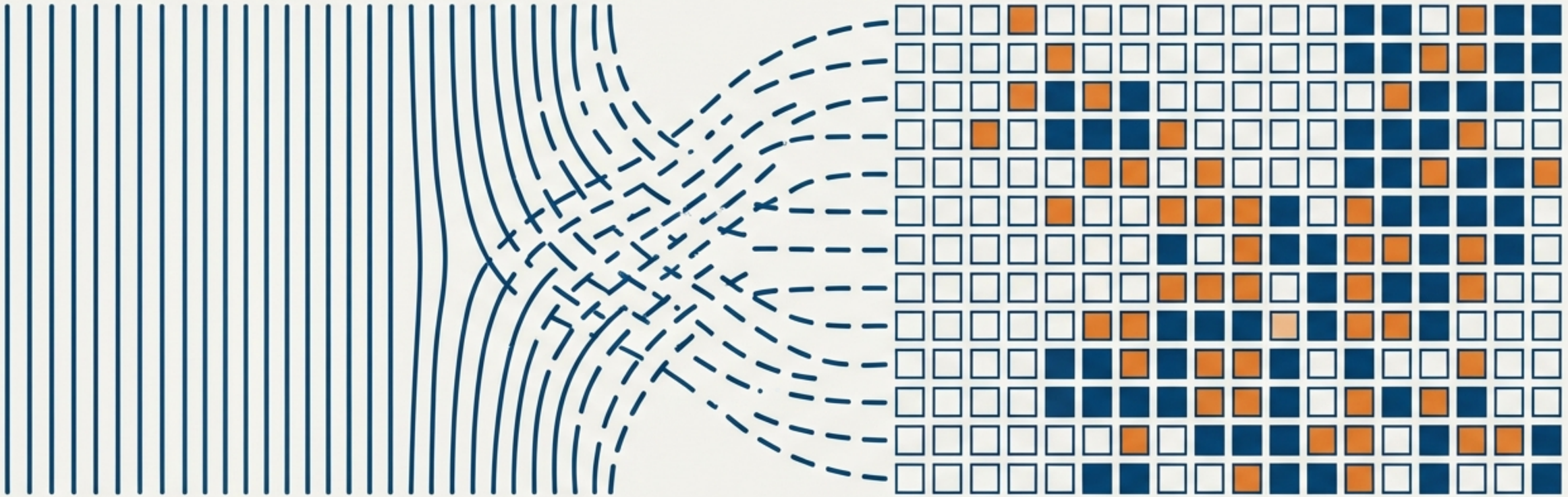
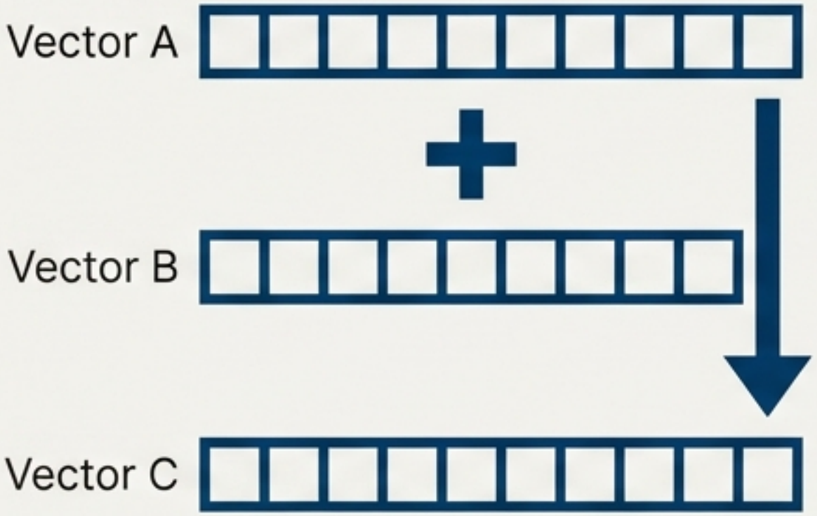
From Lines to Images: Mastering Multi-Dimensional Data in CUDA

