UNIVERSIDADE FEDERAL DE SÃO CARLOS
DEPARTAMENTO DE COMPUTAÇÃO

Programação Paralela e Distribuída

Exercício Programa - EP5

Professor: Hermes Senger

Aluno: Ivan Duarte Calvo, 790739

São Carlos, SP, 2023

1 Metodologia

A parte mais importante a ser implementada foi o Kernel CUDA *matrixMulGPU*.

```
1  __global__ void matrixMulGPU( int * a, int * b, int * c )
2  {
3      int row = blockIdx.y*blockDim.y+threadIdx.y;
4      int col = blockIdx.x*blockDim.x+threadIdx.x;
5
6      int val = 0;
7
8      if (row < N && col < N) {
9          for (int i = 0; i < N; i++) {
10             val += a[row * N + i] * b[i * N + col];
11         }
12     }
13     c[row * N + col] = val;
14 }
```

Inicialmente, nas linhas 3 e 4 são calculados os valores de linha(*row*) e coluna(*col*). O cálculo desses valores é importante para transformar uma matriz 2D em um vetor de uma única dimensão. Após isso, é verificado se os valores determinados para *row* e *col* são menores que N, para garantir que a multiplicação seja realizada em posições corretas na matriz. Finalmente é realizado o cálculo principal, na linha 10, e o valor é atribuído para a matriz de resposta.

2 Hardware

O código foi colocado em execução em um ambiente do *Google Colab*, para avaliar qual o hardware disponibilizado pelo ambiente para a realização das tarefas foram utilizados os comandos *nvidia-smi* e *lscpu* para obter informações sobre a GPU e CPU respectivamente.

