

MC3-III

Programação Avançada com CUDA

Roberto P. Souto (LNCC)
rpsouto@lncc.br

Roteiro

- 1 Módulo 1: uso eficiente de memória
- 2 Módulo 2: transferência otimizada de dados
- 3 Módulo 3: transferência assíncrona de dados
- 4 Módulo 4: processamento em múltiplas GPUs

Roteiro

- 1 Módulo 1: uso eficiente de memória
- 2 Módulo 2: transferência otimizada de dados
- 3 Módulo 3: transferência assíncrona de dados
- 4 Módulo 4: processamento em múltiplas GPUs

Códigos

```
$ cd $SCRATCH
$ mkdir MC3-III
$ cd MC3-III
$ cp /prj/treinamento/professor/modulo3/MC3-III/codigos_MC3-III.tar .
$ tar xvf codigos_MC3-III.tar
$ ls codigos/ -A1
blogforall
multigpu
```

Uso eficiente de memória: Matriz Transposta

- Exemplo retirado do *blog* Parallel for All:

<https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/>

```
$ cd codigos/blogforall/cuda-cpp/transpose
$ make
$ srun -p treinamento_gpu ./transpose
Device : Tesla K40t
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
```

Routine	Bandwidth (GB/s)
copy	168.97
shared memory copy	192.26
naive transpose	68.00
coalesced transpose	119.65
conflict-free transpose	187.66

Uso eficiente de memória: Matriz Transposta

- Exemplo retirado do *blog* Parallel for All:

<https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/>

- Neste exemplo de transposição de matrizes, veremos o impacto que um ineficiente padrão de acesso à memória global tem no desempenho;

```
$ cd codigos/blogforall/cuda-cpp/transpose
$ make
$ srun -p treinamento_gpu ./transpose
Device : Tesla K40t
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
```

Routine	Bandwidth (GB/s)
copy	168.97
shared memory copy	192.26
naive transpose	68.00
coalesced transpose	119.65
conflict-free transpose	187.66

Uso eficiente de memória: Matriz Transposta

- Exemplo retirado do *blog* Parallel for All:

`https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`

- Neste exemplo de transposição de matrizes, veremos o impacto que um ineficiente padrão de acesso à memória global tem no desempenho;
- E como o uso de memória compartilhada pode superar este problema.

```
$ cd codigos/blogforall/cuda-cpp/transpose
$ make
$ srun -p treinamento_gpu ./transpose
Device : Tesla K40t
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
```

Routine	Bandwidth (GB/s)
copy	168.97
shared memory copy	192.26
naive transpose	68.00
coalesced transpose	119.65
conflict-free transpose	187.66

Uso eficiente de memória: Matriz Transposta

- Exemplo retirado do *blog* Parallel for All:

`https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`

- Neste exemplo de transposição de matrizes, veremos o impacto que um ineficiente padrão de acesso à memória global tem no desempenho;
- E como o uso de memória compartilhada pode superar este problema.

```
$ cd codigos/blogforall/cuda-cpp/transpose
$ make
$ ./transpose
Device : Tesla C2050
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
```

Routine	Bandwidth (GB/s)
copy	102.57
shared memory copy	101.33
naive transpose	18.50
coalesced transpose	52.64
conflict-free transpose	96.99

Uso eficiente de memória: Matriz Transposta

Cópia simples (e aglutinada) de `idata` em `odata`

`transpose.cu`

```
// simple copy kernel
// Used as reference case representing best effective bandwidth.
__global__ void copy(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
        odata[(y+j) * width + x] = idata[(y+j) * width + x];
}
```

\$./transpose

Routine
copy

Bandwidth (GB/s)
102.57

Uso eficiente de memória: Matriz Transposta

Transposição ineficiente(*naive*) de *idata* em *odata*

transposeNaive

```
// naive transpose: simplest transpose;
// Global memory reads are coalesced but writes are not.
__global__ void transposeNaive(float *odata, const float *idata)
{
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j+= BLOCK_ROWS)
        odata[x * width + (y+j)] = idata[(y+j) * width + x];
}
```

\$./transpose

Routine	Bandwidth (GB/s)
copy	102.57
naive transpose	18.50

Uso eficiente de memória: Matriz Transposta

Transposição ineficiente(*naive*) de `idata` em `odata`

- Escrita com acesso aglutinado aos endereços de `odata`:

```
odata[(y+j) * width + x] = idata[(y+j) * width + x];
```

Uso eficiente de memória: Matriz Transposta

Transposição ineficiente(*naive*) de `idata` em `odata`

- Escrita com acesso aglutinado aos endereços de `odata`:

```
odata[(y+j) * width + x] = idata[(y+j) * width + x];
```

- Escrita com acesso não aglutinado aos endereços de `odata`:

```
odata[x * width + (y+j)] = idata[(y+j) * width + x]
```

Espaço (*stride*) entre endereços na proporção do tamanho da matriz (variável *width*).

Desempenho Computacional

Padrão de acesso à memória global

- Em linguagem C, os índices de matrizes são orientados por linha (*row-major*);
- Ou seja, os dados são acessados em endereços contíguos de memória nas linhas da matriz;
- O acesso aos dados que segue este padrão, denomina-se acesso agrupado (*coalesced access pattern*);
- Qualquer outro tipo de acesso, que não seja em endereços contíguos de memória nas linhas da matriz, denomina-se acesso não agrupado (*uncoalesced access pattern*);
- O acesso não agrupado resulta em uma menor largura de banda da aplicação;

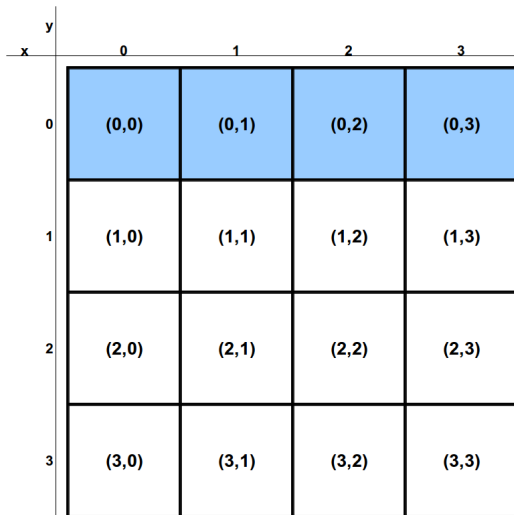
Desempenho Computacional

Padrão de acesso

		0	1	2	3
y					
x	0	(0,0)	(0,1)	(0,2)	(0,3)
	1	(1,0)	(1,1)	(1,2)	(1,3)
	2	(2,0)	(2,1)	(2,2)	(2,3)
	3	(3,0)	(3,1)	(3,2)	(3,3)

Desempenho Computacional

Acesso aglutinado



y \ x	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Desempenho Computacional

Acesso desaglutinado

y \ x	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Uso eficiente de memória: Matriz Transposta

Transposição ineficiente(*naive*) de `idata` em `odata`

- Escrita com acesso aglutinado aos endereços de `odata`:

```
odata[(y+j) * width + x] = idata[(y+j) * width + x];
```

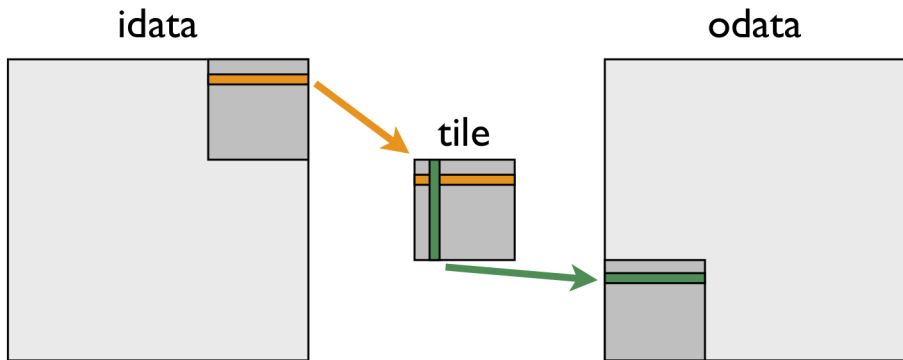
- Escrita com acesso não aglutinado aos endereços de `odata`:

```
odata[x * width + (y+j)] = idata[(y+j) * width + x]
```

Espaço (*stride*) entre endereços na proporção do tamanho da matriz (variável *width*).

