

# MC-SD03-I

## Introdução à Programação CUDA

Programa de Verão do LNCC 2021

Escola Santos Dumont

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

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 Modelo de Paralelismo
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais

# Roteiro

## 1 Módulo 1: Introdução

- Motivação
- Primeiros Passos

## 2 Modelo de Paralelismo

- Módulo 2: Hierarquia, Organização e Identificação
- Módulo 3: Atribuição
- Módulo 4: Escalonamento
- Módulo 5: Hierarquia de Memória

## 3 Módulo 6: Métricas e Otimização de Desempenho

## 4 Módulo 7: Multiplicação de Matrizes

## 5 Considerações Finais

# Roteiro

## 1 Módulo 1: Introdução

- **Motivação**
- Primeiros Passos

## 2 Modelo de Paralelismo

- Módulo 2: Hierarquia, Organização e Identificação
- Módulo 3: Atribuição
- Módulo 4: Escalonamento
- Módulo 5: Hierarquia de Memória

## 3 Módulo 6: Métricas e Otimização de Desempenho

## 4 Módulo 7: Multiplicação de Matrizes

## 5 Considerações Finais

# Motivação

## Por que usar GPU

- Arquitetura com grande potencial de paralelismo, contendo centenas de núcleos computacionais (*cores*);

# Motivação

## Por que usar GPU

- Arquitetura com grande potencial de paralelismo, contendo centenas de núcleos computacionais (*cores*);
- Plataforma de computação de alto desempenho presente em milhões de máquinas;

# Motivação

## Por que usar GPU

- Arquitetura com grande potencial de paralelismo, contendo centenas de núcleos computacionais (*cores*);
- Plataforma de computação de alto desempenho presente em milhões de máquinas;
- Melhor relação desempenho/custo (**Gflops/\$**)

# Motivação

## Por que usar GPU

- Arquitetura com grande potencial de paralelismo, contendo centenas de núcleos computacionais (*cores*);
- Plataforma de computação de alto desempenho presente em milhões de máquinas;
- Melhor relação desempenho/custo (Gflops/\$)
- Melhor relação desempenho/consumo (Gflops/Watts)



# Motivação

## Por que usar GPU

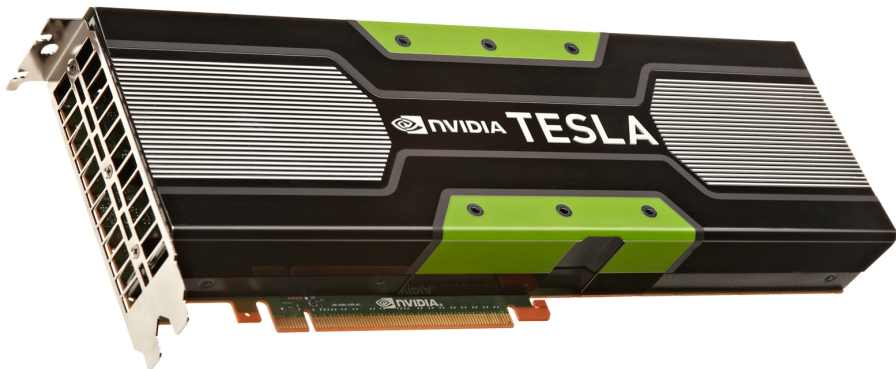
- Arquitetura com grande potencial de paralelismo, contendo centenas de núcleos computacionais (*cores*);
- Plataforma de computação de alto desempenho presente em milhões de máquinas;
- Melhor relação desempenho/custo (Gflops/\$)
- Melhor relação desempenho/consumo (Gflops/Watts)
- Melhor relação desempenho/volume (Gflops/Volume)