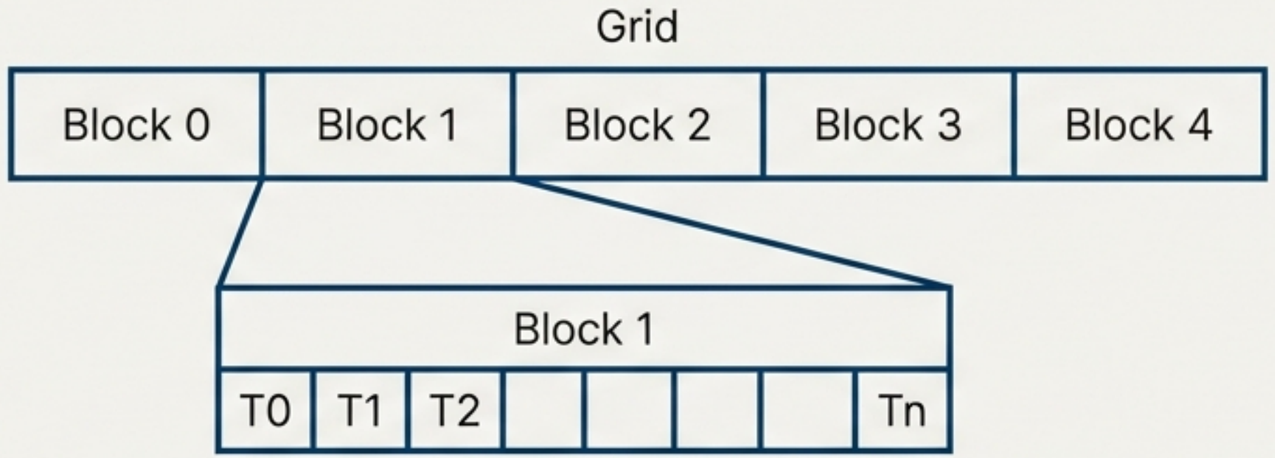
Extending Data Parallelism from 1D to 2D for Real-World Applications



Where We Left Off: Data Parallelism in One Dimension



One thread per element.