Uso eficiente de memória: Matriz Transposta

Versão mais eficiente



- A transposição é feita dentro do bloco (*tile*) usando memória compartilhada, cujo acesso é muito mais rápido;

Uso eficiente de memória: Matriz Transposta

Versão mais eficiente

transposeCoalesced

```
// coalesced transpose
// Uses shared memory to achieve coalescing in both reads and writes
__global__ void transposeCoalesced(float *odata, const float *idata)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

    __syncthreads();

    x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Uso eficiente de memória: Matriz Transposta

Versão mais eficiente

```
$ ./transpose
```

Routine	Bandwidth (GB/s)
copy	102.57
naive transpose	18.50
coalesced transpose	52.64

- Houve significativa melhora na largura de banda alcançada, com relação a versão ineficiente;

Uso eficiente de memória: Matriz Transposta

Versão mais eficiente

```
$ ./transpose
```

Routine	Bandwidth (GB/s)
copy	102.57
naive transpose	18.50
coalesced transpose	52.64

- Houve significativa melhora na largura de banda alcançada, com relação a versão ineficiente;
- Mas ainda é bastante inferior ao desejável;

Uso eficiente de memória: Matriz Transposta

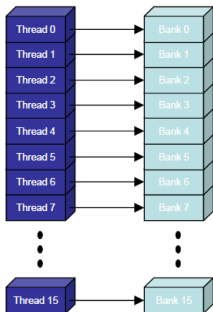
Versão mais eficiente

```
$ ./transpose
```

Routine	Bandwidth (GB/s)
copy	102.57
naive transpose	18.50
coalesced transpose	52.64

- Houve significativa melhora na largura de banda alcançada, com relação a versão ineficiente;
- Mas ainda é bastante inferior ao desejável;
- A causa disso é o **conflito de banco de memória compartilhada** (*bank conflict*).

Uso de memória compartilhada: bancos de memória



- Para alcançar mais alta largura de banda, a memória compartilhada é dividida em módulos de memória de igual tamanho (bancos) que podem ser acessados simultaneamente por diferentes *threads*;

FONTE:

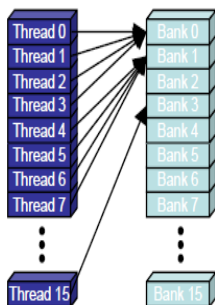
[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/blog-post.html)

[html](http://cuda-programming.blogspot.com.br/2013/02/blog-post.html)

Uso de memória compartilhada: bancos de memória



- Para alcançar mais alta largura de banda, a memória compartilhada é dividida em módulos de memória de igual tamanho (bancos) que podem ser acessados simultaneamente por diferentes *threads*;
- Quando duas ou mais *threads* requisitam endereços em um mesmo banco, ocorre um conflito.

FONTE:

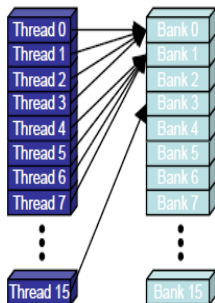
[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/bank-conflicts-in-shared-memory-in-cuda.html)

[html](http://cuda-programming.blogspot.com.br/2013/02/bank-conflicts-in-shared-memory-in-cuda.html)

Uso de memória compartilhada: bancos de memória



- Para alcançar mais alta largura de banda, a memória compartilhada é dividida em módulos de memória de igual tamanho (bancos) que podem ser acessados simultaneamente por diferentes *threads*;
- Quando duas ou mais *threads* requisitam endereços em um mesmo banco, ocorre um conflito.
- O acesso a este banco pelas *threads* é então serializado. As *threads* acessam sequencialmente o banco.

FONTE:

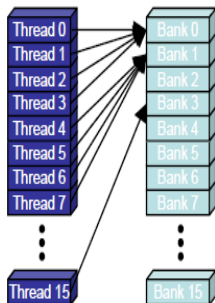
[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/bank-conflicts-in-shared-memory-in-cuda.html)

html

Uso de memória compartilhada: bancos de memória



FONTE:

[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

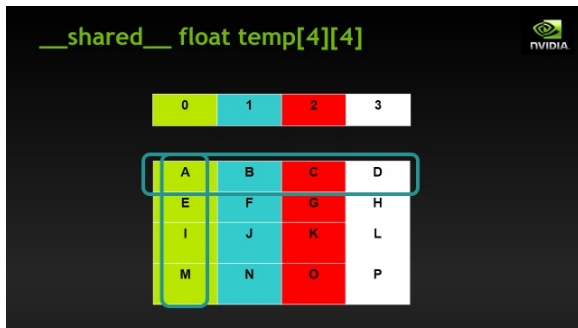
[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/)

html

- Para alcançar mais alta largura de banda, a memória compartilhada é dividida em módulos de memória de igual tamanho (bancos) que podem ser acessados simultaneamente por diferentes *threads*;
- Quando duas ou mais *threads* requisitam endereços em um mesmo banco, ocorre um conflito.
- O acesso a este banco pelas *threads* é então serializado. As *threads* acessam sequencialmente o banco.
- A figura ilustra um exemplo de **4-way bank conflict**.

Uso de memória compartilhada

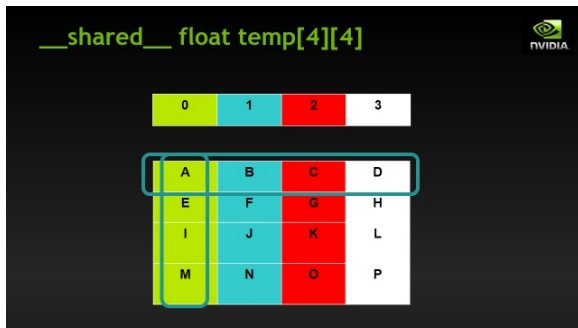
conflito de banco de memória



- bancos (portas) 0, 1, 2, 3
- verde, azul, vermelho e branco
- A, B, C e D são armazenados, respectivamente, nos bancos 0, 1, 2, e 3

Uso de memória compartilhada

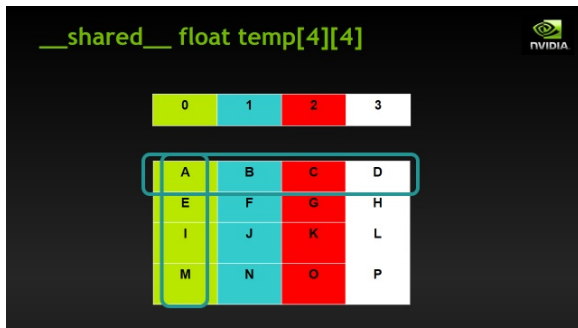
conflito de banco de memória



- Na transposição de matriz, os dados das colunas devem ser acessados
- Os dados de cada coluna estão armazenados no mesmo banco de memória

Uso de memória compartilhada

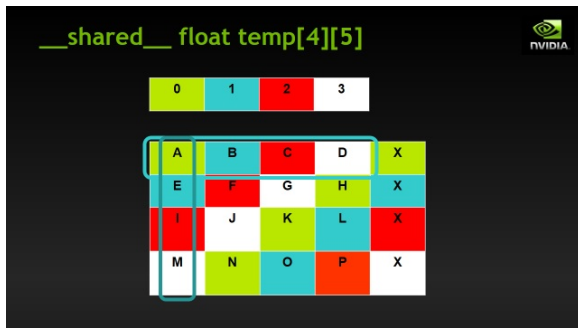
conflito de banco de memória



- Por exemplo, na primeira coluna, os dados A, E, I e M, estão no banco 0 (verde).
- Configura-se neste caso, um *4-way bank conflict*

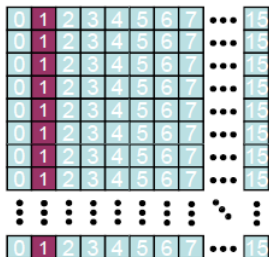
Uso de memória compartilhada

conflito de banco de memória



- Solução: adicionar uma coluna extra

Uso de memória compartilhada: bancos de memória



- Seja um bloco de 2 dimensões de tamanho 16×16 , por exemplo;
- Todos os elementos de uma coluna são mapeados para o mesmo banco de memória;
- Este é o pior caso, que resulta em um *16-way bank conflict*;

FONTE:

[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/bank-conflicts-in-shared-memory-in-cuda.html)

html

Uso de memória compartilhada: bancos de memória

- A solução consiste em acrescentar uma coluna ao bloco:

```
__shared__ float tile[TILE_DIM][TILE_DIM];
```


Uso de memória compartilhada: bancos de memória

- A solução consiste em acrescentar uma coluna ao bloco:

```
__shared__ float tile[TILE_DIM][TILE_DIM];
```

```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Uso de memória compartilhada: bancos de memória

0	1	2	3	4	5	6	7	...	15	0
1	2	3	4	5	6	7	8	...	0	1
2	3	4	5	6	7	8	9	...	1	2
3	4	5	6	7	8	9	10	...	2	3
4	5	6	7	8	9	10	11	...	3	4
5	6	7	8	9	10	11	12	...	4	5
6	7	8	9	10	11	12	13	...	5	6
7	8	9	10	11	12	13	14	...	6	7
:	:	:	:	:	:	:	:	↘	:	:
15	0	1	2	3	4	5	6	...	14	15

- A solução consiste em acrescentar uma coluna ao bloco:

```
__shared__ float tile[TILE_DIM][TILE_DIM];
```

```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

FONTE:

[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/bank-conflicts-in-shared-memory-in-cuda.html)

html

Uso de memória compartilhada: bancos de memória

0	1	2	3	4	5	6	7	...	15	0
1	2	3	4	5	6	7	8	...	0	1
2	3	4	5	6	7	8	9	...	1	2
3	4	5	6	7	8	9	10	...	2	3
4	5	6	7	8	9	10	11	...	3	4
5	6	7	8	9	10	11	12	...	4	5
6	7	8	9	10	11	12	13	...	5	6
7	8	9	10	11	12	13	14	...	6	7
:	:	:	:	:	:	:	:	...	:	:
15	0	1	2	3	4	5	6	...	14	15

- A solução consiste em acrescentar uma coluna ao bloco:

```
__shared__ float tile[TILE_DIM][TILE_DIM];
```

```
__shared__ float tile[TILE_DIM][TILE_DIM + 1];
```

Agora os endereços das colunas são mapeados em diferentes bancos. Não há mais conflito de banco de memória.

FONTE:

[http://cuda-programming.](http://cuda-programming.blogspot.com.br/2013/02/)

[blogspot.com.br/2013/02/](http://cuda-programming.blogspot.com.br/2013/02/)

[bank-conflicts-in-shared-memory-in-cuda.](http://cuda-programming.blogspot.com.br/2013/02/)

html

Uso eficiente de memória: Matriz Transposta

Versão COM conflito de banco de memória

transposeCoalesced

```
// With bank-conflict transpose
__global__ void transposeCoalesced(float *odata, const float *idata)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

    __syncthreads();

    x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Uso eficiente de memória: Matriz Transposta

Versão SEM conflito de banco de memória

transposeNoBankConflicts

```
// With no bank-conflict transpose
__global__ void transposeNoBankConflicts(float *odata, const float *idata)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];

    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;
    int width = gridDim.x * TILE_DIM;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = idata[(y+j)*width + x];

    __syncthreads();

    x = blockIdx.y * TILE_DIM + threadIdx.x; // transpose block offset
    y = blockIdx.x * TILE_DIM + threadIdx.y;

    for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
        odata[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

Uso eficiente de memória: Matriz Transposta

Versão SEM conflito de banco de memória

```
$ ./transpose
Device : Tesla C2050
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
```

Routine	Bandwidth (GB/s)
copy	102.57
shared memory copy	101.33
naive transpose	18.50
coalesced transpose	52.64
conflict-free transpose	96.99

Uso eficiente de memória: Matriz Transposta

Versão SEM conflito de banco de memória

