ChatGPT

CUDA Matrix Multiplication Code Explained

Host and device memory allocation: First the code sets up space for the matrices on both the CPU (host) and GPU (device). On the host it allocates or declares arrays for A, B and C normally (e.g. `float *A = new float[N*N];`). On the device it uses `cudaMalloc` to allocate GPU memory for each matrix, for example:

```
cudaMalloc((void**)&d_A, N*N*sizeof(float));
cudaMalloc((void**)&d_B, N*N*sizeof(float));
cudaMalloc((void**)&d_C, N*N*sizeof(float));
```

This reserves `N*N*sizeof(float)` bytes on the GPU for each matrix pointer (`d_A`, `d_B`, `d_C`) [1].

- **Host arrays:** Regular CPU memory for A, B, C.
- **Device arrays:** GPU memory allocated by `cudaMalloc`.

Data transfer (Host↔Device): After allocation, the code copies the input matrices A and B from host to device with `cudaMemcpy`. For example:

```
cudaMemcpy(d_A, A, N*N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, N*N*sizeof(float), cudaMemcpyHostToDevice);
```

This transfers the contents of A and B into `d_A`, `d_B` on the GPU [1]. After the kernel finishes, it copies the result back:

```
cudaMemcpy(C, d_C, N*N*sizeof(float), cudaMemcpyDeviceToHost);
```

copying `d_C` into the host array C [2]. These calls specify the direction (`cudaMemcpyHostToDevice` or `DeviceToHost`) and the size of the data to move.

Grid and block setup: The code defines a grid of thread blocks to cover all elements of the output matrix C. For example:

```
dim3 dimBlock(1, 1);
dim3 dimGrid(N, N);
matrixMul<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
```

Here `dimBlock(1,1)` means each block has 1×1 = 1 thread (inefficient but simple), and `dimGrid(N,N)` means there are N×N blocks. In total this launches N^2 threads. (All threads run the kernel in parallel [3].)

Each thread handles one element C[i][j]. The actual code may choose different block sizes (e.g. 16×16 threads) to improve efficiency, but the idea is the same.

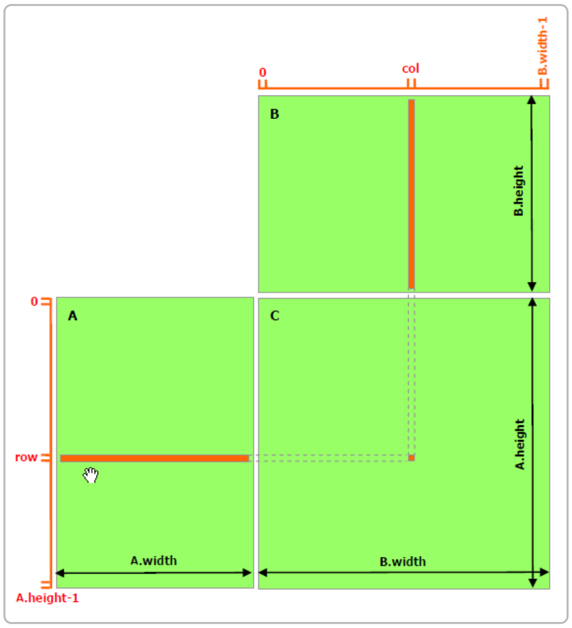
- **Blocks per grid:** e.g. `(N, N)` so there are N×N blocks.
- **Threads per block:** e.g. `(1, 1)` here (just one thread per block) [4].
- **Kernel launch:** The `<<<dimGrid, dimBlock>>>` syntax tells CUDA how many threads to launch [5].