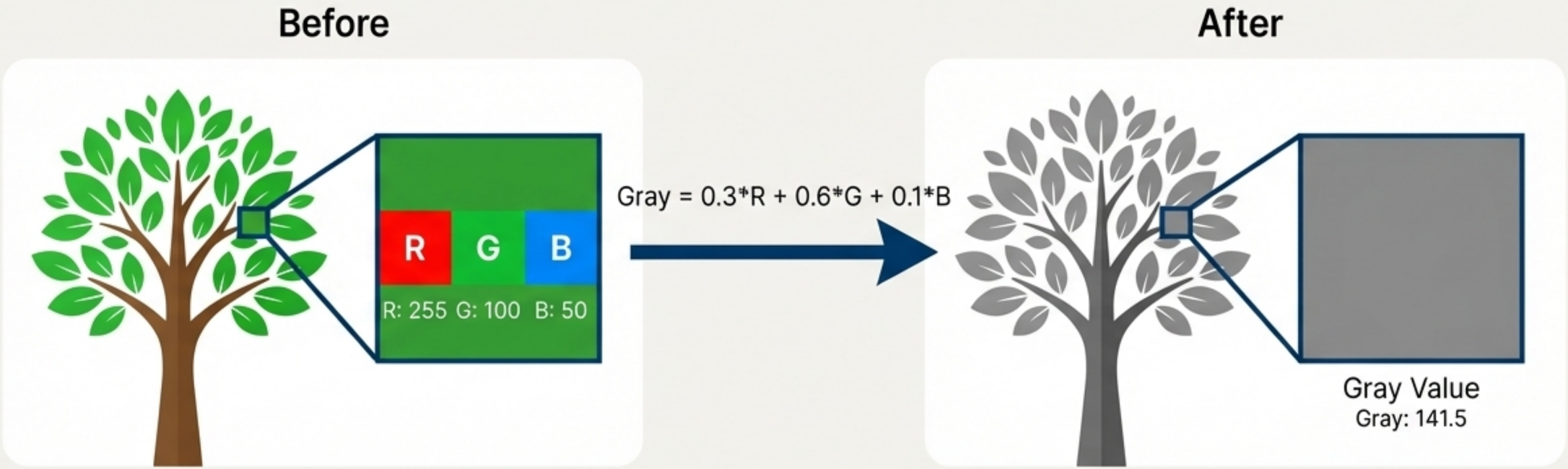



```
// How each thread finds its unique element in 1D
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

We mastered 1D data. Now, let's tackle the complexity of 2D.

Our First Challenge: Converting a Color Image to Grayscale

We need to transform an image where each pixel has three color values (Red, Green, Blue) into an image where each pixel has a single intensity value.

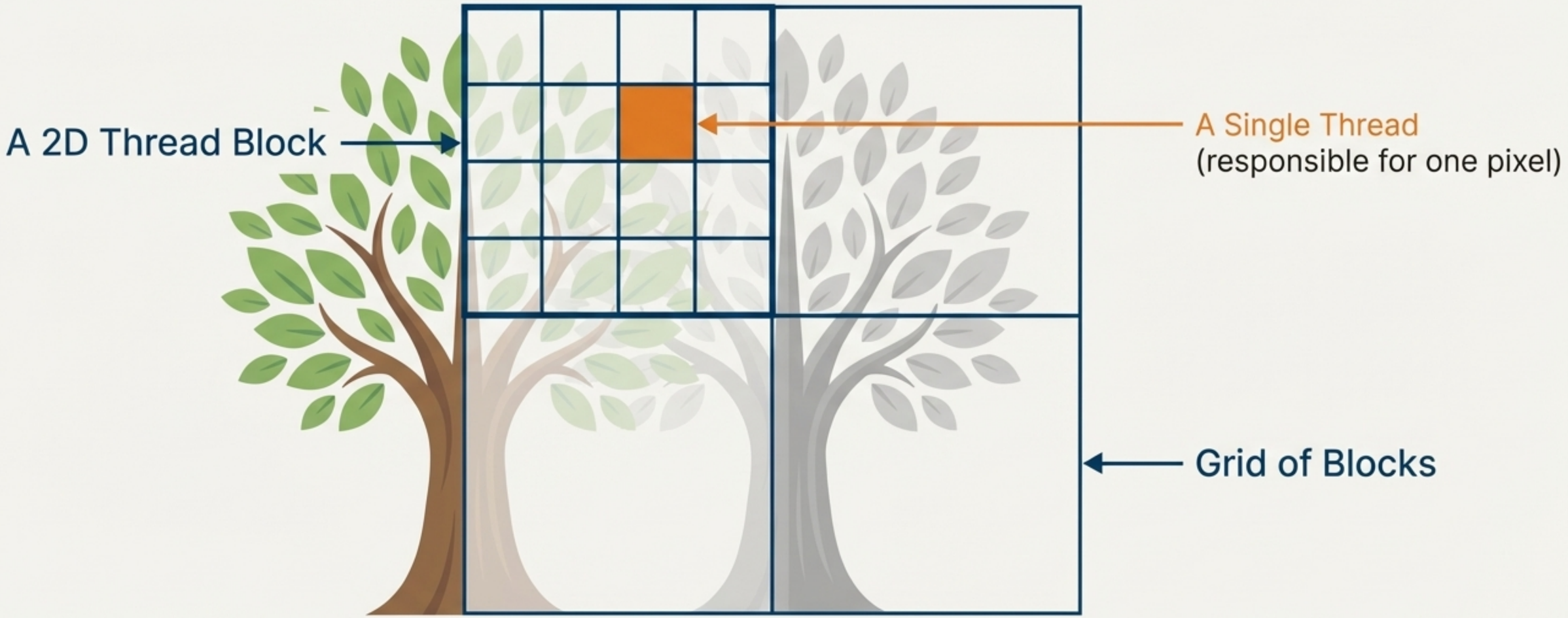


The Parallel Strategy

The most straightforward approach is a direct 1-to-1 mapping: **Assign one thread to convert each pixel in the image.**

Mapping Threads to Pixels with a 2D Grid

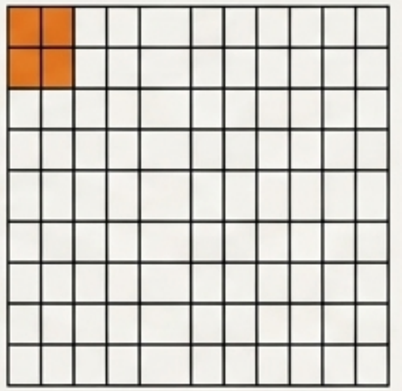
Since our data is a 2D image, it's more intuitive to organize our threads in a 2D grid. CUDA supports this directly.



The Tool for 2D Configuration: `dim3` and Kernel Launch

To launch a 2D grid, we can no longer use simple integers. CUDA provides the `dim3` type, a 3-component vector for defining dimensions.

