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 10 - Introdução a CUDA III: Diminuindo o overhead de transferência de memória

por

Guilherme Martins

Paulo Sérgio Lopes de Souza

São Carlos/BR, junho de 2020

1. Desafio 2
1.1. Encontrar o mínimo para duas matrizes. 2
2. O que você precisa saber para resolver o desafio 3
2.1 Operações atômicas 3
2.1.1 Tipos de operações atômicas 3
2.2 Execução assíncrona 8
2.2.1 Memória de host não paginável (*pinned memory)* 9
2.2.2 Memória mapeada (*mapped memory)* 11
2.2.3 Streams 13
2.2.4 Utilizando múltiplas *streams* 15
3. Um ponto de partida para a solução do desafio 21
3.1 Implementação sequencial do desafio 21
4. Referências Bibliográficas 23

1. Desafio

1.1. Encontrar o mínimo para duas matrizes.

Considere as matrizes A[L*C] e B[L*C].

Matriz A [3, 2]

1	2
3	4
5	6

Matriz B [2,3]

9	8	7
6	5	4

O menor elemento da matriz A é 1
O menor elemento da matriz B é 4.

Faça um programa em CUDA C, considerando os conceitos vistos, para calcular o elemento mínimo (menor elemento) para cada uma de duas matrizes não quadradas do tipo ***int***. Utilize múltiplas *streams* e operações atômicas.

Considere como entrada um arquivo de texto contendo, na primeira linha, o número de linhas (L1) e colunas (C1) da matriz A, separadas por um único espaço. Na segunda linha, o número de linhas (L2) e colunas (C2) da matriz B. A partir da terceira linha, estão os elementos da matriz A, do tipo ***int***, onde as linhas são separadas por uma quebra de linha simples e as colunas por um único espaço. Por fim, estão os elementos da matriz B, também do tipo *int* e de dimensão L2*C2.

Conteúdo do arquivo de entrada, por exemplo, ***entrada.txt***:
3 2
2 3
1 2
3 4
5 6
9 8 7
6 5 4

Para executar no ***bash***, por exemplo, utilize este padrão:
.\calc_matrizes <enter>

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

A saída deve ser impressa, utilizando o *output (**stdout**)* padrão, apenas com o menor elemento da matriz *A* e o menor elemento da matriz *B,* separados por uma quebra de linha. Há uma quebra de linha após o menor elemento da matriz B.

1

4

2. O que você precisa saber para resolver o desafio

2.1 Operações atômicas

As operações atômicas em memória são utilizadas para proteger uma região crítica, evitar condições de disputa e garantir a exclusão mútua entre diferentes *threads* ou processos que acessam essa região crítica.

Em *CUDA*, as operações atômicas são utilizadas para garantir a semântica de operações sobre dados compartilhados (armazenados na memória compartilhada ou global) entre múltiplas *threads*. Os dispositivos a partir da capacidade *CUDA* 2.0 (*compute capability)* suportam operações atômicas em *GPU*. Dispositivos com capacidade *CUDA* a partir de 6.0 suportam novos tipos de operações atômicas, que não serão abordadas neste material. Para mais informações sobre as novas implementações, consulte a documentação oficial da *Nvidia*.

2.1.1 Tipos de operações atômicas

Todas as operações atômicas em *CUDA* consideram um ponteiro que corresponde ao endereço de memória da variável compartilhada entre as *threads* e argumentos da operação atômica. As operações atômicas básicas possíveis em *CUDA* são:

Aritméticas

Soma: Faz a soma entre um valor contido em uma variável e outro valor informado como argumento, armazenando o resultado na mesma variável. A sintaxe desta função é:

type atomicAdd(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int, unsigned long long int, float* e *double.* O parâmetro ***address*** é o endereço de memória do destino da soma e ***val*** é o valor a ser somado ao elemento previamente contido no destino da soma. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Subtração: Faz a subtração entre um valor contido em uma variável e outro valor informado como argumento, armazenando o resultado na mesma variável. A sintaxe desta função é:

type atomicSub(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int.* O parâmetro ***address*** é o endereço de memória do destino da subtração e ***val*** é o valor a ser subtraído do elemento previamente contido no destino da subtração. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Substituição (*exchange*): Substitui o valor contido em uma variável por outro valor informado como argumento. A sintaxe desta função é:

type atomicExch(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int, unsigned long long int e float.* O parâmetro ***address*** é o endereço de memória do destino da substituição e ***val*** é o valor que substitui o elemento previamente contido no destino da substituição. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Mínimo: Calcula o mínimo entre o valor contido em uma variável e outro valor informado como argumento, armazenando o resultado na mesma variável.

type atomicMin(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int e unsigned long long int.* O parâmetro ***address*** é o endereço de memória do destino do cálculo do mínimo. A variável ***val*** é o elemento utilizado no cálculo de mínimo.

O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Máximo: Calcula o máximo entre o valor contido em uma variável e outro valor informado como argumento, armazenando o resultado na mesma variável.

type atomicMax(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int e unsigned long long int.* O parâmetro ***address*** é o endereço de memória do destino do cálculo do máximo. A variável ***val*** é o elemento utilizado no cálculo de máximo. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Incremento condicional: Incrementa o valor contido em uma variável, caso o seu conteúdo seja menor do que um valor informado como argumento; caso contrário, a variável recebe 0. Ou seja: ((valorAnterior < novoValor) ? (valorAnterior+1) : 0).

A sintaxe desta função é:

unsigned int atomicInc(unsigned int address, unsigned int val);*

Onde ***address*** é o endereço de memória do destino do incremento. A variável ***val*** é o elemento utilizado para comparação. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Decremento condicional:

Decrementa o valor contido em uma variável, caso o seu conteúdo seja maior que 0 e menor do que um valor informado como argumento; caso contrário, a variável recebe o elemento informado como argumento da função. Ou seja: (((valorAnterior > 0) & (valorAnterior < novoValor)) ? (valorAnterior-1) : novoValor).

A sintaxe desta função é:

unsigned int atomicDec(unsigned int address, unsigned int val);*

Onde ***address*** é o endereço de memória do destino do incremento. A variável ***val*** é o elemento utilizado para comparação. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Substituição condicional (*Compare and swap*): Substitui o conteúdo de uma variável por um novo valor, caso o conteúdo original seja igual a um elemento para comparação. Caso

contrário, mantém o valor original da variável. Ou seja, (valorAnterior == valorDeComparacao ? valorNovo : valorAnterior).

A sintaxe desta função é:

type atomicCAS (type address, type compare, type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int, unsigned long long int e short int.* O parâmetro ***address*** é o endereço de memória do destino da substituição, ***val*** é o valor que substitui o elemento previamente contido no destino da substituição e ***compare*** é o valor para comparação. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

Lógicas binárias (*bitwise)*:

AND: Faz a operação lógica binária de *AND (&,* em linguagem *C)* entre o conteúdo de uma variável e um elemento informado por argumento, armazenando o resultado da operação da mesma variável. A sintaxe desta função é:

type atomicAnd(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int* e *unsigned long long int.* O parâmetro ***address*** é o endereço de memória do destino da operação de *AND*, ***val*** é o outro valor utilizado na operação. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

OR: Faz a operação lógica binária de *OR (|,* em linguagem *C)* entre o conteúdo de uma variável e um elemento informado por argumento, armazenando o resultado da operação da mesma variável. A sintaxe desta função é:

type atomicAnd(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int* e *unsigned long long int.* O parâmetro ***address*** é o endereço de memória do destino da operação de *OR*, ***val*** é o outro valor utilizado na operação. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.

