Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Sistemas de Computação
Laboratório de Sistemas Distribuídos e de Programação Concorrente

Caderno de Desafio para Programação Paralela

Caderno 09 - Introdução a CUDA II: Organização e uso de memória

por

Guilherme Martins

Paulo Sérgio Lopes de Souza

São Carlos/SP, Brasil, junho de 2020

1. Desafio

1.1. Multiplicação de matrizes

Considere as seguintes matrizes 3*3:

A=

1	2	3
4	5	6
7	8	9

B=

9	8	7
6	5	4
3	2	1

O produto de A * B é:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \times \begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1\times9+2\times6+3\times3 & 1\times8+2\times5+3\times2 & 1\times7+2\times4+3\times1 \\ 4\times9+5\times6+6\times3 & 4\times8+5\times5+6\times2 & 4\times7+5\times4+6\times1 \\ 7\times9+8\times6+9\times3 & 7\times8+8\times5+9\times2 & 7\times7+8\times4+9\times1 \end{pmatrix} = \begin{pmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{pmatrix}$$

A matriz C= A*B, é portanto:

30	24	18
84	69	54
138	114	90

Faça um programa em CUDA C, considerando os conceitos vistos, para calcular o produto de duas matrizes quadradas do tipo ***double***. Utilize as memórias global, compartilhada e local de maneira eficiente na sua solução.

Considere como entrada um arquivo de texto contendo, na primeira linha, a dimensão *dim* das matrizes. A partir da segunda linha e até a linha *dim+1*, estão os elementos da matriz A, do tipo ***double***, onde as linhas são separadas por uma quebra de linha simples e as

colunas por um único espaço. A partir da linha 2*dim+1 até o fim do arquivo, estão os elementos da matriz B, também do tipo *double* e de mesma dimensão. As matrizes devem ser lidas por meio do redirecionamento de fluxo de entrada (***stdin***), ou seja, não é necessário usar ponteiros para arquivos.

Conteúdo do arquivo de entrada, por exemplo, ***entrada.txt***:
3
1.0 2.0 3.0
4.0 5.0 6.0
7.0 8.0 9.0
9.0 8.0 7.0
6.0 5.0 4.0
3.0 2.0 1.0

Para executar no ***bash***, por exemplo, utilize este padrão:
.\mm-cuda < entrada.txt <enter>

Obs: na linha de comando acima, considera-se que o programa foi inserido em ***mm-cuda.cu*** e o executável chama-se ***mm-cuda*** e está no diretório atual.

A saída deve ser impressa, utilizando o *output (**stdout**)* padrão, apenas com os elementos correspondentes a matriz *C,* resultante após o cálculo do produto das matrizes *A* e *B*. Há um espaço a mais no final da linha. Cada métrica é separada por uma quebra de linha simples. Há também uma quebra de linha extra no fim da impressão.

Saída:
30.0 24.0 18.0

84.0 69.0 54.0

138.0 114.0 90.0

2. O que você precisa saber para resolver o desafio

As memórias em CUDA podem ser divididas em local à thread, compartilhada às threads do bloco, global entre blocos ou grades, unificada entre *host* e *device* e constante para valores fixos no *device*. A seguir detalharemos os principais aspectos destas memórias. A Figura 2.1 ilustra as memórias disponíveis (Kirk & Hwu, 2016):