```
// 1. Define the dimensions of a single thread block
// We'll use 32x32 threads per block.
dim3 threadsPerBlock(32, 32); // .z defaults to 1

// 2. Calculate the number of blocks needed for the whole image
// We use ceiling division to ensure we cover all pixels.
dim3 numBlocks( (width + threadsPerBlock.x - 1) / threadsPerBlock.x,
                (height + threadsPerBlock.y - 1) / threadsPerBlock.y );

// 3. Launch the kernel with the 2D configuration
rgbToGray_kernel<<<numBlocks, threadsPerBlock>>>(...);
```



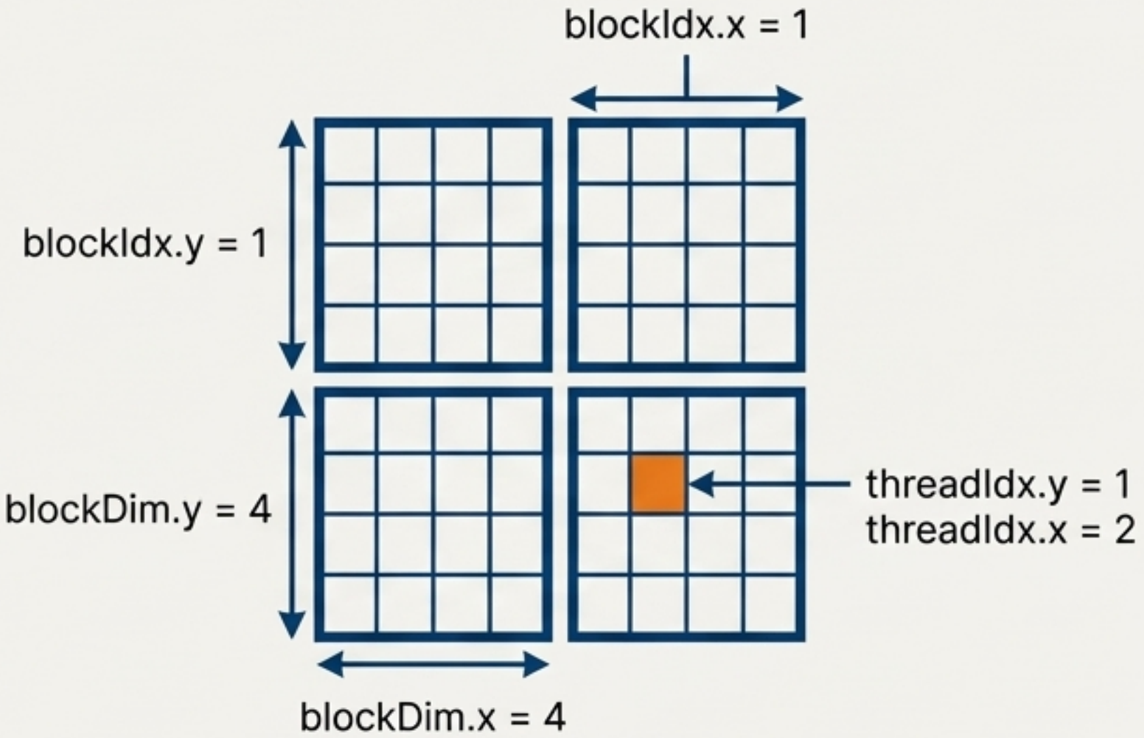
Block Shape

This is a common integer arithmetic trick for `ceil(N/D)`.

Finding Your Place: Calculating a Global 2D Index

Inside the kernel, each thread uses built-in variables with .x and .y components to calculate its unique row and column.

```
// Calculate the thread's global row index
int row = blockIdx.y * blockDim.y + threadIdx.y;

// Calculate the thread's global column index
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