```
$ srun -p treinamento_gpu ./transpose
Device : Tesla K40t
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
dimGrid: 32 32 1. dimBlock: 32 8 1
```

Routine	Bandwidth (GB/s)
copy	168.97
shared memory copy	192.26
naive transpose	68.00
coalesced transpose	119.65
conflict-free transpose	187.66

Roteiro

- 1 Módulo 1: uso eficiente de memória
- 2 Módulo 2: transferência otimizada de dados**
- 3 Módulo 3: transferência assíncrona de dados
- 4 Módulo 4: processamento em múltiplas GPUs

Transferência otimizada de dados

- Exemplo retirado do *blog* Parallel for All:

`https://devblogs.nvidia.com/parallelforall/
how-optimize-data-transfers-cuda-cc/`

Transferência otimizada de dados

- Exemplo retirado do *blog* Parallel for All:

`https://devblogs.nvidia.com/parallelforall/
how-optimize-data-transfers-cuda-cc/`

- A transferência de dados entre o host e o dispositivo, é um dos maiores gargalos de desempenho em programação GPGPU.

Transferência otimizada de dados

- Exemplo retirado do *blog* Parallel for All:

`https://devblogs.nvidia.com/parallelforall/
how-optimize-data-transfers-cuda-cc/`

- A transferência de dados entre o host e o dispositivo, é um dos maiores gargalos de desempenho em programação GPGPU.
- Uma boa prática de programação, consiste em minimizar o número de vezes que este tipo de transferência é realizada no código.

Transferência otimizada de dados

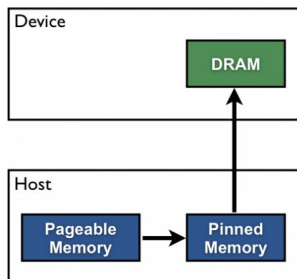
- Exemplo retirado do *blog* Parallel for All:

`https://devblogs.nvidia.com/parallelforall/
how-optimize-data-transfers-cuda-cc/`

- A transferência de dados entre o host e o dispositivo, é um dos maiores gargalos de desempenho em programação GPGPU.
- Uma boa prática de programação, consiste em minimizar o número de vezes que este tipo de transferência é realizada no código.
- Quando for feita transferência, buscar fazê-la de modo mais eficiente possível.

Transferência otimizada de dados

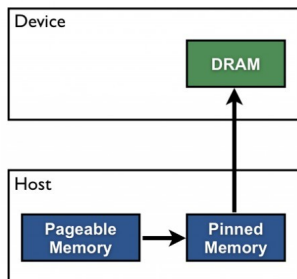
Pageable Data Transfer



- Alocação de dados no host (CPU) são, por padrão, pagináveis

Transferência otimizada de dados

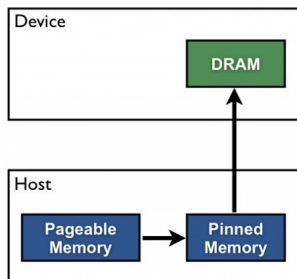
Pageable Data Transfer



- Alocação de dados no host (CPU) são, por padrão, pagináveis
- O dispositivo (GPU) não é capaz de acessar dados diretamente da memória paginável do host

Transferência otimizada de dados

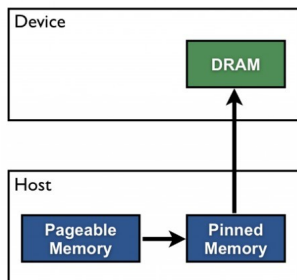
Pageable Data Transfer



- Quando uma transferência de dado da memória paginável é solicitada, o driver CUDA deve primeiro alocar um page-locked array do host

Transferência otimizada de dados

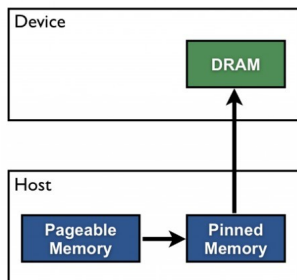
Pageable Data Transfer



- É feita então uma cópia dos dados para o pinned array, e então é feita a transferência dos dados do pinned array para a memória do dispositivo

Transferência otimizada de dados

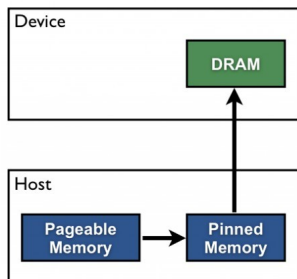
Pageable Data Transfer



- É feita então uma cópia dos dados para o pinned array, e então é feita a transferência dos dados do pinned array para a memória do dispositivo
- A memória pinned é usada como uma etapa intermediária para transferir dados entre o dispositivo e o host

Transferência otimizada de dados

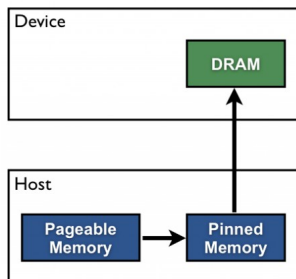
Pageable Data Transfer



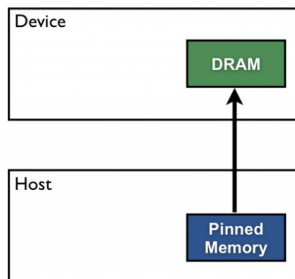
- Este custo de transferência de dados entre a memória paginável e pinned, pode ser evitado ao se alocar espaço diretamente na memória pinned

Transferência otimizada de dados

Pageable Data Transfer



Pinned Data Transfer



- Este custo de transferência de dados entre a memória paginável e pinned, pode ser evitado ao se alocar espaço diretamente na memória pinned

Transferência otimizada de dados

- A alocação de memória pinned em CUDA é feita usando-se `cudaMallocHost()`

`bandwidthtest.cu`

```
int main()
{
    unsigned int nElements = 4*1024*1024;
    const unsigned int bytes = nElements * sizeof(float);

    // host arrays
    float *h_aPageable, *h_bPageable;
    float *h_aPinned, *h_bPinned;

    // device array
    float *d_a;

    // allocate and initialize
    h_aPageable = (float*)malloc(bytes); // host pageable
    h_bPageable = (float*)malloc(bytes); // host pageable
    cudaMallocHost((void**)&h_aPinned, bytes); // host pinned
    cudaMallocHost((void**)&h_bPinned, bytes); // host pinned
    cudaMalloc((void**)&d_a, bytes); // device

    ...
}
```

Transferência otimizada de dados

- Transferência de dados da memória pinned, empregam a mesma função `cudaMemcpy()`, utilizada para as transferências da memória paginável

bandwidthtest.cu

```
void profileCopies(float      *h_a,
                  float      *h_b,
                  float      *d,
                  unsigned int n,
                  char        *desc)
{
    printf("\n%s transfers\n", desc);

    unsigned int bytes = n * sizeof(float);

    cudaMemcpy(d, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(h_b, d, bytes, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; ++i) {
        if (h_a[i] != h_b[i]) {
            printf("*** %s transfers failed ***", desc);
            break;
        }
    }
}
```

Transferência otimizada de dados

```
$ cd codigos/blogforall/cuda-cpp/optimize-data-transfers
$ nvcc -o bandwidthtest bandwidthtest.cu
$ srun -p treinamento_gpu ./bandwidthtest
```

Device: Tesla K40t

Transfer size (MB): 16

Pageable transfers

Host to Device bandwidth (GB/s): 2.418403

Device to Host bandwidth (GB/s): 2.565725

Pinned transfers

Host to Device bandwidth (GB/s): 10.402334

Device to Host bandwidth (GB/s): 10.408116

Transferência otimizada de dados

- É provável haver falha de alocação da memória pinned

Deve-se SEMPRE verificar se há memória pinned suficiente

```
cudaError_t status = cudaMallocHost((void**)&h_aPinned, bytes);  
if (status != cudaSuccess)  
    printf("Error allocating pinned host memoryn");
```

Roteiro

- 1 Módulo 1: uso eficiente de memória
- 2 Módulo 2: transferência otimizada de dados
- 3 Módulo 3: transferência assíncrona de dados**
- 4 Módulo 4: processamento em múltiplas GPUs

Transferência assíncrona de dados

CUDA Stream

- Exemplo retirado do *blog* Parallel for All:

```
https://devblogs.nvidia.com/parallelforall/  
how-overlap-data-transfers-cuda-cc/
```

Transferência assíncrona de dados

CUDA Stream

- Exemplo retirado do *blog* Parallel for All:

```
https://devblogs.nvidia.com/parallelforall/  
how-overlap-data-transfers-cuda-cc/
```

- Uma stream CUDA trata-se de uma sequência (ou fluxo) de operações que são executadas no dispositivo, na ordem em que foram lançadas pelo host

Transferência assíncrona de dados

CUDA Stream

- Exemplo retirado do *blog* Parallel for All:

```
https://devblogs.nvidia.com/parallelforall/  
how-overlap-data-transfers-cuda-cc/
```

- Uma stream CUDA trata-se de uma sequência (ou fluxo) de operações que são executadas no dispositivo, na ordem em que foram lançadas pelo host
- Operações em diferentes streams podem ser intercaladas e, quando possível, podem até mesmo ser executadas simultaneamente

Transferência assíncrona de dados

Stream padrão (*Default Stream*)

- Todas as operações no dispositivo (kernels e transferência de dados), rodam em uma stream

Transferência assíncrona de dados

Stream padrão (*Default Stream*)

- Todas as operações no dispositivo (kernels e transferência de dados), rodam em uma stream
- Quando nenhuma stream é especificada, o stream padrão (ou “null stream”) é utilizado.

Transferência assíncrona de dados

Stream padrão (*Default Stream*)

- Todas as operações no dispositivo (kernels e transferência de dados), rodam em uma stream
- Quando nenhuma stream é especificada, o stream padrão (ou “null stream”) é utilizado.
- O stream padrão é diferente dos demais streams, pois ele sincroniza as operações executadas no dispositivo:

Transferência assíncrona de dados

Stream padrão (*Default Stream*)

- Todas as operações no dispositivo (kernels e transferência de dados), rodam em uma stream
- Quando nenhuma stream é especificada, o stream padrão (ou “null stream”) é utilizado.
- O stream padrão é diferente dos demais streams, pois ele sincroniza as operações executadas no dispositivo:
 - ▶ Nenhuma operação no stream padrão irá começar até que todas as operações anteriormente enviadas, para qualquer stream, tenham sido completadas

Transferência assíncrona de dados

Stream padrão (*Default Stream*)

- Todas as operações no dispositivo (kernels e transferência de dados), rodam em uma stream
- Quando nenhuma stream é especificada, o stream padrão (ou “null stream”) é utilizado.
- O stream padrão é diferente dos demais streams, pois ele sincroniza as operações executadas no dispositivo:
 - ▶ Nenhuma operação no stream padrão irá começar até que todas as operações anteriormente enviadas, para qualquer stream, tenham sido completadas
 - ▶ Operação no stream padrão deve ser completada antes de qualquer outra operação, de qualquer outro stream no dispositivo, ser iniciada

Transferência assíncrona de dados

Transferência síncrona

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, sob o ponto de vista do dispositivo, todas as três operações são enviadas para o mesmo stream (o padrão), e são executadas na ordem em que foram enviadas

Transferência assíncrona de dados

Transferência síncrona

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, sob o ponto de vista do dispositivo, todas as três operações são enviadas para o mesmo stream (o padrão), e são executadas na ordem em que foram enviadas
- Sob o ponto de vista do host, a transferência de dados é síncrona, enquanto que o lançamento da função kernel é assíncrona

Transferência assíncrona de dados

Transferência síncrona

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, sob o ponto de vista do dispositivo, todas as três operações são enviadas para o mesmo stream (o padrão), e são executadas na ordem em que foram enviadas
- Sob o ponto de vista do host, a transferência de dados é síncrona, enquanto que o lançamento da função kernel é assíncrona
- Uma vez que a transferência de dados do host para o dispositivo na primeira linha é síncrona, a chamada a função kernel na segunda linha só será efetuada pelo host, após o término da transferência de dados do host para o dispositivo

Transferência assíncrona de dados

Transferência síncrona

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, sob o ponto de vista do dispositivo, todas as três operações são enviadas para o mesmo stream (o padrão), e são executadas na ordem em que foram enviadas
- Sob o ponto de vista do host, a transferência de dados é síncrona, enquanto que o lançamento da função kernel é assíncrona
- Uma vez que a transferência de dados do host para o dispositivo na primeira linha é síncrona, a chamada a função kernel na segunda linha só será efetuada pelo host, após o término da transferência de dados do host para o dispositivo
- Uma vez que o kernel é enviado para execução, em seguida o host vai para a terceira linha, e a transferência só é iniciada após o término da execução do kernel no dispositivo

Transferência assíncrona de dados

Transferência síncrona/Execução assíncrona no *host*

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
myCpuFunction(b)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, tão logo o kernel `increment()` é lançado no dispositivo, o host então executa `myCpuFunction()`, **sobrepondo-se** a execução do kernel na GPU

Transferência assíncrona de dados

Transferência síncrona/Execução assíncrona no *host*

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
myCpuFunction(b)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, tão logo o kernel `increment()` é lançado no dispositivo, o host então executa `myCpuFunction()`, **sobrepondo-se** a execução do kernel na GPU

Transferência assíncrona de dados

Transferência síncrona/Execução assíncrona no *host*

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
myCpuFunction(b)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, tão logo o kernel `increment()` é lançado no dispositivo, o host então executa `myCpuFunction()`, **sobrepondo-se** a execução do kernel na GPU
- Do ponto de vista do dispositivo, nada se altera com relação ao exemplo anterior

Transferência assíncrona de dados

Transferência síncrona/Execução assíncrona no *host*

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
myCpuFunction(b)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- No código acima, tão logo o kernel `increment()` é lançado no dispositivo, o host então executa `myCpuFunction()`, **sobrepondo-se** a execução do kernel na GPU
- Do ponto de vista do dispositivo, nada se altera com relação ao exemplo anterior
- A execução do kernel no dispositivo não é afetada pela execução da função `myCpuFunction()` no host

Transferência assíncrona de dados

Streams

Em CUDA, as streams são declaradas, criadas e destruídas no host, da seguinte maneira:

```
cudaStream_t stream1;  
cudaError_t result;  
result = cudaStreamCreate(&stream1)  
result = cudaStreamDestroy(stream1)
```

Para iniciar uma transferência de dados em um stream diferente do padrão, é utilizada a função **cudaMemcpyAsync()**, que é similar à função **cudaMemcpy()**, contendo um quinto (**stream1**) parâmetro adicional:

```
result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1)
```

cudaMemcpyAsync() é não-bloqueante no *host*. Logo, o controle retorna imediatamente para o *host*, após a transferência ser emitida ao stream.

Transferência assíncrona de dados

Streams

Para enviar um kernel a um stream diferente do stream padrão, este deve ser identificado como um quarto parâmetro de execução:

```
increment<<<1, N, 0, stream1>>>> (d_a)
```

O terceiro parâmetro de configuração permite alocar memória compartilhada no dispositivo. Por ora, usa-se o valor 0

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- Já foi mostrado como sobrepor a execução de um kernel no stream padrão, com a execução de um código no host

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- Já foi mostrado como sobrepor a execução de um kernel no stream padrão, com a execução de um código no host
- Nosso principal objetivo é, no entanto, mostrar como sobrepor a execução de um kernel, com transferência de dados

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- Já foi mostrado como sobrepor a execução de um kernel no stream padrão, com a execução de um código no host
- Nosso principal objetivo é, no entanto, mostrar como sobrepor a execução de um kernel, com transferência de dados
- Há alguns requisitos que precisam ser atendidos:

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- Já foi mostrado como sobrepor a execução de um kernel no stream padrão, com a execução de um código no host
- Nosso principal objetivo é, no entanto, mostrar como sobrepor a execução de um kernel, com transferência de dados
- Há alguns requisitos que precisam ser atendidos:
 - ▶ O dispositivo deve ser capaz de transferir e executar concorrentemente
 - ▶ Isto pode ser verificado com o programa **deviceQuery**, que vem no Toolkit CUDA
 - ▶ Quase todas as GPUs com capacidade de computação igual ou superior a 1.1, tem esta propriedade

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- Já foi mostrado como sobrepor a execução de um kernel no stream padrão, com a execução de um código no host
- Nosso principal objetivo é, no entanto, mostrar como sobrepor a execução de um kernel, com transferência de dados
- Há alguns requisitos que precisam ser atendidos:
 - ▶ O dispositivo deve ser capaz de transferir e executar concorrentemente
 - ▶ Isto pode ser verificado com o programa **deviceQuery**, que vem no Toolkit CUDA
 - ▶ Quase todas as GPUs com capacidade de computação igual ou superior a 1.1, tem esta propriedade
 - ▶ A execução do kernel e a transferência de dados devem ocorrerem em streams distintos, e diferentes do stream padrão

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- Já foi mostrado como sobrepor a execução de um kernel no stream padrão, com a execução de um código no host
- Nosso principal objetivo é, no entanto, mostrar como sobrepor a execução de um kernel, com transferência de dados
- Há alguns requisitos que precisam ser atendidos:
 - ▶ O dispositivo deve ser capaz de transferir e executar concorrentemente
 - ▶ Isto pode ser verificado com o programa **deviceQuery**, que vem no Toolkit CUDA
 - ▶ Quase todas as GPUs com capacidade de computação igual ou superior a 1.1, tem esta propriedade
 - ▶ A execução do kernel e a transferência de dados devem ocorrerem em streams distintos, e diferentes do stream padrão
 - ▶ A memória do host a ser utilizada na transferência de dados deve ser do tipo **pinned**

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

async.cu

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);  
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);  
}
```