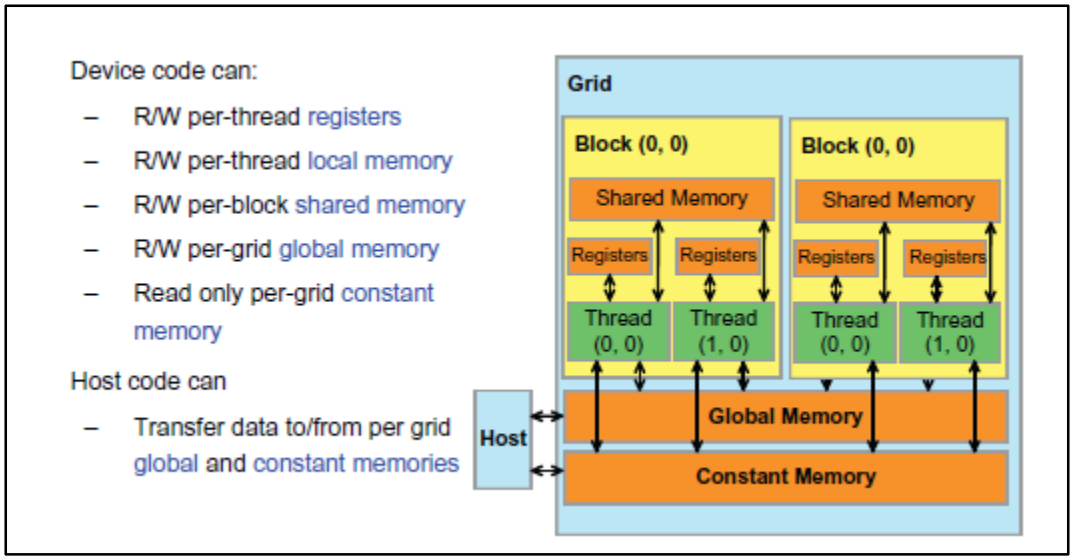

Figura 2.1- Resumo do modelo de memória CUDA (Kirk & Hwu, 2016).

2.1. Memória local

Em *CUDA*, as variáveis declaradas dentro de um *kernel* são locais às *threads* que executam o *kernel*. Isto significa que, as variáveis declaradas dentro de funções do tipo *__global__* ou *__device__* possuem escopo local à *thread* que está em execução e tempo de vida limitado pelo tempo de execução da *thread*, isto é, quando a *thread* termina, todas as suas variáveis locais são destruídas. As variáveis que não são *arrays*, se houver espaço disponível, são armazenadas em registradores e os *arrays* são armazenados em memória local à *thread.* Variáveis armazenadas em memória local possuem, portanto, menor tempo de acesso quando comparadas às variáveis armazenadas em outros tipos de memória. O tamanho da memória local é consideravelmente menor do que o tamanho das demais memórias.

No exemplo abaixo as variáveis *tam, i* e *j* são locais. Isto significa que cada *thread* possui uma cópia, e seu conteúdo só pode ser acessado pela própria *thread*. As variáveis *matrizA, matrizB e matrizC* estão armazenadas em memória global, pois todas foram alocadas na memória do *device* com a primitiva *cudaMalloc* e passadas como argumento de entrada para o *kernel*. O tempo de acesso do conteúdo das variáveis *i* e *j* é consideravelmente mais rápido do que as demais.

```
#define TAM 100
__global__ void soma(int *matrizA, int *matrizB,int *matrizC, int tam){
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   int j= blockDim.y * blockIdx.y + threadIdx.y;

   if (i < tam && j < tam)
   {
      matrizC[i*TAM+j]=matrizA[i*TAM+j]+matrizB[i*TAM+j];
   }
}
```

2.2. Memória compartilhada

A memória compartilhada pode ser acessada por todas as threads de um bloco e é um eficiente mecanismo para as threads interagirem, caso precisem. As memórias compartilhadas são implementadas com uma latência menor e uma largura de banda muito maior, quando comparadas com a memória global. Por esse motivo as variáveis compartilhadas também são usadas para manter dados que são frequentemente utilizados em um determinado momento pelo *kernel*. As variáveis compartilhadas são precedidas pela palavra chave __*shared*__ e o seu escopo está limitado ao escopo do bloco da thread. O tempo de vida de uma variável compartilhada é o mesmo do *kernel*.

