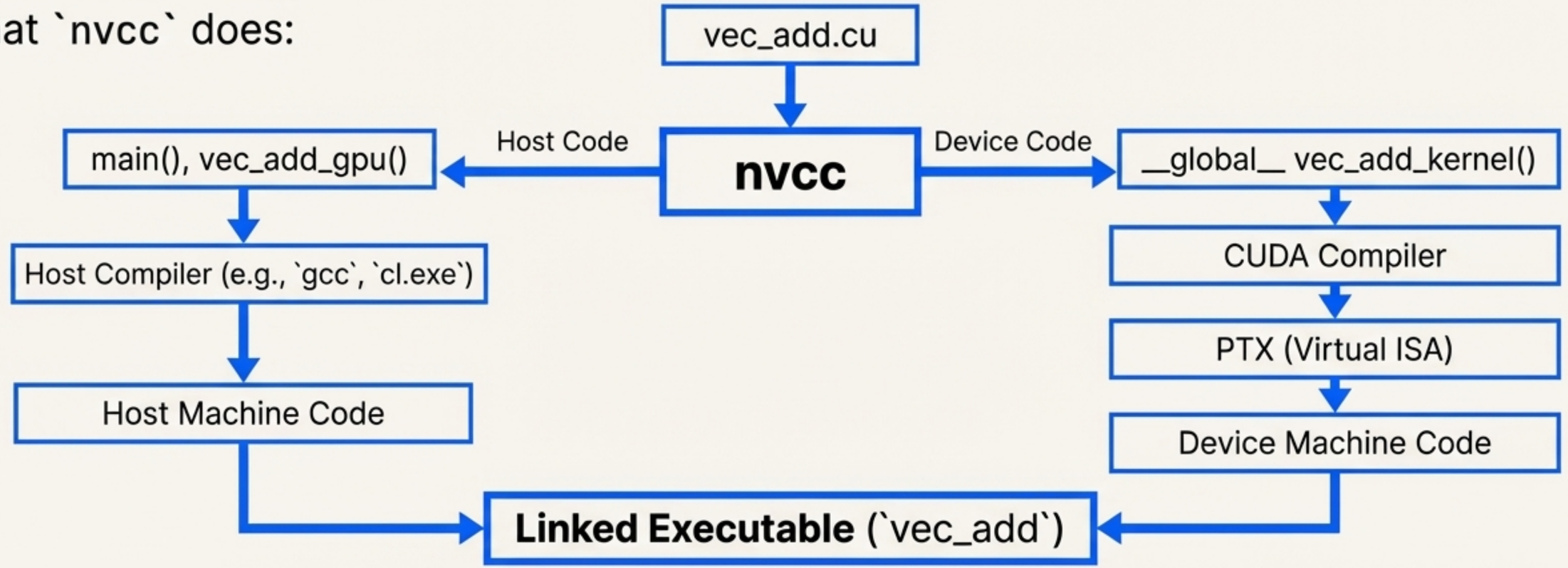
# Compiling Your Code with `nvcc`

CUDA source files typically use the `.cu` extension.

They are compiled using the NVIDIA Cuda Compiler (`nvcc`).

```
nvcc vec_add.cu -o vec_add
```

What `nvcc` does:



# The Payoff: Measuring Performance Correctly

We ran the vector addition for N = $2^{25}$ (≈33.5 million) elements.

### First Look: Total Execution Time

**CPU Time: 0.20 seconds** **GPU Total Time: 0.12 seconds**


The GPU is faster, but only by about **1.7x**. This can feel disappointing.

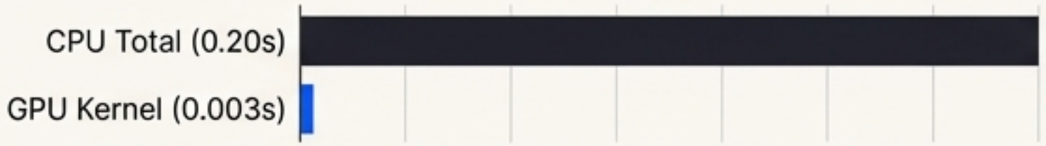
---

## The Real Story: Isolating the Computation

The GPU total time includes memory allocation and data transfers (`cudaMemcpy`), which have significant overhead. In real applications, you copy data once, run many kernels, and copy back once. Let's time *only the kernel execution.*

# GPU Kernel-Only Time: 0.003 seconds

### The 'Aha!' Moment

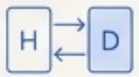


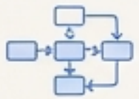
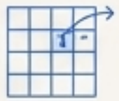

Comparing computation to computation:
**0.20s (CPU) vs. 0.003s (GPU) is a ~67x speedup.**

**Lesson: The massive parallelism of the GPU provides huge speedups on the computation itself. The key to performance is to minimize data transfer overhead relative to the amount of computation.**

# You Just Wrote Your First CUDA Program

You have now learned the complete, end-to-end process for accelerating a C++ function with a GPU:

1.  Identified a **data-parallel** problem.
2. 

Understood the **Host/Device memory model**.
3.  Implemented the **5-step CUDA workflow** (Allocate, Copy, Launch, Copy, Free).
4.  Wrote a **kernel** where each thread computes a unique index to work on a piece of data.
5.  Handled edge cases and correctly **measured the performance** of the parallel computation.

---

## What's Next?

This fundamental pattern is the basis for everything else. The rest of parallel programming is about applying this workflow to more complex problems and using advanced optimization strategies to maximize performance. **You now have the foundational toolkit.**