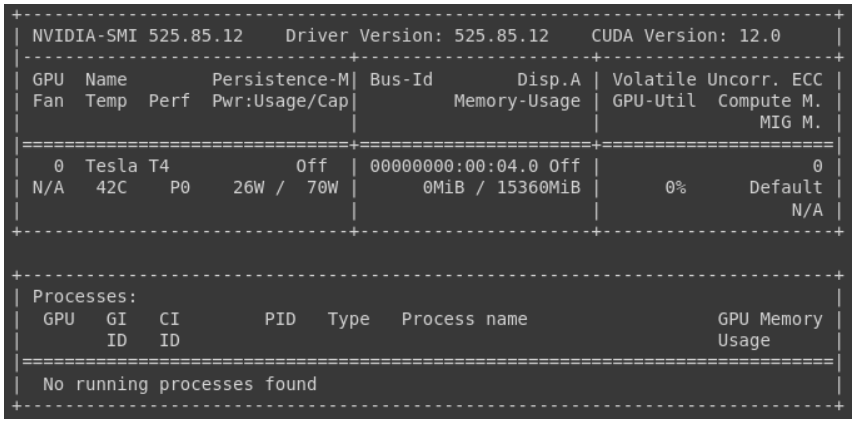

Figura 1: *Output* do comando *nvidia-smi*

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          2
On-line CPU(s) list:             0,1
Thread(s) per core:              2
Core(s) per socket:              1
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           79
Model name:                      Intel(R) Xeon(R) CPU @ 2.20GHz
Stepping:                        0
CPU MHz:                         2199.998
BogoMIPS:                        4399.99
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       32 KiB
L1i cache:                       32 KiB
L2 cache:                        256 KiB
L3 cache:                        55 MiB
NUMA node0 CPU(s):               0,1
```

3 Resultados

Para a execução foram utilizadas matrizes quadradas NxN, com o N tendo os tamanhos: 128, 256, 512, 1024 e 2048. A quantidade de *threads* por bloco foi uma matriz 2D de tamanho (16, 16) e o número de blocos foi calculado através da divisão de N pela quantidade de threads por bloco. Os resultados obtidos foram os seguintes:

É importante salientar que nas tabelas e gráficos com resultados são apresentados valores para *speedup*, no entanto, não o *speedup* calculado não é o mesmo usual, pois nesse caso a quantidade de blocos está aumentando de acordo com o tamanho das matrizes multiplicadas. Portanto o calculo foi feito da seguinte forma:

$$Speedup = \frac{T_{cpu}}{T_{gpu}}$$

Tabela 1: Tempos de execução

Tamanho da matriz	Tempo na CPU	Tempo na GPU	*Speedup*
128	0.006966	0.000565	12.334225
256	0.098065	0.001374	71.361000
512	0.495764	0.002654	186.781904
1024	6.105657	0.013107	465.825727
2048	116.277195	0.086975	1336.910414

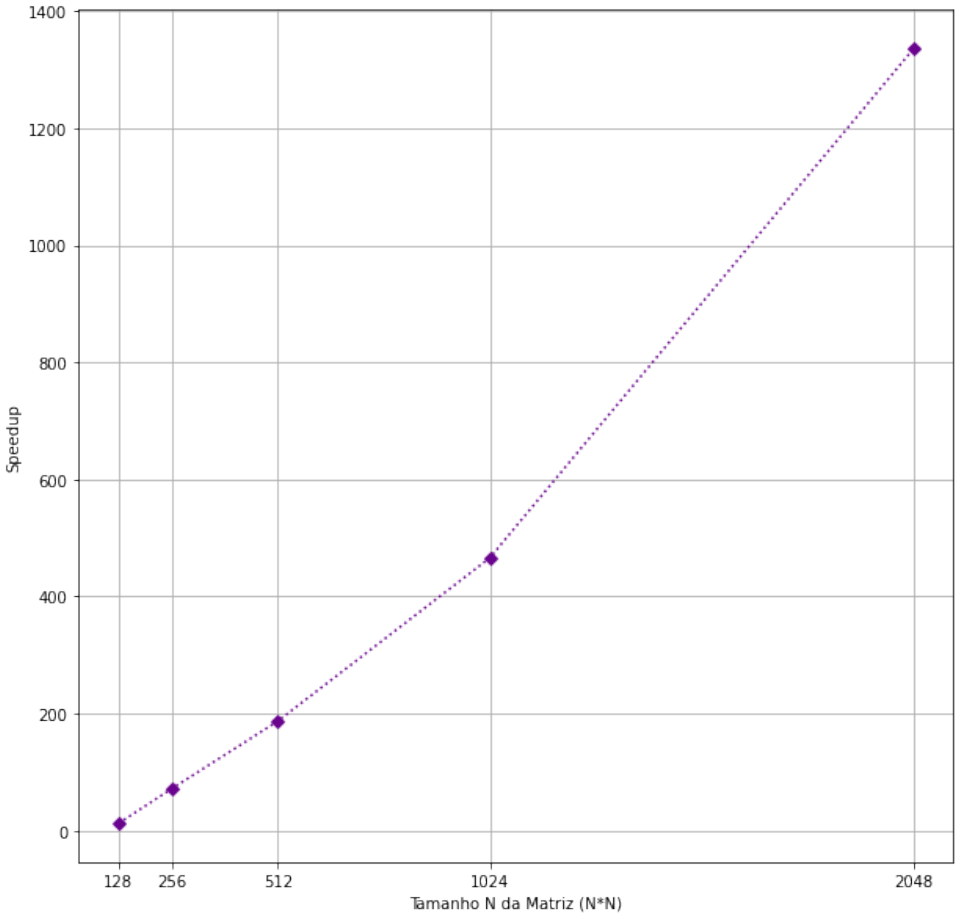

Figura 2: *Speedup* do tempo na GPU em comparação com a GPU

4 Conclusão

Os valores obtidos nos tempos de execução foram extremamente bem sucessidos, indicando a grande vantagem na implementação da paralelização através do CUDA visto que os ganhos nos tempos de execução são muito significativos.

A única possível contraindicação para a utilização seria por conta da alta complexidade para a implementação, contudo, ainda se mostra uma alternativa interessante por conta dos ótimos resultados de maneira que pode se considerar vantajoso superar a barreira inicial do aprendizado necessário.