É possível alocar variáveis em memória compartilhada de duas formas:

estática: Basta declarar uma variável com o modificador __*shared*__.

dinâmica: A variável deve ser declarada como *extern* __*shared*__ e um parâmetro extra deve ser informado na passagem do *kernel*, que constitui no tamanho da alocação, em *bytes*.

O exemplo abaixo ilustra a alocação dinâmica de shared.

Considere a soma de dois vetores em CUDA, com o uso de memória compartilhada. Neste caso, o vetor C é alocado dinamicamente em memória compartilhada, enquanto os outros dois vetores utilizam apenas memória global. Note a declaração do vetorC_dev no kernel soma com a notação extern __shared__.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda_runtime.h>

__global__ void soma(int *vetorA, int *vetorB, int *vetorC, int tam){
   //Declara o vetorC _dev em memória compartilhada, alocado dinamicamente
  // tam aqui determina o nr itens do vetor, nao de bytes

   extern __shared__ int vetorC_dev[ ];
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if (idx < tam)
   {
      //Calcula a soma dos vetores
      vetorC_dev[idx]=vetorA[idx]+vetorB[idx];
      /*
      Faz o vetor C, armazenado em memória global,
      receber o elemento do vetorC_dev (compartilhado) na posição idx
      */
      vetorC[idx]=vetorC_dev[idx];
   }
}

int main(int argc,char **argv){
   //declara os vetores no host
   int i,*vetorA,*vetorB,*vetorC,threadsPerBlock,blocksPerGrid;
   //Declara os vetores no device
   int *vetorA_d,*vetorB_d,*vetorC_d;

   //define o tamanho dos vetores
   int tam=5000;

   //Define a quantidade de threads por bloco
   threadsPerBlock = 256;

   //Aloca os vetores no host
   vetorA=(int *)malloc(tam * sizeof(int));
   vetorB=(int *)malloc(tam * sizeof(int));
   vetorC=(int *)malloc(tam * sizeof(int));

   //Aloca os vetores no device
   cudaMalloc((void**)&vetorA_d,tam*(sizeof(int)));
   cudaMalloc((void**)&vetorB_d,tam*(sizeof(int)));
   cudaMalloc((void**)&vetorC_d,tam*(sizeof(int)));
```

```
    //Preenche os vetores no host
    for(i=0;i<tam;i++){
       vetorA[i]=i;
       vetorB[i]=-i;
    }

    //Define a quantidade de blocos por grade
    blocksPerGrid=(tam+threadsPerBlock-1)/threadsPerBlock;

    //Copia o conteúdo dos vetores para o device

    cudaMemcpy(vetorA_d,vetorA,tam*(sizeof(int)), cudaMemcpyHostToDevice);
    cudaMemcpy(vetorB_d,vetorB,tam*(sizeof(int)), cudaMemcpyHostToDevice);

    /*
    Invoca o kernel com blocksPerGrid blocos e threadsPerBlock threads,
    passando o tamanho do vetor como argumento (tam*sizeof(int))
    */
    soma <<<blocksPerGrid, threadsPerBlock, tam*sizeof(int)>>>
(vetorA_d,vetorB_d,vetorC_d, tam);

    //Copia o resultado da soma de volta para o host
    cudaMemcpy(vetorC, vetorC_d, tam*(sizeof(int)), cudaMemcpyDeviceToHost);

    //Imprime o resultado no host
    for(i=0;i<tam;i++){
      printf("%d ",vetorC[i]);
    }

    //Desaloca os vetores no host
    free(vetorA);
    free(vetorB);
    free(vetorC);