- No código acima, o array de tamanho **N** é dividido em pedados (chunks) de tamanho **streamSize**

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

`async.cu`

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);  
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);  
}
```

- No código acima, o array de tamanho **N** é dividido em pedados (chunks) de tamanho **streamSize**
- Dado que o kernel é executado de forma independente em todos os elementos, cada um dos pedados podem ser processados em paralelo

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

async.cu

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);  
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);  
}
```

- No código acima, o array de tamanho **N** é dividido em pedados (chunks) de tamanho **streamSize**
- Dado que o kernel é executado de forma independente em todos os elementos, cada um dos pedados podem ser processados em paralelo
- O número de streams usados será **nStreams=N/streamSize**

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

`async.cu`

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);  
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);  
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);  
}
```

- No código acima, o array de tamanho **N** é dividido em pedados (chunks) de tamanho **streamSize**
- Dado que o kernel é executado de forma independente em todos os elementos, cada um dos pedados podem ser processados em paralelo
- O número de streams usados será **nStreams=N/streamSize**
- O laço é aplicado a todas as operações para cada pedado do array

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

async.cu

```
for (int i = 0; i < nStreams; ++i) {  
    int offset = i * streamSize;  
    cudaMemcpyAsync(&d_a[offset], &a[offset],  
                   streamBytes, cudaMemcpyHostToDevice, stream[i]);  
}
```

- Outra abordagem possível, consiste em se fazer antes todas as transferências do host para o dispositivo

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

async.cu

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset],
                   streamBytes, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}
```

- Em seguida, todos os kernels são lançados

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

`async.cu`

```
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset],
                   streamBytes, cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&a[offset], &d_a[offset],
                   streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}
```

- Por último, todas as transferências do dispositivo para o host são então realizadas

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

```
$ cd codigos/blogforall/cuda-cpp/overlap-data-transfers
$ make
$ ./async
```

Device : GeForce GTX 285

Time for sequential transfer and execute (ms): 10.976416

max error: 2.384186e-07

Time for asynchronous V1 transfer and execute (ms): 11.859616

max error: 2.384186e-07

Time for asynchronous V2 transfer and execute (ms): 7.000256

max error: 2.384186e-07

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

```
$ cd codigos/blogforall/cuda-cpp/overlap-data-transfers  
$ make  
$ ./async
```

Device : Tesla C2050

Time for sequential transfer and execute (ms): 8.776512

max error: 1.192093e-07

Time for asynchronous V1 transfer and execute (ms): 5.005696

max error: 1.192093e-07

Time for asynchronous V2 transfer and execute (ms): 6.655712

max error: 1.192093e-07

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

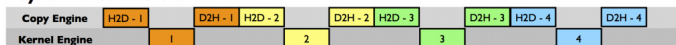
- GTX 285:
 - ▶ Compute Capability 1.3
 - ▶ 1 copy engine and 1 kernel engine

Execution Time Lines

Sequential Version



Asynchronous Version 1



Asynchronous Version 2



Time →

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

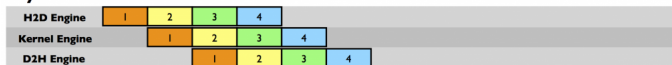
- C 2050:
 - ▶ Compute Capability 2.0
 - ▶ 2 copy engines and 1 kernel engine

Execution Time Lines

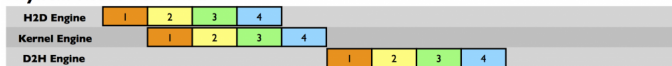
Sequential Version



Asynchronous Version 1



Asynchronous Version 2



Time →

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

```
$ cd codigos/blogforall/cuda-cpp/overlap-data-transfers
$ make
$ ./async
```

Device : Tesla K20c

Time for sequential transfer and execute (ms): 8.841568

max error: 1.192093e-07

Time for asynchronous V1 transfer and execute (ms): 6.481280

max error: 1.192093e-07

Time for asynchronous V2 transfer and execute (ms): 6.388800

max error: 1.192093e-07

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

```
$ cd codigos/blogforall/cuda-cpp/overlap-data-transfers
$ make
$ srun -p treinamento_gpu ./async
```

Device : Tesla K40t

Time for sequential transfer and execute (ms): 4.507520

max error: 1.192093e-07

Time for asynchronous V1 transfer and execute (ms): 2.597024

max error: 1.192093e-07

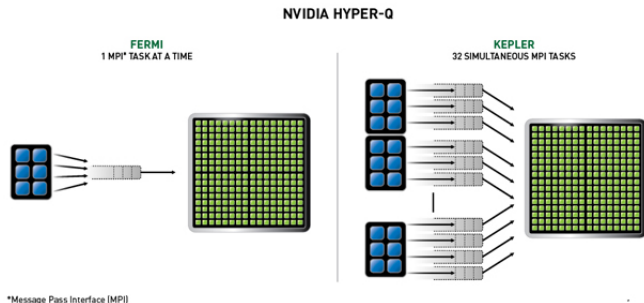
Time for asynchronous V2 transfer and execute (ms): 2.595488

max error: 1.192093e-07

Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

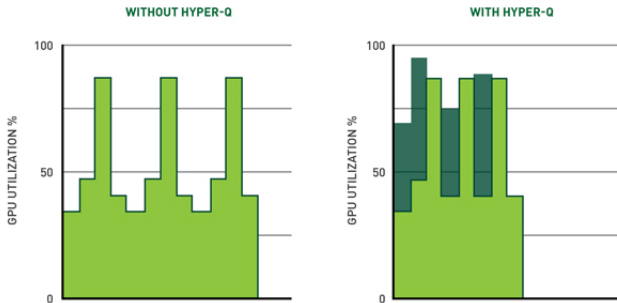
- A partir da Tesla K20 (compute capability 3.5), pode-se fazer chamadas de mais de um kernel simultaneamente, realizadas por um ou mais processos do host (HYPER-Q)



Transferência assíncrona de dados

Sobreposição execução do kernel/transferência de dados

- A partir da Tesla K20 (compute capability 3.5), pode-se fazer chamadas de mais de um kernel simultaneamente, realizadas por um ou mais processos do host (HYPER-Q)



Roteiro

- 1 Módulo 1: uso eficiente de memória
- 2 Módulo 2: transferência otimizada de dados
- 3 Módulo 3: transferência assíncrona de dados
- 4 **Módulo 4: processamento em múltiplas GPUs**

Processamento em múltiplas GPUs

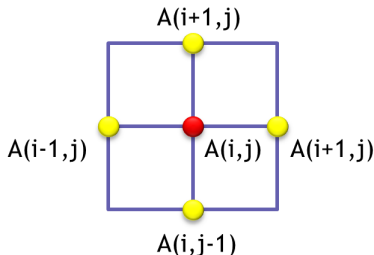
Motivos para utiliza múltiplas GPUs

- **Rapidez** - múltiplas GPUs significa obter um tempo de solução mais rápido.
- **Escalabilidade** - múltiplas GPUs significa haver mais memória para tratar problemas grandes.
- **Custo** - múltiplas GPUs por nó computacional significa um melhor retorno por investimento financeiro, energético e espaço no *data center*.

