Great! Let's walk through the **concept** and then **step-by-step explanation** of your CUDA matrix multiplication program:

🧠 Concept: Matrix Multiplication on GPU

You are multiplying two matrices `A` and `B` of size **N × N = 1024 × 1024** using the GPU. Here's what matrix multiplication means:

📘 Matrix Multiplication Rule:

For two matrices:

C[i][j] = sum_over_k (A[i][k] * B[k][j])

- `A` is NxN
- `B` is NxN
- `C` is the result matrix, also NxN
- Each value of C is computed by a **dot product** between a row of A and a column of B.

⚡ On GPU:

- Each thread computes one value in matrix `C[i][j]`.
- Threads are organized in blocks, and blocks are organized in a grid.
- You use `blockDim`, `blockIdx`, and `threadIdx` to find out which part of the matrix each thread handles.

🧩 Step-by-Step Explanation of the Code

✅ Includes and Constants

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cuda_runtime.h>

#define N 1024           // Size of matrix (N x N)
#define TILE_WIDTH 16       // Block size (16x16 threads)
```

- `N = 1024`: Your matrices are 1024×1024
- `TILE_WIDTH = 16`: Each block has 16×16 = 256 threads

🔧 CUDA Kernel: matMulKernel

```
__global__ void matMulKernel(const int* A, const int* B, int* C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y; // Calculate row index
    int col = blockIdx.x * blockDim.x + threadIdx.x; // Calculate column index
```

- `blockDim`: dimensions of each block (`16x16`)
- `blockIdx`: index of the block in the grid
- `threadIdx`: thread position inside a block

So:

- `row` and `col` give each thread its target `C[row][col]`

```
    if (row < width && col < width) {
        int sum = 0;
        for (int k = 0; k < width; ++k) {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}
```

- The kernel computes a **dot product** of row of A and column of B.
- Result is stored in `C[row][col]`.

Note: Matrices are stored in **1D arrays**, so:

A[row][k] → A[row * width + k]
B[k][col] → B[k * width + col]

🧠 Host Code (`main()`)

🔹 Allocate and Initialize Matrices

```
int* h_A = (int*)malloc(bytes);
```

```
int* h_B = (int*)malloc(bytes);
int* h_C = (int*)malloc(bytes);

srand((unsigned)time(NULL));
for (int i = 0; i < size; ++i) {
    h_A[i] = rand() % 2;
    h_B[i] = rand() % 2;
}
```

- `h_A`, `h_B`, `h_C`: matrices on host (CPU)
- Filled with random 0s and 1s

🔹 **Allocate GPU Memory**

```
cudaMalloc(&d_A, bytes);
cudaMalloc(&d_B, bytes);
cudaMalloc(&d_C, bytes);
```

- `d_A`, `d_B`, `d_C`: matrices on device (GPU)

🔹 **Copy Data to Device**

```
cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
```

- Transfers data from CPU → GPU

🔹 **Define Grid and Block Dimensions**

```
dim3 blockSize(TILE_WIDTH, TILE_WIDTH);
dim3 gridSize((N + blockSize.x - 1) / blockSize.x,
              (N + blockSize.y - 1) / blockSize.y);
```

- Block: `16x16 = 256` threads
- Grid: Enough blocks to cover the entire 1024x1024 matrix

For example:

gridSize.x = ceil(1024 / 16) = 64

So:

- Total threads = 64 * 64 blocks * 256 threads = 1,048,576 threads (1 per pixel in C)

🔹 Measure Time Using CUDA Events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

// Launch kernel
matMulKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);

// Stop timing
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);
```

- Measures **GPU execution time in milliseconds**

🔹 Copy Result Back to Host

```
cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);
```

- Brings result matrix C from GPU to CPU

🔹 Print Result (15x15)

```
for (int i = 0; i < 15; ++i) {
    for (int j = 0; j < 15; ++j) {
        std::cout << h_C[i * N + j] << " ";
    }
    std::cout << std::endl;
}
```

- Only prints a small portion of C (first 15 rows × 15 columns)

🧹 Clean Up

```
cudaFree(d_A);
cudaFree(d_B);
```

cudaFree(d_C);
free(h_A);
free(h_B);
free(h_C);

- Frees all allocated memory on both CPU and GPU

✅ Output Example

GPU Execution Time: 104.3 ms
Result Matrix C (first 15x15):
1 0 2 3 1 4 2 ...
...

🚀 Summary

- Each **thread** computes **one element** of the output matrix.
- Matrices are stored as **1D arrays**, so indexing uses: `row * width + col`.
- The program is **scalable** and works for any N.
- Uses **CUDA events** for high-resolution timing.
- For even more speed, you could use **shared memory + tiling optimization**, which reduces global memory accesses.

Let me know if you want the optimized version next!