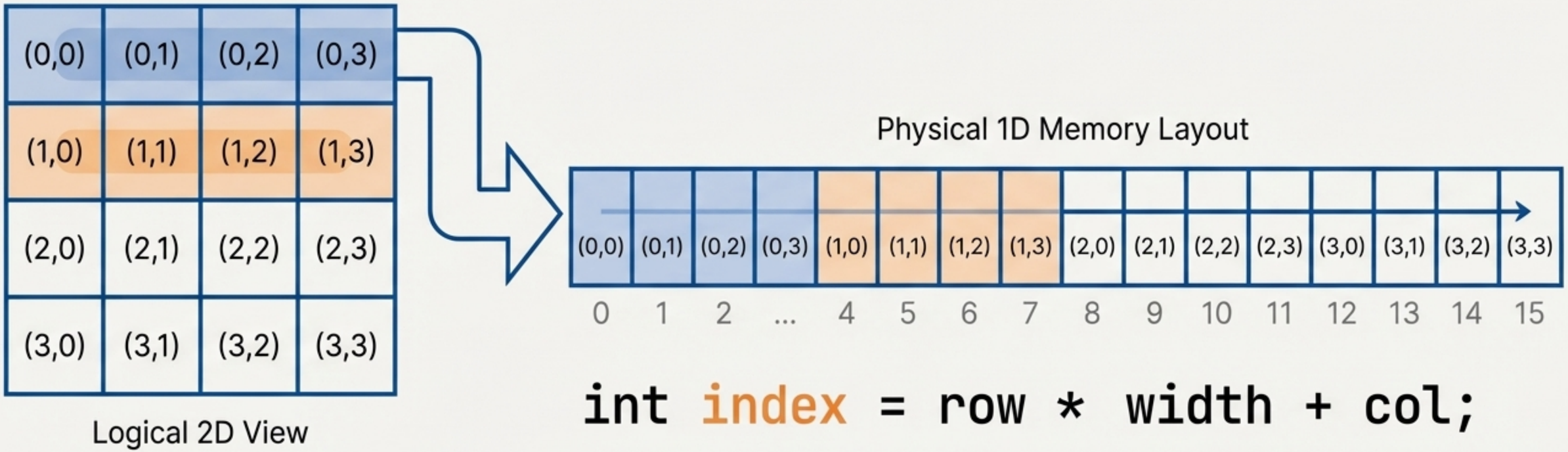


row = 1 * 4 + 1 = **5**
col = 1 * 4 + 2 = **6**

The Reality of Memory: How 2D Data Lives in a 1D World

CPUs and GPUs access memory as a single, contiguous 1D array. How do we map our 2D (row, col) coordinate to an index in this array?

The Solution: Row-Major Order. C/C++ stores 2D arrays in 'row-major' order: Row 0, followed by Row 1, and so on.



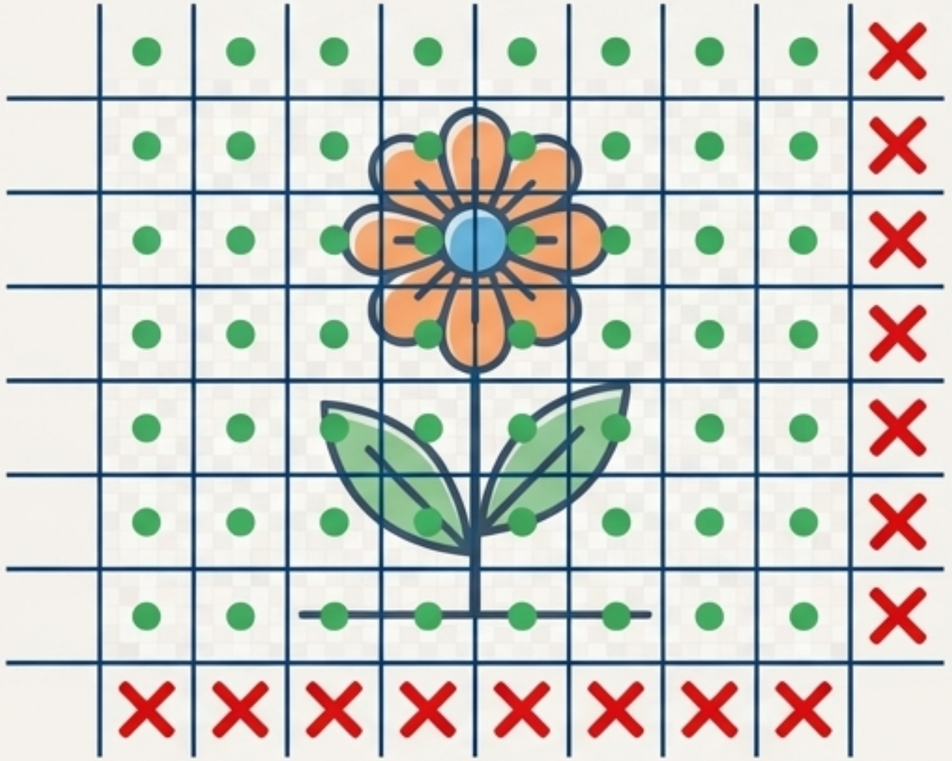

```
int index = row * width + col;
gray_out[index] = ...;
```

The Hidden Complexity: What If the Grid is Larger Than the Image?

The Ideal vs. The Real

Our ceiling division ensures we launch *enough* threads, but it often means we launch *too many*.

What happens to threads whose `(row, col)` falls outside the image dimensions?



The Consequence

These out-of-bounds threads will attempt to read from and write to invalid memory addresses, leading to incorrect results or crashes.

