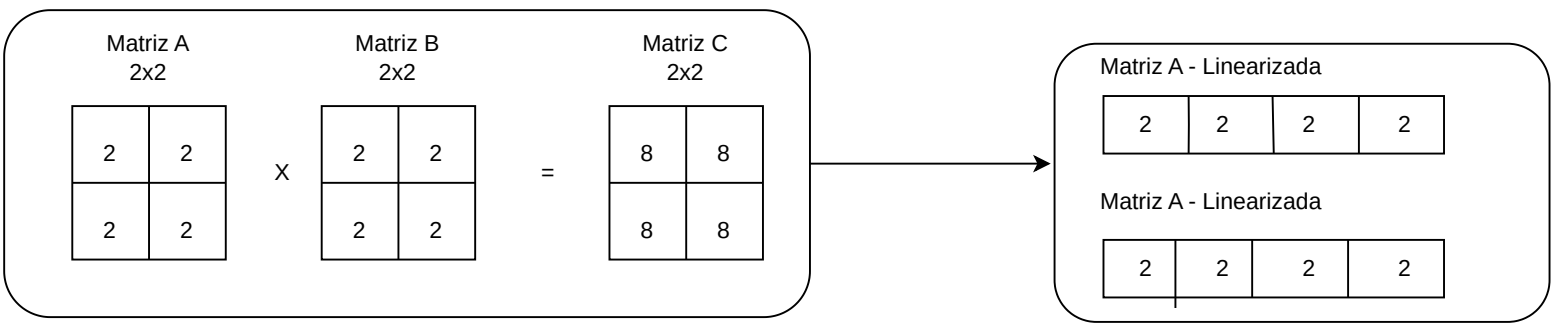
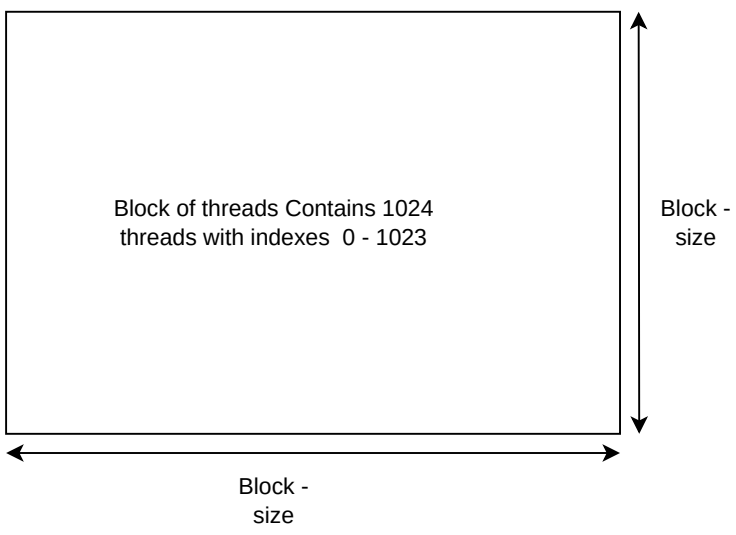
# Atenção:

A manipulação de arrays dentro dos kerneis pode ser complexa, é preciso levar em conta quando lidamos com matrizes, que é mais facil trabalhar com elas de forma linearizada.

## Matrizes Linearizadas são vetores:

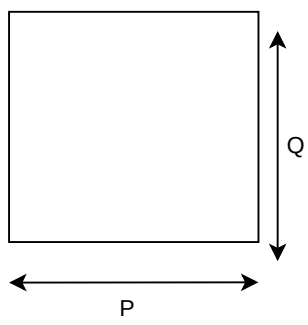



## Resolvendo o problema :

Para resolver o problema dessa forma, precisamos pensar o que cada kernel deve fazer. Se cada kernel for responsável por multiplicar a linha A[0] pela coluna B[i][0], precisamos então de um bloco que contenha 4 threads. Pois são 4 operações dessa forma, para a multiplicação de matrizes quadradas 2x2.

Da multiplicação de matrizes sabemos que a matriz resultante terá o numero de colunas de A , como numero de linhas, e terá o numero de linhas de B como o numero de colunas, chamaremos isso de uma matrix P X Q .

