

Universidade Federal de São Carlos

Centro de Ciências e de Tecnologia

Departamento de Computação

Programação Paralela e Distribuída

Exercício Programa 5 - EP 5

Professor Hermes Senger

Nome: Jhon Wislin Ribeiro Citron

RA: 776852

2º Semestre de 2023

1 - Especificações da CPU e GPU

Para fazer a multiplicação de matriz usando a GPU, foi utilizado os recursos fornecidos pelo google colab em relação a CPU e a GPU. A seguir é possível observar as configurações de ambos dentro do ambiente do google colab. Para visualizar as configurações do processador foi utilizado o comando **lscpu**, que apresenta as informações abaixo. O comando **nvidia-smi** apresenta algumas informações referentes a GPU sendo utilizada, como mostrado na figura abaixo (Figura 1). Link do google colab: https://colab.research.google.com/drive/17mRlKLAe0Or8ZkZSbrZ1Q_RHULyoqZUG?usp=s_haring

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:          46 bits physical, 48 bits virtual
Byte Order:             Little Endian
CPU(s):                 2
On-line CPU(s) list:    0,1
Vendor ID:              GenuineIntel
Model name:             Intel(R) Xeon(R) CPU @ 2.00GHz
CPU family:             6
Model:                 85
Thread(s) per core:     2
Core(s) per socket:     1
Socket(s):              1
Stepping:               3
BogoMIPS:               4000.41
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clf lush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good
nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fm a cx16 pcid
sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm
3dnowprefetch invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 hle avx2 smep
bmi2 erms invpcid rtm mpx avx512f avx512d q rdtseed adx smap clflushopt clwb avx512cd
avx512bw avx512vl xsaveopt xsave c xgetbv1 xsaves arat md_clear arch_capabilities
Virtualization features:
Hypervisor vendor:     KVM
Virtualization type:    full
Caches (sum of all):
L1d:                   32 KiB (1 instance)
L1i:                   32 KiB (1 instance)
L2:                    1 MiB (1 instance)
L3:                   38.5 MiB (1 instance)
NUMA:
NUMA node(s):          1
NUMA node0 CPU(s):     0,1
Vulnerabilities:
Gather data sampling:   Not affected
Itlb multihit:         Not affected
```

L1tf: Mitigation; PTE Inversion
 Mds: Vulnerable; SMT Host state unknown
 Meltdown: Vulnerable
 Mmio stale data: Vulnerable
 Retbleed: Vulnerable
 Spec rstack overflow: Not affected
 Spec store bypass: Vulnerable
 Spectre v1: Vulnerable: __user pointer sanitization and usercopy barriers only; no swap
 Spectre v2: gs barriers Vulnerable, IBPB: disabled, STIBP: disabled, PBRSE-eIBRS: Not affected
 Srbds: Not affected
 Tsx async abort: Vulnerable

Opções: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid xsaveopt arat md_clear flush_l1d arch_capabilities

CPU

NVIDIA-SMI 535.104.05				Driver Version: 535.104.05		CUDA Version: 12.2	
GPU	Name		Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.
							MIG M.
0	Tesla T4		Off	00000000:00:04.0	Off		0
N/A	52C	P8	10W / 70W	0MiB / 15360MiB		0%	Default
							N/A
Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	
	ID	ID				Usage	
No running processes found							

Figura 1 - GPU

2 - Estratégia

De forma geral, para o kernel matrixMulGPU, a estratégia abordada foi utilizar dois loops stride para ajustar os saltos nas posições de linha e coluna da matriz em relação a X e Y na grid. Dessa forma, cada thread X e Y ficou responsável por calcular uma ou mais posições

de C. Os cálculos feitos para as posições X e Y, e saltos stride X e stride Y, estão representados abaixo.

- $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ - Posição global da thread em X.
- $\text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y}$ - Posição global da thread em Y.
- $\text{blockDim.x} * \text{gridDim.x}$ - Número de Threads na grid em X.
- $\text{blockDim.y} * \text{gridDim.y}$ - Número de Threads na grid em Y.

Para a criação do kernel para a multiplicação de matrizes, foi adotada a seguinte estratégia. Como a estrutura do problema definiu que a grid possuía duas dimensões, dentro do kernel foram obtidas inicialmente as posições X e Y globais para cada thread, sendo armazenadas nas variáveis X e Y. Após isso, foram determinados os números de threads em X e Y visando determinar os saltos (strideX e strideY) do nosso problema para os loops stride. Com isso, o loop mais interno foi mantido inalterado de forma que cada thread ficou responsável por calcular uma ou mais posições da matriz C. Os loops externos foram alterados de forma que, row e col começassem nas posições globais de cada thread, onde, a cada interação os loops davam saltos correspondentes ao número de threads em X e Y de forma que uma mesma thread consiga ficar responsável por calcular mais de uma posição de C. Essa abordagem garante que seja possível lidar com qualquer tamanho do problema (Número de elementos na matriz) seja maior, menor ou igual ao número de threads. O kernel desenvolvido se encontra abaixo (Figura 2).

```
__global__ void matrixMulGPU(int *a, int *b, int *c, int N)
{
    int X = blockIdx.y * blockDim.y + threadIdx.y; //Índice global para x na grid
    int Y = blockIdx.x * blockDim.x + threadIdx.x; //Índice global para y na grid
    int strideX = blockDim.x * gridDim.x; //Número de threads em x
    int strideY = blockDim.y * gridDim.y; //Número de threads em y

    for( int row = X; row < N; row+=strideX )
        for( int col = Y; col < N; col+=strideY )
        {
            int val = 0;
            for (int k = 0; k < N; ++k)
            {
                val += a[row * N + k] * b[k * N + col];
            }
            c[row * N + col] = val;
        }
}
```

Figura 2 - Kernel (matrixMulGPU)

3. Execução

Para a execução, foram definidos os tamanhos 10, 100 e 1000 para a matriz. Onde, os seguintes resultados de tempos de execução foram obtidos.

Tamanho	Tempo CPU	Tempo GPU
10 x 10	0,000063	0,529344
100 x 100	0,004822	0,799072
1000 x 1000	3,873425	12,529856

```

✓ 1s [48] !nvcc -arch=sm_75 -o ep5 ep5.cu && ./ep5 10
      GPU Time: 0.529344 seconds
      CPU Time: 0.000063 seconds
      Success!

✓ 2s [49] !nvcc -arch=sm_75 -o ep5 ep5.cu && ./ep5 100
      GPU Time: 0.799072 seconds
      CPU Time: 0.004822 seconds
      Success!

✓ 5s [50] !nvcc -arch=sm_75 -o ep5 ep5.cu && ./ep5 1000
      GPU Time: 12.529856 seconds
      CPU Time: 3.873425 seconds
      Success!

```

Figura 3 - Resultado da execução