XOR: Faz a operação lógica binária de *XOR (^,* em linguagem *C)* entre o conteúdo de uma variável e um elemento informado por argumento, armazenando o resultado da operação da mesma variável. A sintaxe desta função é:

type atomicAnd(type address,type val);*

Onde ***type*** é o tipo de dados das variáveis utilizadas na operação, que pode ser *int, unsigned int* e *unsigned long long int.* O parâmetro ***address*** é o endereço de memória do destino da operação de *XOR*, ***val*** é o outro valor utilizado na operação. O retorno da função é o valor previamente contido (antes da operação atômica) na variável apontada por *address*.
Exemplo:

Considere a soma de todos os elementos de um vetor de inteiros. Observe o uso da operação *atomicAdd* para fazer o incremento da variável compartilhada *soma*, que está armazenada na memória global do *device.*

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

__global__ void soma_elementos(int *vetorA,int *soma,int tam)
{
   //Calcula o índice global da thread
   int idx = threadIdx.x+blockIdx.x*blockDim.x;
   if (idx < tam)
   {
      //Faz a soma entre elemento do vetor no índice idx e o conteúdo de soma
      atomicAdd(soma,vetorA[idx]);
   }
}

int main(int argc,char **argv)
{

   //Declara as variáveis para uso no host
   int i,*vetorA,threadsPerBlock,blocksPerGrid,soma;

   //Declara os ponteiros para alocação no device
   int *vetorA_d,*soma_d;

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

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

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

   //Aloca o vetore no device
   cudaMalloc((void**)&vetorA_d,tam*(sizeof(int)));
   //Aloca uma variável para armazenar a soma dos elementos do vetor
   cudaMalloc((void**)&soma_d,sizeof(int));
```

```
    //Inicializa o conteúdo da variável no device com 0
    cudaMemset(soma_d,0,sizeof(int));

    //Preenche os vetores no host
    for(i=0;i<tam;i++)
    {
       vetorA[i]=1;
    }

    //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);

    //Invoca o kernel com blocksPerGrid blocos e threadsPerBlock threads
    soma_elementos <<<blocksPerGrid,threadsPerBlock>>> (vetorA_d,soma_d,tam);

    //Copia o resultado da soma de volta para o host
    cudaMemcpy(&soma,soma_d,sizeof(int), cudaMemcpyDeviceToHost);

    //Imprime o resultado no host
    printf("%d\n",soma);

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

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

2.2 Execução assíncrona

A transferência de dados entre a memória do *host* e a memória do *device* pode degradar significativamente o desempenho de uma aplicação *CUDA*. No intuito de diminuir o *overhead* da transferência de memória, os dispositivos *Nvidia* com capacidade de computação (*compute capability)* a partir de 2.0 permitem a execução assíncrona de diretivas CUDA, como lançamentos de *kernel,* cópias e alocações de memória. O objetivo do uso destes recursos é sobrepor transferências de memória com outras computações, mitigando o *overhead* e ocultando a latência de memória.

A execução assíncrona em CUDA ocorre por meio do uso de primitivas denominadas *streams,* que são sequências ordenadas de comandos que são executados sequencialmente. Entretanto, múltiplos *streams* podem executar concorrentemente, maximizando o uso dos recursos computacionais disponíveis.

2.2.1 Memória de host não paginável (*pinned memory)*

O primeiro conceito necessário para utilizar *streams* eficientemente é o de **memória não paginável** (*pinned memory* ou *page-locked host memory*). Este tipo de memória trata-se de um espaço de endereçamento da memória física que não é paginado, ou seja, as páginas da memória física não são substituídas (*swap)* por páginas contidas em memória secundária.

CUDA permite a alocação de um *buffer* de memória não paginável no *host* que permite que o *device* transfira dados entre o mesmo e a *CPU* por *DMA* (*Direct Memory Access - Acesso Direto à Memória*), que não precisa da intervenção da *CPU.* Desta forma, permite-se que a *CPU* não seja bloqueada durante cópias de memória entre *host* e *device*.

É importante notar que o tamanho da alocação não paginável pode, potencialmente, degradar todo sistema e outras aplicações que estão em execução, visto que a memória alocada não pode ser utilizada por outras aplicações o que reduz, efetivamente, a memória física disponível para todo sistema.

Para alocar memória não paginável no *host*, utiliza-se a função *cudaMallocHost,* cujos parâmetros (idênticos à função *cudaMalloc*) são descritos abaixo.