The Solution: Guarding Every Memory Access

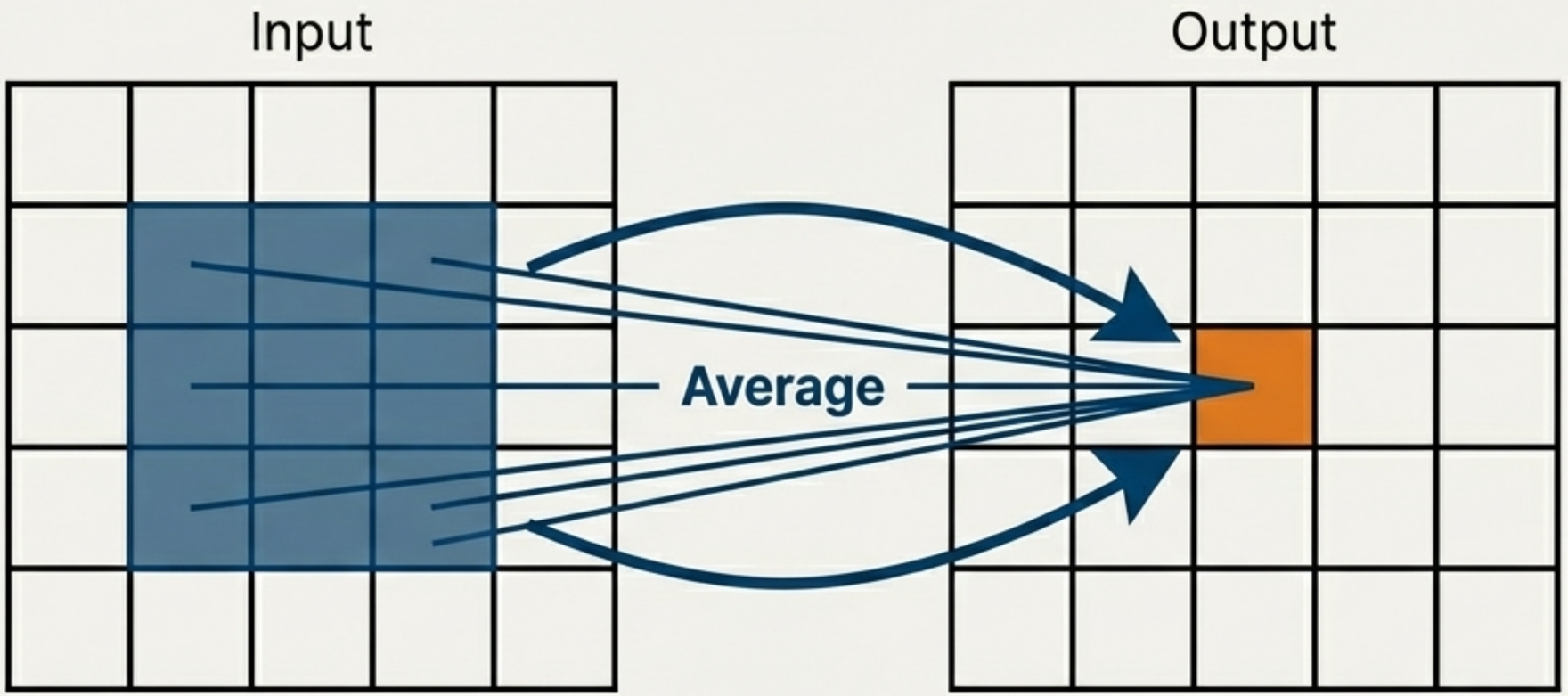
A simple `if` statement ensures that only **threads corresponding to valid pixels** perform any work.

```
__global__
void rgbToGray_kernel(...) {
    // 1. Calculate global row and column
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // 2. The Boundary Check (Guard)
    if (row < height && col < width) {
        // 3. Convert to 1D index
        int index = row * width + col;

        // 4. Perform the computation
        gray[index] = 0.3f * red[index] + 0.6f * green[index] + 0.1f * blue[index];
    }
}
```

Result: GPU time of 1.5ms vs. CPU time of 3.8ms (**>2x speedup** including memory copies). The kernel execution alone is over **150x** faster (0.063ms)!

Leveling Up: Image Blurring with Stencil Operations

To blur an image, each output pixel becomes the average of its corresponding input pixel and its immediate neighbors.



The strategy remains similar: **Assign one thread to compute each *output* pixel.**
The key difference: each thread now needs to read **multiple input pixels**.