CUDA kernel function: The kernel is a `__global__` function that runs on the GPU. In CUDA C++, `__global__` marks a function as a kernel callable from host code [5]. For example:

```
__global__ void matrixMul(float *A, float *B, float *C, int N) {
    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;
        for (int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + col];
        }
        C[row*N + col] = sum;
    }
}
```

[5] [6]

- The lines `int row = blockIdx.y*blockDim.y + threadIdx.y;` and similarly for `col` compute the global row and column index for this thread [6]. Each thread knows its block coordinates (`blockIdx`) and its thread coordinates within the block (`threadIdx`), so multiplying by `blockDim` (the block size) gives the overall index.
- The `if (row < N && col < N)` check ensures threads outside the matrix bounds do nothing.
- The loop `for (int k = 0; k < N; ++k)` computes the dot product of row `row` of A with column `col` of B, accumulating into `sum`. Finally `sum` is written to `C[row*N + col]`. This means **each thread computes one element of C** [6].



The image above shows this concept: each GPU thread takes one row from A and one column from B, multiplies element-wise and sums to produce the corresponding element of C [6]. In code, the lines inside the loop (`sum += A[row*N + k] * B[k*N + col]`) and the write `C[row*N + col] = sum;` implement this dot-product [6].

CUDA-specific terms:
- `__global__`: A function qualifier that marks the kernel function. It means *this function runs on the GPU and is called from the CPU*. Each call to a `__global__` function launches many parallel threads [5].
- `blockIdx`, `threadIdx`: Built-in 3-component vectors giving the block and thread indices. For a 2D grid/block, `blockIdx.x`/`blockIdx.y` are the block's (x,y) coordinates in the grid, and `threadIdx.x`/`threadIdx.y` are the thread's coordinates within its block [6].
- `blockDim`: A 3-component vector giving the dimensions of each block. For example, if `dim3 dimBlock(16,16)`, then `blockDim.x=16`, `blockDim.y=16`. Multiplying `blockIdx * blockDim + threadIdx` converts local (block,thread) indices to global indices [6].

These built-in variables let each thread compute its unique position. For example, `int row = blockIdx.y * blockDim.y + threadIdx.y;` and `int col = blockIdx.x * blockDim.x + threadIdx.x;` determine which element of C this thread will produce [7].

Parallel execution: When the host calls `matrixMul<<<dimGrid, dimBlock>>>(...)`, CUDA launches **all the threads at once** [3]. In our example with `dimGrid(N,N)` and `dimBlock(1,1)`, there are N×N = N^2 threads. Each of those threads runs the kernel code in parallel, so all elements of C are computed concurrently. In general, the total number of threads is `dimGrid.x * dimGrid.y * dimBlock.x * dimBlock.y`. The CUDA model guarantees that a kernel with many threads (say N threads) executes them in parallel on the GPU [3].

Output printing: After the kernel finishes and the result is copied back, the code typically prints the matrix C from host code. For example, a nested loop on the CPU might do:

```
for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++){
        printf("%f ", C[i*N + j]);
    }
    printf("\n");
}
```

This is just regular C/C++ code, running on the CPU, to display the computed values of C.

Performance considerations: A few important notes on this simple code:

- **Block size:** Using a block of `1×1` (one thread per block) is *very inefficient*. Modern GPUs can have up to 1024 threads per block [8]. Packing more threads into each block (e.g. 16×16) usually gives much better performance by leveraging the hardware's parallelism.
- **Small problem size:** For very small matrices (e.g. N=16), the overhead of launching kernels and copying memory may outweigh the computation time. GPUs shine when many threads do work (large matrices).
- **Memory usage:** This code uses only global memory and a simple loop. More advanced implementations tile the matrices into shared memory blocks to reduce global-memory reads and increase speed [9].

In summary, the code allocates memory on the host and GPU, copies data to the GPU, launches a kernel with a grid of threads, and each thread computes one element of the result matrix in parallel. It then copies the result back and prints it. The CUDA features (`__global__`, `blockIdx`, `threadIdx`, `blockDim`) let the code identify each thread's work and run many threads at once for parallel matrix multiplication [5] [6].

Sources: CUDA Programming Guide and tutorials provide these details, for example showing how `__global__` kernels and thread indices work [5] [6]. The image and code snippets above illustrate the basic idea of matrix multiplication in CUDA.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] **1. Introduction — CUDA C++ Programming Guide**
https://docs.nvidia.com/cuda/cuda-c-programming-guide/