    //Desaloca os vetores no device
    cudaFree(vetorA_d);
    cudaFree(vetorB_d);
    cudaFree(vetorC_d);
}
```

Exemplo:

Considere a inversão de um vetor de inteiros com 64 posições em CUDA. O código a seguir ilustra a utilização de memória compartilhada estaticamente e dinamicamente alocada, produzindo o mesmo resultado (vetor invertido). Observe o uso da primitiva __*syncthreads*__ para garantir a semântica da operação.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda_runtime.h>

__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];
  int *d_d;

  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }
```

```
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

  staticReverse<<<1,n>>>(d_d, n);

  cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);

  // run dynamic shared memory version
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);

  dynamicReverse<<<1, n, n*sizeof(int)>>> (d_d, n);

  cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

Fonte: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

2.3. Memória global

A memória global possui o maior tamanho, quando comparada com as demais memórias de uma *GPU*, porém a velocidade de seu acesso é o mais lento. As variáveis armazenadas em memória global podem ser acessadas por qualquer *thread* no *grid* e seu tempo de vida é limitado pelo tempo de execução da aplicação, ou seja, as variáveis existem durante toda a execução do código. Qualquer variável declarada com o modificador *__device__*, variáveis globais do código (variáveis declaradas fora das funções ou com a diretiva *#define* em C), variáveis passadas como argumento das funções de *kernel* e variáveis alocadas dinamicamente com o uso do *cudaMalloc* são armazenadas em memória global. O uso frequente de memória global do *device*, pode afetar negativamente o desempenho de uma aplicação *CUDA*.

2.4. Memória constante

A memória constante é um tipo de memória somente leitura do *device*. Trata-se de uma memória com maior velocidade de acesso, quando comparada com a memória global do *device*. O escopo da memória constante é o *grid*, assim todas as *threads* de um *grid* podem acessar o conteúdo de uma variável armazenada em memória constante. O tempo de vida

das variáveis constantes é limitado pelo tempo de execução da aplicação. As *GPUs Nvidia* também oferecem a partir de *8 KB* (variando de acordo com a arquitetura) de *cache* de memória constante por *stream multiprocessor (SM)*, além de permitir operações de *broadcast* de um mesmo valor para todas as *threads* de um *warp*. Entretanto, o tamanho da memória constante do *device* é bastante limitado, tendo em torno de *64 KB* (também variando de acordo com a arquitetura da *GPU*). Para declarar uma variável em memória constante no *device*, utiliza-se o operador *__constant__*, como neste exemplo: ***__constant__ int naomuda=100***. Esta instrução declara uma variável de nome ***naomuda*** do tipo inteiro (***int***) contendo o valor ***100***. O valor atribuído à variável não pode ser modificado pelo *device*; porém, o *host* pode modificar o conteúdo de ***naomuda*** por meio da função ***cudaMemcpyToSymbol***. Tal função possui a seguinte sintaxe:

```
cudaError_t cudaMemcpyToSymbol      (
        const char *   symbol,
        const void *   src,
        size_t count,
        size_t offset,
        enum cudaMemcpyKind     kind
)
```

Onde: ***symbol*** é o nome da variável no *device*, ***src*** é o endereço de memória da origem dos dados no *host*, ***count*** é a quantidade de dados a serem copiados, ***offset*** é o deslocamento em bytes no destino e ***kind*** é o tipo da transferência de memória. Os dois últimos parâmetros (*offset e kind*) possuem valores pré-definidos, que são 0 e *cudaMemcpyHostToDevice,* respectivamente. Assim a função pode ser invocada com apenas três argumentos.

Não é possível declarar variáveis *__constant__* dentro de uma função de *kernel*, pois o escopo da variável não é local à *thread.*

Exemplo:

Considere o produto de um escalar por um vetor, ambos do tipo *int*. O código a seguir ilustra a alocação do escalar em memória constante do *device*. Observe que o valor do escalar é lido do terminal no *host* e armazenado na variável *escalar_h.* Com a função *cudaMemcpyToSymbol*, o valor da variável *escalar_d*, armazenada na memória constante do *device* é definido como o valor lido do terminal.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda_runtime.h>

#define TAM 100

__device__ __constant__ int escalar_d;

__global__ void mult(int *vetorA,int tam){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < tam)
    {
        vetorA[idx]=escalar_d*vetorA[idx];
    }
}

int main(int argc,char **argv){
    int i,*vetorA,threadsPerBlock,blocksPerGrid,escalar_h;
    int *vetorA_d;

    escalar_h=atoi(argv[1]);

    //Define a quantidade de threads por bloco
    threadsPerBlock = 256;

    //Aloca o vetor no host
    vetorA=(int *)malloc(TAM * sizeof(int));

    //Aloca o vetor no device
    cudaMalloc((void**)&vetorA_d,TAM*(sizeof(int)));

    //Preenche o vetor no host
    for(i=0;i<TAM;i++){
        vetorA[i]=i;
    }

    //Define a quantidade de blocos por grade
    blocksPerGrid=(TAM+threadsPerBlock-1)/threadsPerBlock;

    //Copia o conteúdo do vetor para o device

    cudaMemcpy(vetorA_d,vetorA,TAM*(sizeof(int)), cudaMemcpyHostToDevice);
```

```
   //Copia o conteúdo de escalar_h, lido do terminal, para a variável constante escalar_d, no device
   cudaMemcpyToSymbol(escalar_d, &escalar_h, sizeof(int));

   //Invoca o kernel com blocksPerGrid blocos e threadsPerBlock threads

   mult <<<blocksPerGrid,threadsPerBlock>>> (vetorA_d,TAM);

   //Copia o resultado da soma de volta para o host
   cudaMemcpy(vetorA,vetorA_d,TAM*(sizeof(int)), cudaMemcpyDeviceToHost);

   //Imprime o resultado no host
   for(i=0;i<TAM;i++){
     printf("%d ",vetorA[i]);
   }

   //Desaloca os vetores no host
   free(vetorA);

   //Desaloca os vetores no device
   cudaFree(vetorA_d);

}
```

2.5. Memória unificada

Memória unificada foi introduzida na versão 6.0 do kit de desenvolvimento *CUDA*. Trata-se de uma abstração que permite a cópia implícita de memória entre o *host* e o *device*. O uso de memória unificada não reduz o tempo de execução de um programa, pois as transferências de memória continuam ocorrendo de maneira transparente para o programador. A principal vantagem do uso de memória unificada é simplificar a programação em CUDA, eliminando a necessidade de uso da primitiva *CudaMemcpy* e suas variantes.

Na terminologia *CUDA*, memória gerenciada (*managed memory*) é o mesmo que memória unificada. A origem do termo deriva-se do fato de que a memória é alocada simultaneamente na *CPU* e *GPU* e é gerenciada (*managed*) pelo *driver* do *host* usando um único ponteiro.

É possível alocar variáveis na memória unificada de duas formas distintas:

Estática: Declarando uma variável global do tipo __*managed*__. Deve-se declarar uma variável *managed* dentro de uma função executada no *host*.

Dinâmica: Utilizando a função *cudaMallocManaged* que, segundo a literatura, deve ser invocada no *host*. A sua sintaxe é:

*cudaError_t cudaMallocManaged (const char ** ptr, size_t size, unsigned flag)*

Neste caso **ptr** é o endereço de memória para alocação tanto no *host* quanto no *device,* ***size*** é o tamanho, em bytes, da alocação, e ***flag*** indica o tipo da alocação que pode ser: **cudaMemAttachGlobal** (default, onde a memória alocada é acessível para qualquer *kernel* em execução, e **cudaMemAttach*Host*** (a memória alocada só é acessível para os *kernels* lançados pela *thread* que fez a alocação).

Por que a *flag cudaMemAttachGlobal* é default, a função *cudaMallocManaged pode ser* invocada com somente dois parâmetros.