*cudaError_t cudaMallocHost(void **ptr, size_t size)*

Onde *ptr* é o destino da alocação e size é o tamanho, em *bytes*, da alocação.

Para desalocar memória não paginável utiliza-sa a função:

*cudaFreeHost(void *ptr);*

Exemplo:

Considere a soma de dois vetores com memória não paginada. Observe que as únicas alterações são o uso da função *cudaMallocHost* para alocação no *host* e da função *cudaFreeHost* para desalocar a memória não paginada.

```
#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;
    int *vetorA_d,*vetorB_d,*vetorC_d;

    int tam= 5000;
    //Define a quantidade de threads por bloco
    threadsPerBlock = 256;

    //Aloca os vetores no host
    cudaMallocHost((void**)&vetorA,tam*(sizeof(int)));
    cudaMallocHost((void**)&vetorB,tam*(sizeof(int)));
    cudaMallocHost((void**)&vetorC,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

    soma <<<blocksPerGrid,threadsPerBlock>>> (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
    cudaFreeHost(vetorA);
    cudaFreeHost(vetorB);
    cudaFreeHost(vetorC);

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

2.2.2 Memória mapeada (*mapped memory)*

Na terminologia *CUDA,* ***zero-copy memory*** **ou** ***mapped memory*** definem o espaço de endereçamento não paginável do *host* que pode ser mapeado na memória do *device*. Isto significa que um mesmo ponteiro pode ser utilizado para alocar memória tanto no *host* quanto no *device*. A mapeamento da mesma região de memória tanto no *host* quanto no *device* é chamado de **Endereço Virtual Unificado** (*Unified Virtual Address - UVA*).

A **diferença entre memória unificada e memória mapeada** é que na memória unificada a transferência de memória (*cudaMemcpy*) é implícita, enquanto na memória mapeada ela ainda pode ser feita explicitamente. Além disso, na memória unificada as transferências de memória são feitas imediatamente antes do lançamento de um *kernel* e após a finalização do mesmo, enquanto na memória mapeada as transferências são feitas sob demanda (por exemplo, durante a execução de um *kernel)*.

Para alocar memória mapeada utiliza-se a função *cudaHostAlloc*, cujos parâmetros se encontram abaixo:

*cudaError_t cudaHostAlloc(void **ptr, size_t size, unsigned int flag)*

Onde ***ptr*** é o destino da alocação, ***size*** indica o tamanho, em *bytes,* da alocação, *flag* define as opções da alocação, que podem ser:

cudaHostAllocDefault: A função se comporta de maneira idêntica à função *cudaMallocHost*.
cudaHostAllocPortable: A memória alocada será considerada não paginável para todos os contextos *CUDA*, não somente aquele que fez a alocação.
cudaHostAllocMapped: Mapeia a alocação no espaço de endereçamento do *device*. Com esse parâmetro, utiliza-se um único ponteiro para acessar a memória do *host* ou *device*. Pode-se atribuir um ponteiro específico para acessar a memória do *device* por meio da função *cudaHostGetDevicePointer.*
cudaHostAllocWriteCombined: Aloca a memória como escrita combinada (WC - *write combined*). A memória com *WC* pode ser transferida através do barramento mais

rapidamente em algumas configurações de sistema, mas pode não ser lida eficientemente pela maioria das *CPU*s.

Exemplo:

Considere a soma de dois vetores com memória mapeada. Observe que a instrução cudaHostAlloc faz a alocação tanto na memória do *host* quanto no *device*, utilizando o parâmetro *cudaHostAllocMapped.* Observe que, neste caso, as cópias de memória entre o *host* e o *device* são feitas de maneira implícita, de forma idêntica à memória unificada.