Implementing the Blur Kernel: A Deceptive Bug

Each thread calculates its output (out_row, out_col) and then loops over a 3×3 neighborhood of input pixels (in_row, in_col).

```
// DANGER: Using unsigned int here!
#E67E22#unsigned int out_row = blockIdx.y * blockDim.y + threadIdx.y;
#E67E22#unsigned int out_col = blockIdx.x * blockDim.x + threadIdx.x;

if (out_row < height && out_col < width) {
    float sum = 0.0f;
    for (int r = -1; r <= 1; r++) {
        for (int c = -1; c <= 1; c++) {
            // This can be negative!
            int in_row = out_row + r;
            // ... sum up pixels ...
        }
    }
    // ...
}
```

What happens when #E67out_row# is 0 and we calculate #E67E2#out_row - 1#22??? With #E67Eunsigned int, this doesn't become #E67--1; it wraps around to a very large positive number, causing incorrect memory access.

The Rule of Thumb for Safe Memory Access

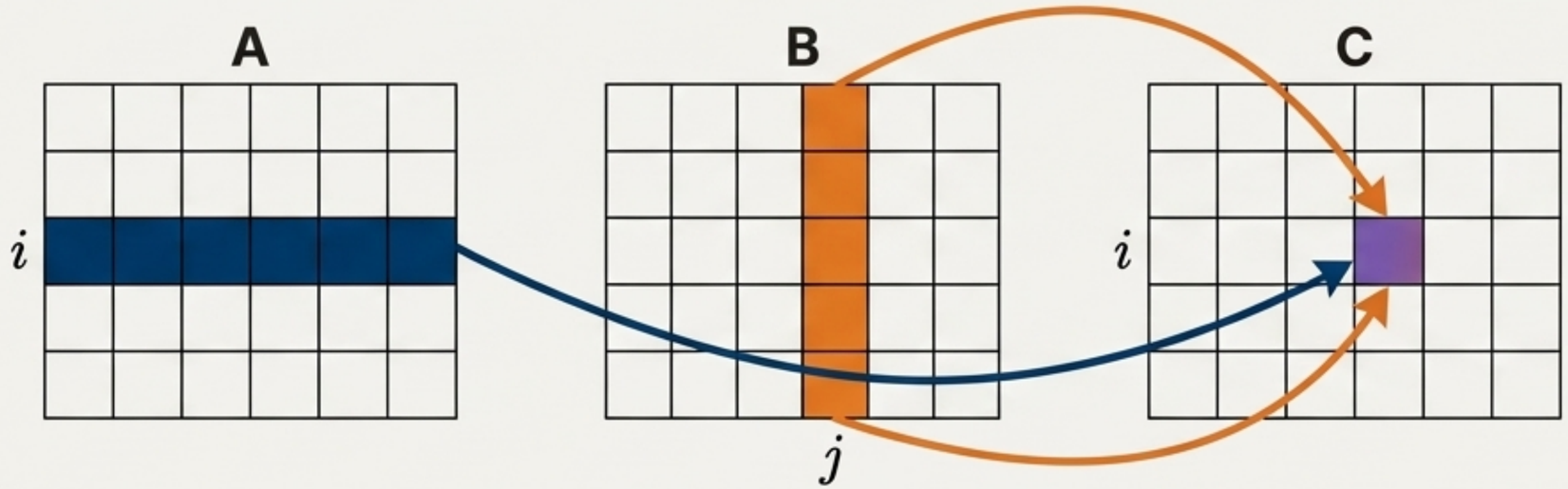
Every memory access must have a corresponding guard that compares its index to the array dimensions.

```
// Use 'int', not 'unsigned int'
int out_row = ...;
int out_col = ...;

// CHECK 1: Is the OUTPUT pixel in bounds?
if (out_row < height && out_col < width) {
    float sum = 0.0f;
    int count = 0;
    for (int r = ...; ...) {
        for (int c = ...; ...) {
            int in_row = out_row + r;
            int in_col = out_col + c;

            // CHECK 2: Is the INPUT pixel we are reading in bounds?
            if (in_row >= 0 && in_row < height &&
                in_col >= 0 && in_col < width) {
                sum += input[in_row * width + in_col];
                count++;
            }
        }
    }
    output[...] = sum / count;
}
```

Stencil operations often require two sets of guards: one for the final ***write*** location and another inside the loop for each ***read*** location.

The Final Ascent: High-Intensity Computing with Matrix Multiplication

Compute `C = A * B`. Each element `C(row, col)` is the dot product of row `row` from matrix A and column `col` from matrix B.



$$C(i,j) = \sum A(i,k) * B(k,j)$$

The Parallel Strategy

Again, we assign **one thread to compute each element of the output matrix C**.
Each thread will perform an entire dot product calculation.