# Motivação

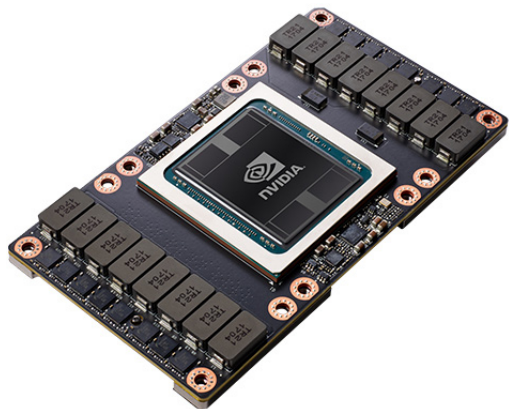
## Por que usar GPU

- Arquitetura com grande potencial de paralelismo, contendo centenas de núcleos computacionais (*cores*);
- Plataforma de computação de alto desempenho presente em milhões de máquinas;
- Melhor relação desempenho/custo (Gflops/\$)
- Melhor relação desempenho/consumo (Gflops/Watts)
- Melhor relação desempenho/volume (Gflops/Volume)
- **Inteligência Artificial/Aprendizado de Máquina**

# NVIDIA Tesla K40

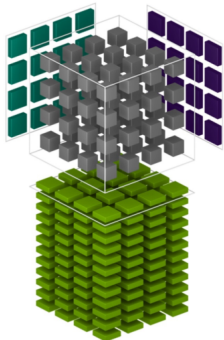


# NVIDIA Volta V100



- 5376 núcleos FP32
- 5376 núcleos INT32
- 2688 núcleos FP64
- 672 Tensor Cores (TC)

# NVIDIA Volta V100



**Tensor Core**

$$D = \begin{matrix} \text{FP16 or FP32} & \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} & \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} & \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} \\ & \text{FP16} & \text{FP16} & \text{FP16 or FP32} \end{matrix} +$$

**Precisão mista  
(computação aproximada)**

# Compute Unified Device Architecture - CUDA

## Aplicações não-gráficas em GPU

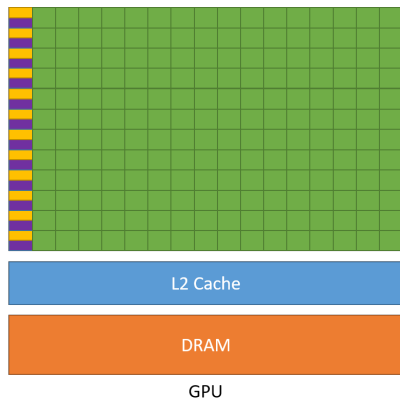
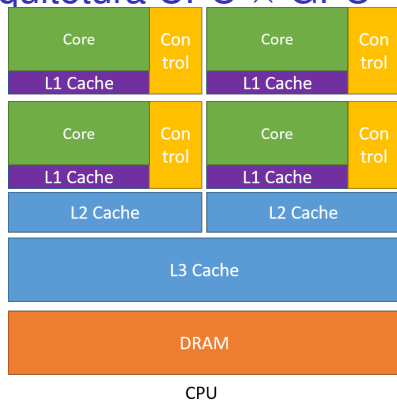
- Em Novembro de 2006 foi lançada a série G80 de GPUs, com a GeForce 8800 GTX, que introduziu entre outras inovações:

Suporte linguagem C, por meio das extensões inseridas na CUDA  
O programador tem acesso ao poder computacional da GPU, sem precisar aprender nova linguagem

Processador que unificou as operações em vértices, geometria e pixel a execução de programas

- “Programming Massively Parallel Processors: A Hands-on Approach” [Kirk and Hwu, 2012]
- “CUDA by Example: An Introduction to General-Purpose GPU Programming” [Sanders and Kandrot, 2010]
- “CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs” [Cook, 2012]
- “The CUDA Handbook: A Comprehensive Guide to GPU Programming” [Wilt, 2013]