```
#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;

   int tam= 5000;
   //Define a quantidade de threads por bloco
   threadsPerBlock = 256;

   //Aloca os vetores no host e no device (memória mapeada em endereço virtual unificado)
   cudaHostAlloc((void**)&vetorA,tam*(sizeof(int)),cudaHostAllocMapped);
   cudaHostAlloc((void**)&vetorB,tam*(sizeof(int)),cudaHostAllocMapped);
   cudaHostAlloc((void**)&vetorC,tam*(sizeof(int)),cudaHostAllocMapped);

   //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);

   //Imprime o resultado no host
```

```
    for(i=0;i<tam;i++)
    {
      printf("%d ",vetorC[i]);
    }

    //Desaloca os vetores no host e no device
    cudaFreeHost(vetorA);
    cudaFreeHost(vetorB);
    cudaFreeHost(vetorC);
}
```

2.2.3 Streams

Streams são sequências ordenadas de comandos CUDA que são executadas sequencialmente. Múltiplas *streams* podem ser executadas concorrentemente, desde que haja recursos computacionais disponíveis. As *streams* são representadas pelo tipo de dados *cudaStream_t.*

Para criar uma *stream* utiliza-se:
*cudaError_t cudaStreamCreate(cudaStream_t *pStream)*
onde *pStream* é um ponteiro do tipo *cudaStream_t* que identifica uma *stream*.

Para destruir uma *stream* utiliza-se:
cudaError_t cudaStreamDestroy(cudaStream_t stream)

Para utilizar *streams*, há dois mecanismos fundamentais:
cudaMemcpyAsync é uma variação do *cudaMemcpy* que permite cópias de memória assíncronas entre *host* e *device*, quando memória não paginável é utilizada.

cudaError_t cudaMemcpyAsync(void *dst, void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)
Todos os parâmetros são idênticos à função *cudaMemcpy,* com exceção do parâmetro **stream** que representa a *stream* atualmente em execução.

O outro mecanismo é a alteração nos parâmetros de lançamento de *kernel*, que recebe um novo elemento, corresponde a *stream* atualmente em execução.
funcaoDeKernel <<<gridDim,blockDim,sharedMemory,**stream**>>> (parâmetros da função).

Para sincronizar o *host* com uma *stream*, isto é, bloquear a execução do *host* até finalizar todos os comandos já lançados em uma *stream* específica, utiliza-se a função *cudaStreamSynchronize*.

cudaError_t *cudaStreamSynchronize(*cudaStream_t stream)

Exemplo:

Considere a soma de dois vetores, onde os dados são transferidos assincronamente, por porções de tamanho *bloco*, com o uso de uma *stream*. Observe a alocação de memória não paginada no *host*, as cópias de memória com *cudaMemcpyAsync*, o lançamento do *kernel* com a *stream* como parâmetro e a sincronização da *stream* com *cudaStreamSynchronize.*

```
#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;
   int *vetorA_d,*vetorB_d,*vetorC_d;

   //Declaração da variável do tipo cudaStream_t
   cudaStream_t stream;

   //Criação da stream
   cudaStreamCreate(&stream);

   //Define o tamanho do vetor, múltiplo de 256
   int tam= 5120;

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

   //Define o tamanho do bloco para divisão dos dados.
   int bloco=tam/threadsPerBlock;

   //Aloca os vetores no host
   cudaMallocHost((void**)&vetorA,tam*(sizeof(int)));
   cudaMallocHost((void**)&vetorB,tam*(sizeof(int)));
   cudaMallocHost((void**)&vetorC,tam*(sizeof(int)));

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

    for(i=0;i<tam;i+=bloco)
    {
        //copia um bloco de tamanho bloco do vetor A do host para o device
        cudaMemcpyAsync(vetorA_d,vetorA+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream);
        //copia um bloco de tamanho bloco do vetor B do host para o device
        cudaMemcpyAsync(vetorB_d,vetorB+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream);
        //Invoca o kernel soma passando a stream como argumento
        soma <<<blocksPerGrid,threadsPerBlock,0,stream>>> (vetorA_d,vetorB_d,vetorC_d,bloco);
        //Copia um bloco de tamanho bloco do resultado de volta para o host
        cudaMemcpyAsync(vetorC+i,vetorC_d,bloco*(sizeof(int)),cudaMemcpyDeviceToHost,stream);
    }
    //Sincroniza a stream
    cudaStreamSynchronize(stream);