Processamento em múltiplas GPUs

Solver Laplace 2D

Dada uma grade 2D de vértices, o *solver* tenta definir todos os vértices iguais à média dos vértices vizinhos. Ele será repetido até o sistema convergir para um valor estável. Portanto, para um determinado vértice e seus vizinhos:



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Processamento em múltiplas GPUs

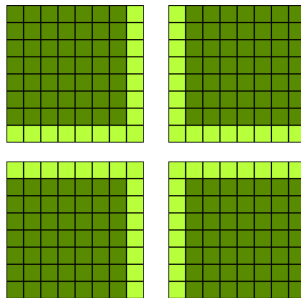
Decomposição de domínio

Opções para dividir a grade 2D de vértices, ou domínio, para paralelizar o trabalho entre as várias GPUs. A região do halo mostrada em verde claro nas imagens são os dados que precisam ser compartilhados entre as GPUs que trabalham no problema.

Processamento em múltiplas GPUs

- Minimiza a área de comunicação:

- ▶ Menor tráfego de dados
- ▶ Bom para *bandwidth-bound communication*

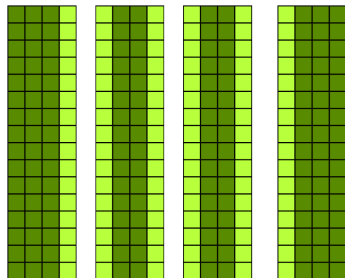


Tiles

Processamento em múltiplas GPUs

- Minimiza o número de vizinhos:

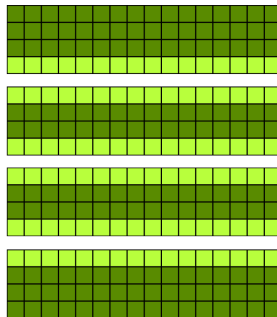
- ▶ comunicação com menos vizinhos
- ▶ Bom para *latency-bound communication*
- ▶ Orientação *column-major* (Fortran)



Vertical Stripes

Processamento em múltiplas GPUs

- Minimiza o número de vizinhos:
 - ▶ comunicação com menos vizinhos
 - ▶ Bom para *latency-bound communication*
 - ▶ Orientação *row-major* (C/C++)



Horizontal Stripes

Processamento em múltiplas GPUs: **task1**

Tarefa 1

```
$ cd codigos/multigpu/task1
$ nvcc -o task1 task1.cu -Xcompiler -fopenmp
$ srun -p treinamento_gpu ./task1
```

OMP Threads: 1

Num GPUs: 2

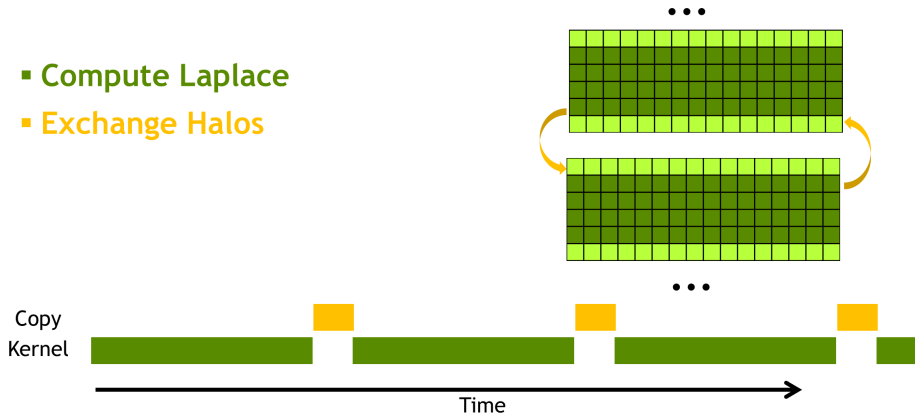
2048x2048: solve time, 1 GPU: 0.536904 s, 2 GPUs: 0.528833 s,
speedup: 1.015260, efficiency: 50.7631%

4096x4096: solve time, 1 GPU: 2.097620 s, 2 GPUs: 2.113510 s,
speedup: 0.992484, efficiency: 49.6242%

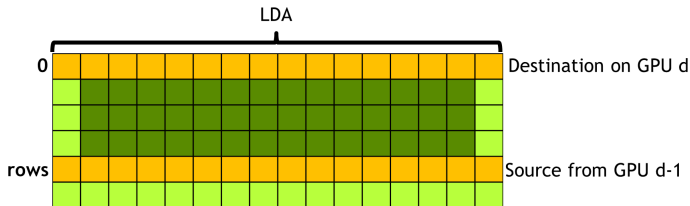
done

Processamento em múltiplas GPUs: `task2`

- **Compute Laplace**
- **Exchange Halos**



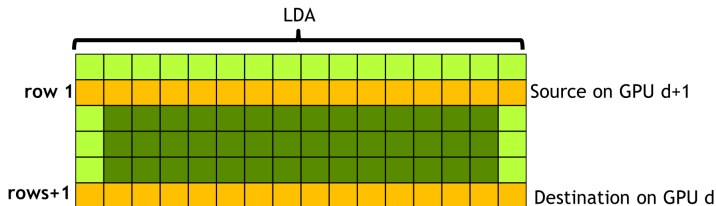
Processamento em múltiplas GPUs: **task2**



Grab lower boundary

```
if (d>0)
    cudaMemcpyPeerAsync(d_Aout[d], d,
        d_Aout[d-1]+IDX(...), d-1,
        LDA*sizeof(double), 0);
```

Processamento em múltiplas GPUs: task2



Grab upper boundary

```
if (d < numDev - 1)
    cudaMemcpyPeerAsync(d_Aout[d] + IDX(...), d,
                        d_Aout[d+1] + IDX(...), d+1,
                        LDA * sizeof(double), 0);
```

Processamento em múltiplas GPUs: task2

Tarefa 2

```
$ cd codigos/multigpu/task2  
$ nvcc -o task2 task2.cu -Xcompiler -fopenmp  
$ srun -p treinamento_gpu ./task2
```