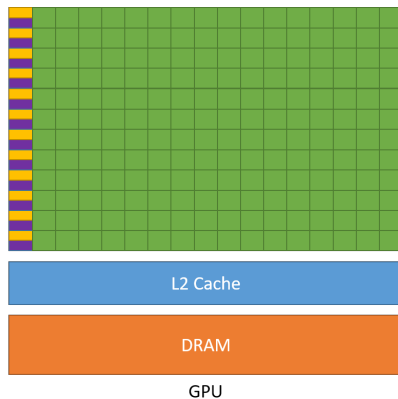
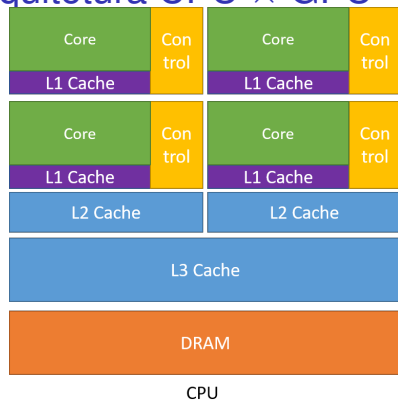
# Arquitetura CPU × GPU



- CPU: historicamente, priorizava-se a otimização da execução do código sequencial;

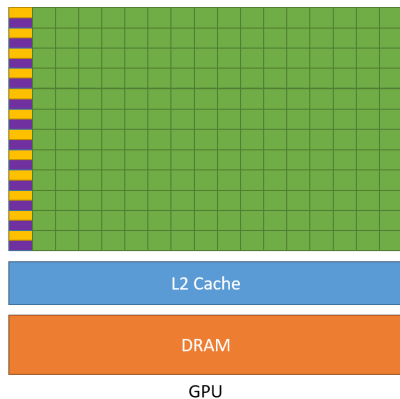
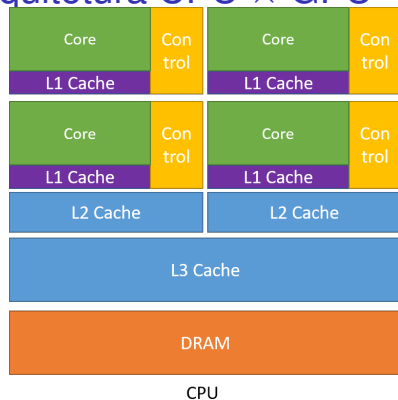


# Arquitetura CPU × GPU



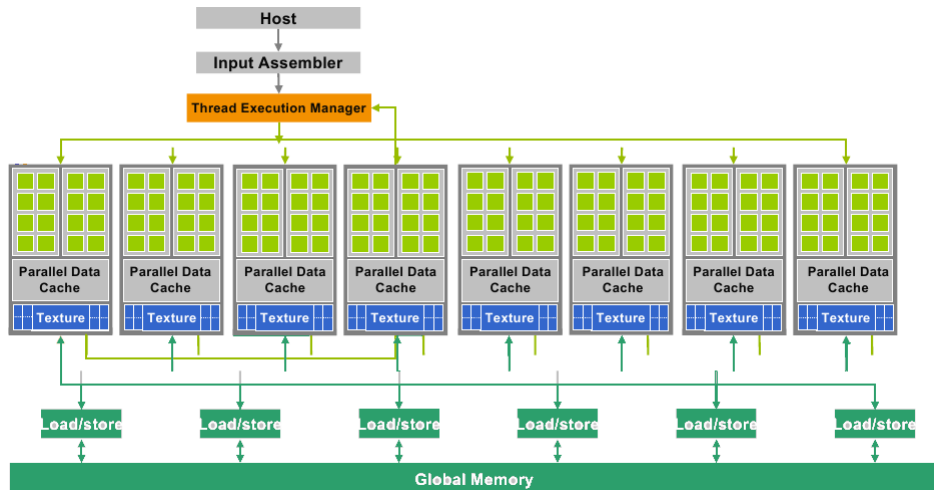
- CPU: historicamente, priorizava-se a otimização da execução do código sequencial;
- CPU: mais recentemente, buscou-se maior desempenho com o emprego de arquitetura *multi-core*;