O exemplo abaixo ilustra o uso da memória unificada, o qual calcula a soma de dois vetores com o uso de memória unificada. Não é necessário usar *CudaMemcpy,* pois *cudaMallocManaged* aloca a memória tanto no *host* quanto no *device*, que é gerenciada pelo *driver* do *host*.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda_runtime.h>

__global__ void soma(int *vetorA, int *vetorB,int *vetorC,int tam){
   int idx = blockDim.x * blockIdx.x + threadIdx.x;
   if (idx < tam)
   {
      vetorC[idx]=vetorA[idx]+vetorB[idx];
   }
}

int main(int argc,char **argv){
   int i,*vetorA,*vetorB,*vetorC,threadsPerBlock,blocksPerGrid;

   //Define o tamanho do vetor
   int tam=5000;

   //Define a quantidade de threads por bloco
   threadsPerBlock = 256;

   //Aloca os vetores no host e no device
```

```
    cudaMallocManaged((void**)&vetorA,tam*(sizeof(int)));
    cudaMallocManaged((void**)&vetorB,tam*(sizeof(int)));
    cudaMallocManaged((void**)&vetorC,tam*(sizeof(int)));

    //Preenche os vetores no host
    for(i=0;i<tam;i++){
        vetorA[i]=i;
        vetorB[i]=-i;
    }

    //Define a quantidade de blocos por grade
    blocksPerGrid=(tam+threadsPerBlock-1)/threadsPerBlock;

    //Invoca o kernel com blocksPerGrid blocos e threadsPerBlock threads
    soma <<<blocksPerGrid,threadsPerBlock>>> (vetorA, vetorB, vetorC, tam);

    //Sincroniza as threads do device para impressão do resultado
    cudaDeviceSynchronize();

    //Imprime o resultado no host
    for(i=0;i<tam;i++){
      printf("%d ",vetorC[i]);
    }

    //Desaloca os vetores no device
    cudaFree(vetorA);
    cudaFree(vetorB);
    cudaFree(vetorC);
}
```

2.6. Impacto dos tipos de memória no desempenho

Para ilustrar a importância desses diferentes tipos de memória CUDA no desempenho das aplicações paralelas, mesmo que de maneira mais superficial, observe o desempenho obtido com o programa que faz a soma de vetores em três diferentes GPUs e usando memórias constante, local, compartilhada e global (Tabelas 1, 2 e 3). Os testes realizados também analisaram o desempenho da memória unificada em relação às demais, principalmente em relação à global, visto que a alocação feita na memória unificada é também baseada nesta memória global do *device*.

GeForce GTX650 (Kepler, GDDR5, 1GB, 384 cuda cores). Ubuntu 18.04.		
Nome	Tempo de execução do *kernel* (ms)	Tempo de execução total (ms)
soma_vet_global	10,019	583,933
soma_vet_managed	74,209	697,400
soma_vet_const	nulo	nulo
soma_vet_shared	0,006	533,000
soma_vet_local	8,259	262,233

Tabela 1: Soma de três vetores de 50.000.000 de inteiros em Cuda. (GTX650)

GeForce 940MX (Maxwell, GDDR5, 4GB, 384 cuda cores). Linux Mint 19.		
Nome	Tempo de execução do *kernel* (ms)	Tempo de execução total (ms)
soma_vet_global	17,745	831,300
soma_vet_managed	255,275	1079,333
soma_vet_const	nulo	nulo
soma_vet_shared	0,009	819,633
soma_vet_local	17,027	150,367

Tabela 2: Soma de três vetores de 50.000.000 de inteiros em Cuda. (940MX)

GeForce Tesla V100 (Volta, GDDR5, 16GB, 5120 cuda cores). Debian Strech 9.9.		
Nome	Tempo de execução do *kernel* (ms)	Tempo de execução total (ms)
soma_vet_global	0,756	401,600
soma_vet_managed	126,258	497,800
soma_vet_const	nulo	nulo
soma_vet_shared	0,012	335,400
soma_vet_local	0,703	34,600

Tabela 3: Soma de três vetores de 50.000.000 de inteiros em Cuda. (Tesla V100)

2.7. Sincronização explícita

A sincronização explícita das *threads* em *CUDA* pode ser feita com barreiras, de duas formas principais:
__syncthreads() pode ser executado em uma função de *kernel* para forçar a sincronização de todas as *threads* de um mesmo bloco.
Exemplo: no exemplo anterior da inversão de um vetor de inteiros utilizando memória compartilhada (Seção 2.2) temos o seguinte trecho de código (função de *kernel staticReverse):*

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Neste código, *__syncthreads* é utilizado para garantir que todas as *threads* façam a inversão dos elementos do vetor nos índices *t* e *tr* na ordem correta.

cudaDeviceSynchronize() é utilizado no *host* para forçar a sincronização de todas as *threads* do *device*. O lançamento dos *kernel*s não é síncrono, i.e. a próxima instrução após ele pode ser executada pela CPU antes do *kernel* terminar a sua execução na GPU. Caso a próxima instrução do código do *host* também solicite uma execução na GPU, então, por padrão, essas execuções são serializadas em uma mesma sequência (*stream*).

O código a seguir faz a impressão da mensagem *hello* pelo *device* e *world* pelo *host*. A função *cudaDeviceSynchronize* é utilizada no *host* para garantir a impressão da mensagem na ordem correta.