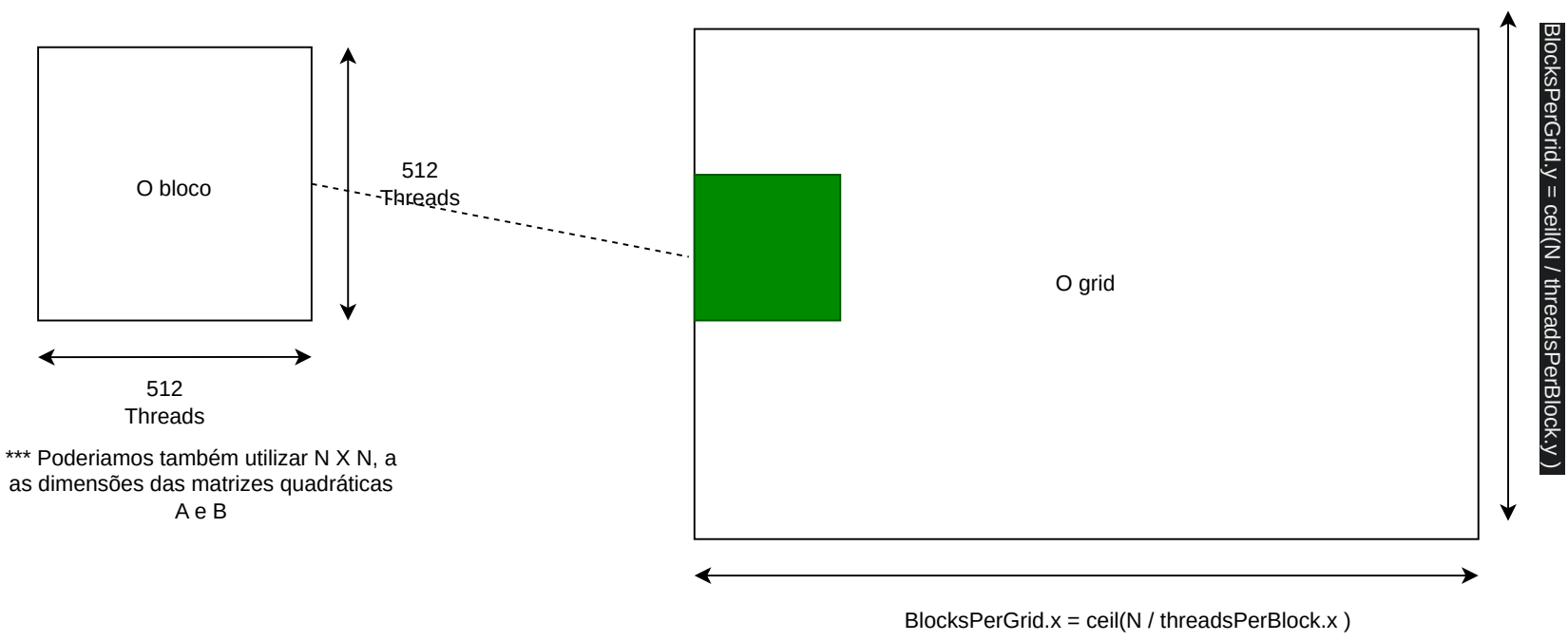



# Definindo o grid e os blocos:

Cada bloco deve ter no maximo 1024, threads. Nesse caso, vamos utilizar 512 threads por bloco. Operando com 50% da sua capacidade. Para calcular o numero de blocos por grid é preciso pensar na dimensão da matriz resultante. Nesse caso, precisamos de blocos com no mínimo PXQ de dimensão.

```
blocksPerGrid.x = ceil(double(N)/double(threadsPerBlock.x));
blocksPerGrid.y = ceil(double(N)/double(threadsPerBlock.y));
```

Atenção : Use sempre ceil, arredonde sempre para cima, é preferivel alocar um bloco que fique ocioso, do que faltar blocos na alocação.