# Arquitetura CPU × GPU

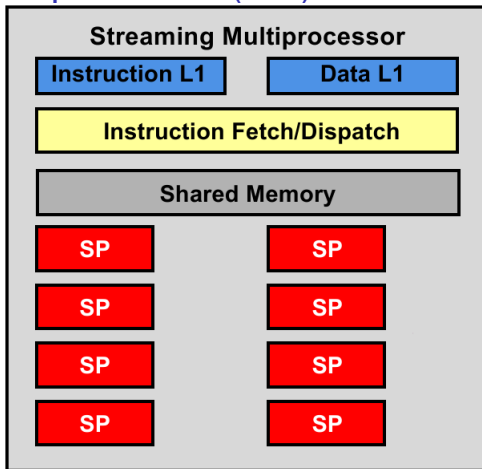


- CPU: historicamente, priorizava-se a otimização da execução do código sequencial;
- CPU: mais recentemente, buscou-se maior desempenho com o emprego de arquitetura *multi-core*;
- GPU: execução paralela em arquitetura *many-core*;

# Arquitetura GPU: série G80 (NVIDIA)

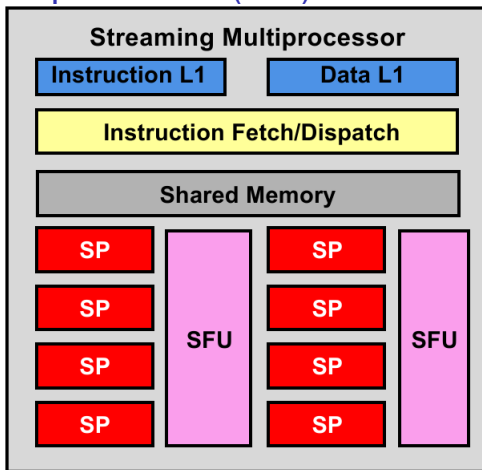


# Streaming Multiprocessor (SM)



- 8 Streaming Processor (SP)

# Streaming Multiprocessor (SM)



- 8 Streaming Processor (SP)
- 2 Special Function Unit (SFU)

# Desempenho máximo teórico $R_{peak}$

## NVIDIA GeForce 8800 GTX (Q406):

- 16 SMs com 8 SPs cada: 128 *cores*;
- Cada SP pode executar 2 unidades MAD (multiply-add) e 1 unidade MUL (multiply) por ciclo de *clock*;
- $FLOPS \approx f \times SM \times (8SP * (2MAD + 1MUL))$   
 $FLOPS \approx 1.35GHz \times 16 \times (8 * (2flop + 1flop))$   
 $FLOPS \approx 1.35GHz \times 16 \times (8 * (3flop))$   
 $FLOPS \approx \mathbf{518.4Gflop/s}$  em precisão simples  
Esta GPU pertence a uma geração que não suportava precisão dupla
- Maior  $R_{peak}$  de uma CPU na época era de **32 Gflop/s**;
- Adicionalmente, cada SFU ainda é responsável por executar operações de funções especiais tais como raiz quadrada e transcendentais (logarítmica, exponencial, trigonométrica, etc).

# Desempenho máximo teórico $R_{peak}$

## Intel Core i7-3970X Processor Extreme Edition (Sandy Bridge):

- Possui 6 *cores* (#cores=6);
- Executando 4 operações de ponto flutuante de precisão simples por ciclo de clock (#flop/cycle=4);
- Utiliza o conjunto de instruções SIMD **AVX (Advanced Vector Extensions)**, de 256 bits
- Frequência de clock de 3.5GHz ( $f=3.5\text{GHz}$ );  
 $FLOPS \approx f \times \#cores \times flop/cycle \times \#simd\_avx$   
 $FLOPS \approx 3.5\text{GHz} \times 6 \times 4 \times 8$   
 $FLOPS \approx 672\text{Gflop/s}$  (precisão simples)  
 $FLOPS \approx 336\text{Gflop/s}$  (precisão dupla)