OMP Threads: 1

Num GPUs: 2

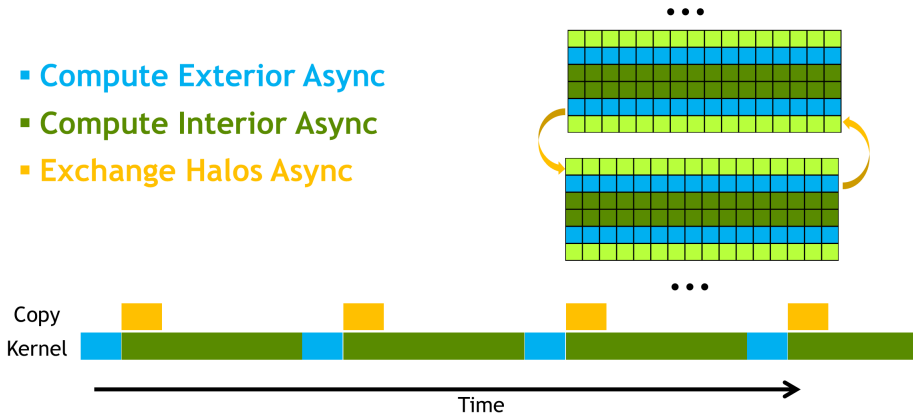
2048x2048: solve time, 1 GPU: 0.531551 s, 2 GPUs: 0.317938 s,
speedup: 1.671870, efficiency: 83.5935%

4096x4096: solve time, 1 GPU: 2.097870 s, 2 GPUs: 1.113530 s,
speedup: 1.883980, efficiency: 94.1990%

done

Processamento em múltiplas GPUs: task3

- Compute Exterior Async
- Compute Interior Async
- Exchange Halos Async



Processamento em múltiplas GPUs

Tarefa 3

```
$ cd codigos/multigpu/task3  
$ nvcc -o task3 task3.cu -Xcompiler -fopenmp  
$ srun -p treinamento_gpu ./task3
```

OMP Threads: 1

Num GPUs: 2

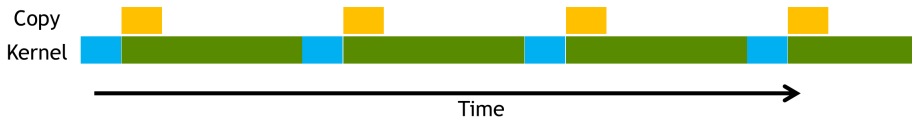
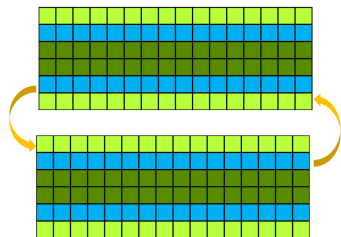
2048x2048: solve time, 1 GPU: 0.526863 s, 2 GPUs: 0.335753 s,
speedup: 1.569200, efficiency: 78.4599%

4096x4096: solve time, 1 GPU: 2.097850 s, 2 GPUs: 1.13348 s,
speedup: 1.850810, efficiency: 92.5407%

done

Processamento em múltiplas GPUs: task4

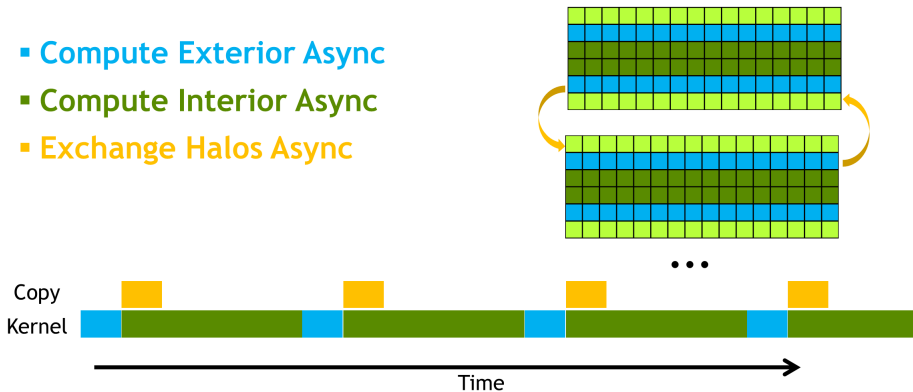
- Compute Exterior Async
- Compute Interior Async
- Exchange Halos Async



- Nesta etapa, criaremos 3 fluxos, 1 para kernels e um para transferências de cada uma das regiões de fronteira.

Processamento em múltiplas GPUs: task4

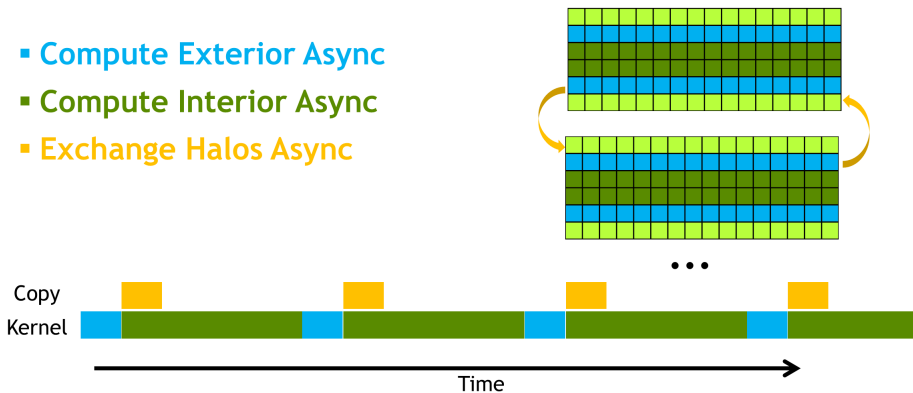
- Compute Exterior Async
- Compute Interior Async
- Exchange Halos Async



- Nesta etapa, criaremos 3 fluxos, 1 para kernels e um para transferências de cada uma das regiões de fronteira.
- *Streams* nos permitem obter execução simultânea assíncrona criando filas de trabalho independente.

Processamento em múltiplas GPUs: task_4

- Compute Exterior Async
- Compute Interior Async
- Exchange Halos Async



- Nesta etapa, criaremos 3 fluxos, 1 para kernels e um para transferências de cada uma das regiões de fronteira.
- *Streams* nos permitem obter execução simultânea assíncrona criando filas de trabalho independente.
- Vamos ignorar a sincronização entre os *streams* nesta etapa...

Processamento em múltiplas GPUs: task4

Tarefa 4

```
$ cd codigos/multigpu/task4
$ nvcc -o task4 task4.cu -Xcompiler -fopenmp
$ srun -p treinamento_gpu ./task4

OMP Threads: 1
Num GPUs: 2
2048x2048: Error solutions do not match at i: 848, j: 176
2048x2048: solve time, 1 GPU: 0.540329 s, 2 GPUs: 0.278882 s,
           speedup: 1.937480, efficiency: 96.8742%
4096x4096: Error solutions do not match at i: 1879, j: 176
4096x4096: solve time, 1 GPU: 2.097490 s, 2 GPUs: 1.0714 s,
           speedup: 1.957720, efficiency: 97.8858%
done
```

Processamento em múltiplas GPUs: task5

Streams sincronizados

Tarefa 5

```
$ cd codigos/multigpu/task5
$ nvcc -o task5 task5.cu -Xcompiler -fopenmp
$ srun -p treinamento_gpu ./task5
```

OMP Threads: 1

Num GPUs: 2

2048x2048: solve time, 1 GPU: 0.523688 s, 2 GPUs: 0.279833 s,
speedup: 1.871430, efficiency: 93.5716%

4096x4096: solve time, 1 GPU: 2.097390 s, 2 GPUs: 1.072340 s,
speedup: 1.955890, efficiency: 97.7946%

done

Agradecimentos



Laboratório
Nacional de
Computação
Científica