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

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

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

    //Destroi a stream
    cudaStreamDestroy(stream);
}
```

2.2.4 Utilizando múltiplas *streams*

Para múltiplas *streams* o processo permanece o mesmo. Elas devem ser declaradas, como variáveis do tipo *cudaStream_t*, inicializadas com a função *cudaStreamCreate*, distribuir o processamento entre elas (cópias assíncronas de memória e lançamentos de *kernel),* sincronizadas com *cudaStreamSynchronize* e destruídas, com *cudaStreamDestroy*.

A principal razão para utilização de múltiplas *streams* é implementar paralelismo de tarefas ou de funcionalidade, onde múltiplos *kernels* podem operar sobre o mesmo conjunto de dados ou sobre dados distintos.

Exemplo:

Considere o exemplo da soma de vetores. Considere agora distribuição do conjunto de dados em blocos com a metade daquele utilizado no exemplo anterior. São alocados 6 vetores (A, B e C) no *device*: 3 para cada *stream.* Cada uma das *streams* encapsula a cópia de memória entre blocos do *host* e seus 3 vetores e o lançamento do *kernel* soma, com conjuntos de dados diferentes.

```
#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;

   //Declara os vetores para uso na primeira stream
   int *vetorA_d1,*vetorB_d1,*vetorC_d1;

   //Declara os vetores para uso na segundo stream
   int *vetorA_d2,*vetorB_d2,*vetorC_d2;

   //Declaração da variável do tipo cudaStream_t
   cudaStream_t stream1,stream2;

   //Criação das streams
   cudaStreamCreate(&stream1);
   cudaStreamCreate(&stream2);

   //Define o tamanho do vetor, múltiplo de 256
   int tam= 5120;

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

   //Define o tamanho do bloco para divisão dos dados.
   //Divide o conjunto para computação dos dados em duas streams
```

```
int bloco=(tam/threadsPerBlock)/2;    // 5120 /256 = 20 blocos / 2 = 10 blocos

//Aloca os vetores no host
cudaMallocHost((void**)&vetorA,tam*(sizeof(int)));
cudaMallocHost((void**)&vetorB,tam*(sizeof(int)));
cudaMallocHost((void**)&vetorC,tam*(sizeof(int)));

//Aloca os vetores no device para a stream 1
cudaMalloc((void**)&vetorA_d1,bloco*(sizeof(int)));
cudaMalloc((void**)&vetorB_d1,bloco*(sizeof(int)));
cudaMalloc((void**)&vetorC_d1,bloco*(sizeof(int)));

//Aloca os vetores no device para a stream 2
cudaMalloc((void**)&vetorA_d2,bloco*(sizeof(int)));
cudaMalloc((void**)&vetorB_d2,bloco*(sizeof(int)));
cudaMalloc((void**)&vetorC_d2,bloco*(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;

for(i=0;i<tam;i+=bloco*2)
{
   //copia um bloco de tamanho bloco do vetor A do host para o device (stream1)
   cudaMemcpyAsync(vetorA_d1,vetorA+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream1);

   //copia um bloco de tamanho bloco do vetor B do host para o device (stream1)
   cudaMemcpyAsync(vetorB_d1,vetorB+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream1);

   //Invoca o kernel soma passando a stream 1 como argumento
   soma <<<blocksPerGrid,threadsPerBlock,0,stream1>>> (vetorA_d1,vetorB_d1,vetorC_d1,bloco);

   //Copia um bloco de tamanho bloco do resultado da stream 1 de volta para o host
   cudaMemcpyAsync(vetorC+i,vetorC_d1,bloco*(sizeof(int)),cudaMemcpyDeviceToHost,stream1);

   //copia um bloco de tamanho bloco do vetor A do host para o device (stream2)
   cudaMemcpyAsync(vetorA_d2,vetorA+i+bloco,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream2);