# Desempenho máximo teórico $R_{peak}$

## GeForce GTX 580 (Fermi):

- 512 *cores*;
  - Frequência de clock de 1.544GHz ( $f=1.544\text{GHz}$ );
  - Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
  - $FLOPS \approx f \times \#cores \times flop/cycle$ ;
  - $FLOPS \approx 1.544 \times 512 \times 2$ ;
  - Desempenho teórico em precisão simples de **1581** Gflop/s;
  - Desempenho teórico em precisão dupla de **197** Gflop/s:
- 1/8 da precisão simples;**



# Desempenho máximo teórico $R_{peak}$

Tesla C 2070 (Fermi):

- 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz);
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;
- $FLOPS \approx 1.15 \times 448 \times 2$ ;
- Desempenho teórico em precisão simples de **1030,4** Gflop/s;
- Desempenho teórico em precisão dupla de **512,2** Gflop/s:  
**1/2 da precisão simples**;

# Desempenho máximo teórico $R_{peak}$

## GeForce GTX 690 (Kepler):

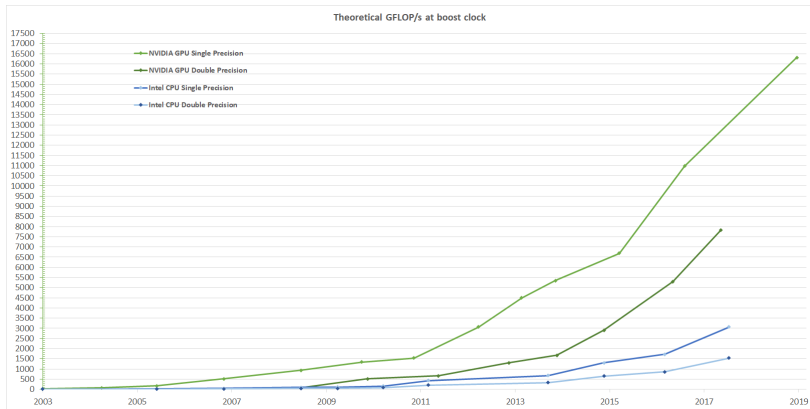
- 3072 *cores*;
- Frequência de clock de 0.915 GHz ( $f=0.915$  GHz);
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;
- $FLOPS \approx 0.915 \times 3072 \times 2$ ;
- Desempenho teórico em precisão simples de **5683,2** Gflop/s (5,68 Tflop/s);
- Desempenho teórico em precisão dupla de **236,8** Gflop/s:  
**1/24 da precisão simples;**

# Desempenho máximo teórico $R_{peak}$

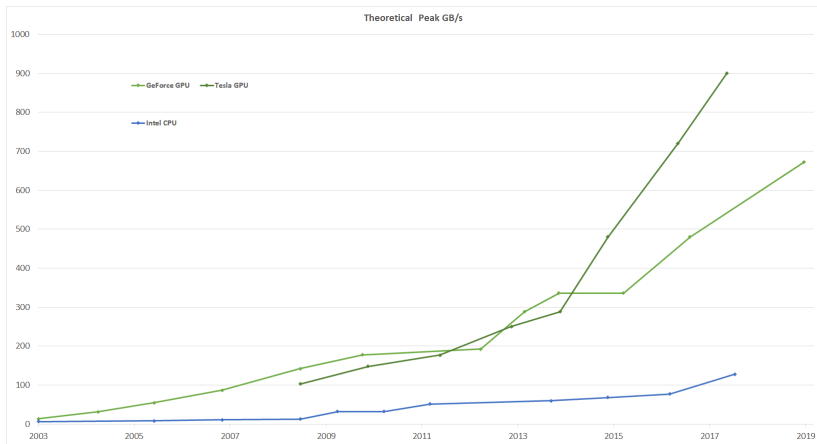
## K20 (Kepler):