A Simple and Powerful Matrix Multiplication Kernel

```
__global__
void matmul_kernel(float* A, float* B, float* C, int N) {
  // Same 2D index calculation as before
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < N && col < N) {
    float sum = 0.0f;
    // Dot product calculation
    for (int i = 0; i < N; ++i) {
      sum += A[row * N + i] * B[i * N + col];
    }
    C[row * N + col] = sum;
  }
}
```

Accesses elements contiguously across a row of A.

Accesses elements strided down a column of B.

~50x Speedup

(incl. memory copy)

~100x Speedup

(kernel only)

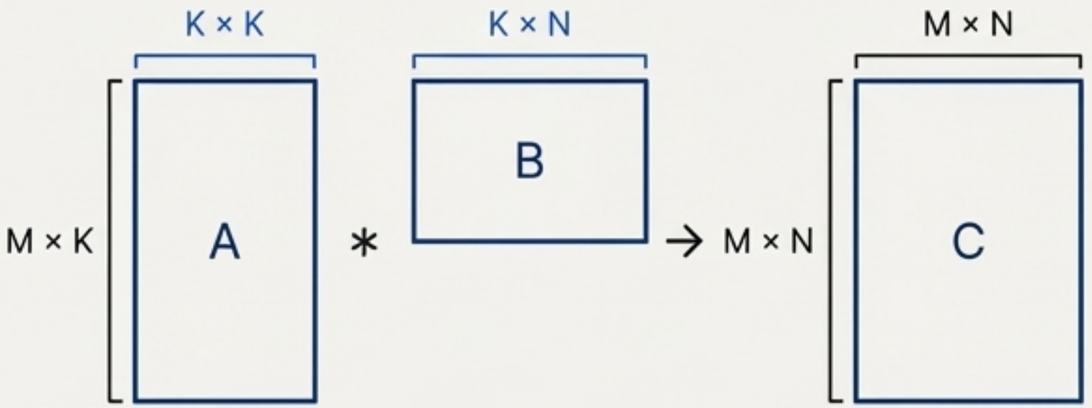
Higher arithmetic intensity (more calculations per memory access) allows the GPU to achieve massive performance gains.

Your Next Challenge: The Road Ahead

Our examples used square matrices and simplified boundary checks. Real-world problems require more robustness.

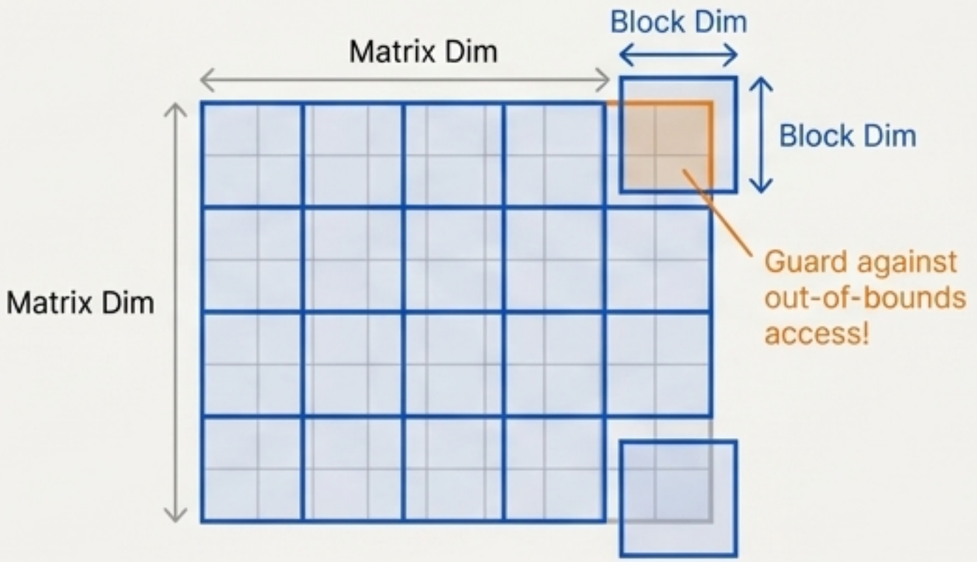
1. General Matrix Dimensions

How do you handle multiplying a non-square M x K matrix by a K x N matrix to get an M x N result?



2. Robust Boundary Conditions

How do you apply the ‘guarding’ principle to the MatMul kernel, especially when matrix dimensions are not perfectly divisible by block dimensions?



You will implement these features in your next assignment. The concepts from today are your foundation for solving it.