   //copia um bloco de tamanho bloco do vetor B do host para o device (stream2)
   cudaMemcpyAsync(vetorB_d2,vetorB+i+bloco,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream2);

   //Invoca o kernel soma passando a stream 2 como argumento
   soma <<<blocksPerGrid,threadsPerBlock,0,stream2>>> (vetorA_d2,vetorB_d2,vetorC_d2,bloco);

   //Copia um bloco de tamanho bloco do resultado da stream 2 de volta para o host
   cudaMemcpyAsync(vetorC+i+bloco,vetorC_d2,bloco*(sizeof(int)),cudaMemcpyDeviceToHost,stream2);
}
```

```
    //Sincroniza as streams
    cudaStreamSynchronize(stream1);
    cudaStreamSynchronize(stream2);

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

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

    //Desaloca os vetores da stream 1
    cudaFree(vetorA_d1);
    cudaFree(vetorB_d1);
    cudaFree(vetorC_d1);

    //Desaloca os vetores da stream 2
    cudaFree(vetorA_d2);
    cudaFree(vetorB_d2);
    cudaFree(vetorC_d2);

    //Destroi as streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);
}
```

É possível também utilizar *streams* para fazer o lançamento de diferentes *kernels* e fazer operações sobre diferentes conjuntos de dados, emulando o modelo *MIMD.*

Exemplo:

Considere o código em *CUDA* a seguir que faz a soma de dois vetores em CUDA (vetores *A* e *B*) e faz o cálculo do produto de um vetor (vetor *D*) por um escalar (um valor constante gerado aleatoriamente). Neste caso, são utilizadas duas *streams*: a primeira faz o cálculo da soma dos vetores e a segunda faz o cálculo do produto do escalar pelo vetor.

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

//Kernel que faz a soma de vetores
__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];
    }
}

//Kernel que faz a multiplicação de um escalar por um vetor
__global__ void mult_escalar(int *vetorA, int escalar,int tam)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < tam)
    {
        vetorA[idx]=escalar*vetorA[idx];
    }
}

int main(int argc,char **argv)
{
    //Declara as variáveis de índice
    int i,threadsPerBlock,blocksPerGrid;

    //Inicializa a seed para geração de números pseudo aleatórios
    srand(time(NULL));

    //Declara os vetores no host
    int *vetorA,*vetorB,*vetorC,*vetorD;

    int escalar=rand()%10+1;

    //Declara os vetores para uso na primeira stream
    int *vetorA_d1,*vetorB_d1,*vetorC_d1;

    //Declara o vetor para uso na segunda stream
    int *vetorD_d2;

    //Declaração das variáveis do tipo cudaStream_t
    cudaStream_t stream1,stream2;

    //Criação das streams
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);

    //Define o tamanho do vetor, múltiplo de 256
    int tam= 5120;

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

    //Define o tamanho do bloco para divisão dos dados.
    //Divide o conjunto para computação dos dados em duas streams
    int bloco=tam/threadsPerBlock;

    //Aloca os vetores no host
    cudaMallocHost((void**)&vetorA,tam*(sizeof(int)));
```

```
cudaMallocHost((void**)&vetorB,tam*(sizeof(int)));
cudaMallocHost((void**)&vetorC,tam*(sizeof(int)));
cudaMallocHost((void**)&vetorD,tam*(sizeof(int)));

//Aloca os vetores no device para a stream 1
cudaMalloc((void**)&vetorA_d1,bloco*(sizeof(int)));
cudaMalloc((void**)&vetorB_d1,bloco*(sizeof(int)));
cudaMalloc((void**)&vetorC_d1,bloco*(sizeof(int)));

//Aloca os vetores no device para a stream 2
cudaMalloc((void**)&vetorD_d2,bloco*(sizeof(int)));

//Preenche os vetores no host
for(i=0;i<tam;i++)
{
   vetorA[i]=i;
   vetorB[i]=i;
   vetorD[i]=10;
}

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

for(i=0;i<tam;i+=bloco)
{
   //copia um bloco de tamanho bloco do vetor A do host para o device (stream1)
   cudaMemcpyAsync(vetorA_d1,vetorA+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream1);
   //copia um bloco de tamanho bloco do vetor B do host para o device (stream1)
   cudaMemcpyAsync(vetorB_d1,vetorB+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream1);

   //Invoca o kernel soma passando a stream 1 como argumento
   soma <<<blocksPerGrid,threadsPerBlock,0,stream1>>> (vetorA_d1,vetorB_d1,vetorC_d1,bloco);


   //Copia um bloco de tamanho bloco do resultado da stream 1 de volta para o host
   cudaMemcpyAsync(vetorC+i,vetorC_d1,bloco*(sizeof(int)),cudaMemcpyDeviceToHost,stream1);

   //copia um bloco de tamanho bloco do vetor D do host para o device (stream2)
   cudaMemcpyAsync(vetorD_d2,vetorD+i,bloco*(sizeof(int)),cudaMemcpyHostToDevice,stream2);

   //Invoca o kernel mult_escalar passando a stream 2 como argumento
   mult_escalar <<<blocksPerGrid,threadsPerBlock,0,stream2>>> (vetorD_d2,escalar,bloco);

   //Copia um bloco de tamanho bloco do resultado da stream 2 de volta para o host
  cudaMemcpyAsync(vetorD+i,vetorD_d2,bloco*(sizeof(int)),cudaMemcpyDeviceToHost,stream2);
}


//Sincroniza as streams
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
```

```
    //Imprime o resultado da soma de vetores no host
    for(i=0;i<tam;i++)
    {
       printf("%d ",vetorC[i]);
    }
    printf("\n");

    //Imprie o resultado da multiplicação pelo escalar no host
    for(i=0;i<tam;i++)
    {
        printf("%d ",vetorD[i]);
     }

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

    //Desaloca os vetores da stream 1
    cudaFree(vetorA_d1);
    cudaFree(vetorB_d1);
    cudaFree(vetorC_d1);

    //Desaloca o vetor da stream 2
    cudaFree(vetorD_d2);

    //Destroi as streams
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);
}
```