```
#include<stdio.h>
#include<stdlib.h>
#include<cuda_runtime.h>

__global__ void hello(){
   printf("Hello ");
}

int main(int argc,char **argv){
   hello<<<1,1>>>();
```

```
    cudaDeviceSynchronize();
    printf("World\n");
}
```

3. Um ponto de partida para a solução do desafio

3.1 Implementação sequencial do desafio

```
#include<stdio.h>
#include<stdlib.h>

int main(int argc,char **argv){
    //Declara as matrizes
    double *matrizA,*matrizB,*matrizC;
    //Declara as variáveis de tamanho e índice
    int tam,i,j,k;

    //Lê a dimensão da matriz
    fscanf(stdin,"%d\n",&tam);

    //Aloca as matrizes
    matrizA=(double*)malloc(tam*tam*sizeof(double));
    matrizB=(double*)malloc(tam*tam*sizeof(double));
    matrizC=(double*)malloc(tam*tam*sizeof(double));

    //Lê as matrizes A e B
    for(i=0;i<tam;i++)
        for(j=0;j<tam;j++)
            fscanf(stdin, "%lf ", &matrizA[i * tam + j]);
    for(i=0;i<tam;i++)
        for(j=0;j<tam;j++)
            fscanf(stdin,"%lf ",&matrizB[i*tam+j]);

    //Calcula C=A*B
    for(i=0;i<tam;i++)
        for(j=0;j<tam;j++)
```

```
            for(k=0;k<tam;k++)
                matrizC[i*tam+j]+=matrizA[i*tam+k]*matrizB[k*tam+j];


    //Imprime o resultado
    for(i=0;i<tam;i++){
        for(j=0;j<tam;j++)
            printf("%.1lf ",matrizC[i*tam+j]);
        printf("\n");
    }

    //Desaloca as matrizes
    free(matrizA);
    free(matrizB);
    free(matrizC);

    return 0;
}
```

4. Referências Bibliográficas

Kirk, David B., and W. Hwu Wen-Mei. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, Cap 04, 2016. Third edition. (Livro Texto)

Bibliografia Complementar

Barlas, G. (2014). Multicore and GPU Programming: An integrated approach. Elsevier. Capítulo 6.

Patterson, D. A., & Hennessy, J. L. (2013). Computer Organization and Design MIPS Edition: The Hardware/Software Interface. Newnes. Apêndice C.

Rauber, T., & Rünger, G. (2013). Parallel Programming. Springer. Second edition. Capítulo 7.

Sanders, J., & Kandrot, E. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/ - Guia de programação oficial CUDA.