*** Poderiamos também utilizar N X N, a as dimensões das matrizes quadráticas A e B

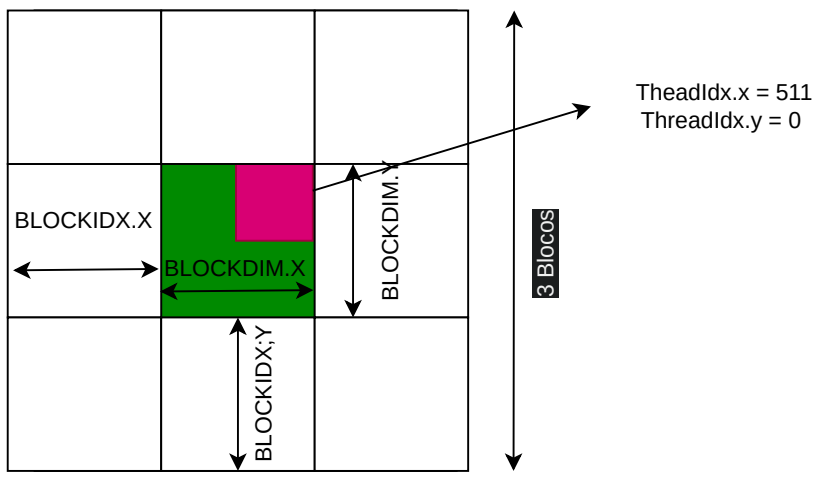
# Definindo o kernel

Uma vez definida as dimensões do grid e dos blocos, deve ser definido o kernel. ( A função que será executada por cada thread) dentro do device. A parte mais complexa de definir o kernel nessa aproach, é compreender como faremos o mapeamento dos indices dentro da matriz linearizada C , em termos das variáveis disponibilizadas pelo cuda, que representam as cordenadas do bloco e da thread bidimensionalmente.

# Identificando thread no grid:

Para identificar uma thread no grid thread[i,j] precisamos entender que ela estára em uma posição *[ threadidx.x , threadIdx.y ]* de um dos blocos que estára na posição [ *blockIdx.x, blockIdx.y* ] . Entao basicamente, primeiro precisamos identificar o bloco, e só então identificar a thread dentro deste bloco, isso pode ser feito pelas variáveis:

```
int ROW = blockIdx.y*blockDim.y+threadIdx.y;
int COL = blockIdx.x*blockDim.x+threadIdx.x;
```




Digamos que N = 1536, e que decidimos que cada bloco, tera 512 threads, na dimensão x e 512 threads na dimensão y. Temos então um grid, 3x3 blocos,

```
int ROW = blockIdx.y*blockDim.y+threadIdx.y;
int COL = blockIdx.x*blockDim.x+threadIdx.x;
```

ROW = 1 * 512 + 0 = 512 + 0 = 512
COL = 1 * 512 + 511 = 1023

Logo chegamos a conclusão que a identificação desse thread dentro do GRID, é [512. 1023]

# Análise do Laço:

```
using namespace std;

__global__ void matrixMultiplicationKernel(float* A, float* B, float* C, int N) {

    int ROW = blockIdx.y*blockDim.y+threadIdx.y;
    int COL = blockIdx.x*blockDim.x+threadIdx.x;

    float tmpSum = 0;

    if (ROW < N && COL < N) {
        // each thread computes one element of the block sub-matrix
        for (int i = 0; i < N; i++) {
            tmpSum += A[ROW * N + i] * B[i * N + COL];
        }
    }
    C[ROW * N + COL] = tmpSum;
}
```

Deixa as threads extras inativas.

Como cada thread é responsável por gerar um elemento do bloco em cada sub matriz, vamos verificar oq a thread [ 512, 1023 ] faria. Nessa execução :

A[ROW * N + i] = A[ 512 * 1536 + i ] = A[ 632832 + i ] , vale lembrar aqui, que a matriz se encontra linearizada, nesse caso, A ´é um vetor, onde a cada 1536 elementos, temos uma nova linha. Nesse caso, estamos processando, toda a linha que começa em 632832 e acaba em 634368 , que correspondem a linha 412 da matriz inicial. Para N = 1536

B[i * N + COL] = B[ i * 1536 + 1023] . A ideia aqui é sempre representar a coluna em B também linearizado, para i = 0 , por exemplo, teremos :

A[632832] * B[ 0 * 1536 + 1023] = A[632832] * B[1023]

i = 1
A[ 632832 + 1 ] * B[ 1 * 1536 + 1023] = A[632833] * B[2559]

Basicamente estamos localizando todos os elementos da coluna 1 , numa matriz linearizada, baseados no seguinte calculo :

Tamanho da linha X indice do elemento relativo á coluna + Indice da coluna.

# Final:

Esta implementação dummy, serve para entender bem como lidar com blocos bimensionais de threads, no contexto de multiplicação de matrizes dentro de uma GPU usando cuda, lembrando que essa é uma implementação dummy, que pode ser substituida com tecnicas mais avançadas, porém serve como um bom exercicio para compreender como funciona o cuda, e a definição de blocos bidimensionais.