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
    int *matrizA,*matrizB;
    //Declara as variáveis de tamanho e índice
    int i,j,k;
    //Declara as linhas e colunas para as matrizes A e B
```

```
int lin_A,col_A;
int lin_B,col_B;

int minA,minB;

//Lê as dimensões da matriz A
fscanf(stdin,"%d ",&lin_A);
fscanf(stdin,"%d\n",&col_A);

//Lê as dimensões da matriz B
fscanf(stdin,"%d ",&lin_B);
fscanf(stdin,"%d\n",&col_B);

//Aloca as matrizes
matrizA=(int*)malloc(lin_A*col_A*sizeof(int));
matrizB=(int*)malloc(lin_B*col_B*sizeof(int));

//Lê a matriz A
for(i=0;i<lin_A;i++)
   for(j=0;j<col_A;j++)
      fscanf(stdin,"%d ", &matrizA[i*col_A+j]);

//Lê a matriz B
for(i=0;i<lin_B;i++)
   for(j=0;j<col_B;j++)
      fscanf(stdin,"%d ",&matrizB[i*col_B+j]);

//Inicializa o contador de A
minA=matrizA[0];

//Inicializa o contador de B
minB=matrizB[0];

 //Calcula os elementos de A
 for(i=0;i<lin_A;i++)
 {
    for(j=0;j<col_A;j++)
    {
       if(matrizA[i*col_A+j]<minA)
          minA=matrizA[i*col_A+j];
   }
 }
  //Calcula os elementos de B
 for(i=0;i<lin_B;i++)
 {
```

```
        for(j=0;j<col_B;j++)
        {
            if(matrizB[i*col_B+j]<minB)
                minB=matrizB[i*col_B+j];
        }
    }
    printf("%d\n",minA);
    printf("%d",minB);

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

    return 0;
}
```

4. Referências Bibliográficas

Livro texto

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

Bibliografia Complementar

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

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.

Referências Eletrônicas

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