- 2496 *cores*;
- Frequência de clock de 0.705 GHz ( $f=0.705$  GHz);
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;
- $FLOPS \approx 0.705 \times 2688 \times 2$ ;
- Desempenho teórico em precisão simples de **3519** Gflop/s (3,79 Tflop/s);
- Desempenho teórico em precisão dupla de **1173** Gflop/s (1,17 Tflop/s):  
**1/3 da precisão simples**;

# Desempenho máximo teórico ( $R_{peak}$ ): CPU $\times$ GPU







# Largura de banda: CPU × GPU



# Compute Unified Device Architecture - CUDA

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CUDA MAGMA	Thrust NPP	VSIP, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	
CUDA-enabled NVIDIA GPUs						
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series		
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series		
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series		
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series		
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series		
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER		

# Compute Unified Device Architecture - CUDA

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIP SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
NVIDIA Ampere Architecture (compute capabilities 8.x)					Tesla A Series	
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series		Tesla T Series	
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series		Tesla V Series	
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series		Tesla P Series	
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center		

# CUDA Toolkit

Pacote principal composto de compilador C NVIDIA, de ferramentas e aplicativos de análise e depuração de execução, contendo também bibliotecas matemáticas otimizadas para CUDA.

CUDA Toolkit (Versão atual 11.2):

- Compilador C/C++ (`nvcc`)
- Ferramenta de perfil de desempenho: `nvprof`
- NVIDIA Visual Profiler (`nvvp`)
- NVIDIA Developer Tools: Nsight Compute
- CUDA-GDB (`cuda-gdb`)
- CUDA Memcheck (`cuda-memcheck`)
- bibliotecas otimizadas para GPU



# Conhecendo a GPU na máquina

```
$ cd $SCRATCH
$ tar xvf codigos-cuda-lncc.tar.gz
$ cd codigos/
$ source env_cuda
```

# Conhecendo a GPU na máquina

```
$ cd samples/1_Uutilities/deviceQuery/  
$ make  
$ srun -p treinamento_gpu ./deviceQuery
```

```
./deviceQuery Starting...
```

```
CUDA Device Query (Runtime API) version (CUDA static linking)
```

```
Detected 2 CUDA Capable device(s)
```

```
Device 0: "Tesla K40t"
```

```
CUDA Driver Version / Runtime Version          10.0 / 10.0  
CUDA Capability Major/Minor version number:    3.5  
Total amount of global memory:                 11441 MBytes (11996954624 bytes)  
(15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores  
GPU Max Clock rate:                            745 MHz (0.75 GHz)  
Memory Clock rate:                             3004 Mhz  
Memory Bus Width:                              384-bit  
L2 Cache Size:                                 1572864 bytes  
Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)  
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers  
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers  
Total amount of constant memory:               65536 bytes  
Total amount of shared memory per block:       49152 bytes  
Total number of registers available per block: 65536  
Warp size:                                     32  
Maximum number of threads per multiprocessor:  2048  
Maximum number of threads per block:           1024
```

# Conhecendo a GPU na máquina (cont.)

```
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Enabled
Device supports Unified Addressing (UVA): Yes
Device supports Compute Preemption: No
Supports Cooperative Kernel Launch: No
Supports MultiDevice Co-op Kernel Launch: No
Device PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
Compute Mode:
  < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

## Device 1: "Tesla K40t"

```
CUDA Driver Version / Runtime Version 10.0 / 10.0
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory: 11441 MBytes (11996954624 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP: 2880 CUDA Cores
GPU Max Clock rate: 745 MHz (0.75 GHz)
Memory Clock rate: 3004 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 1572864 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
```

# Conhecendo a GPU na máquina (cont.)

```
Total amount of constant memory:          65536 bytes
Total amount of shared memory per block:   49152 bytes
Total number of registers available per block: 65536
Warp size:                                 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block:       1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                      2147483647 bytes
Texture alignment:                         512 bytes
Concurrent copy and kernel execution:      Yes with 2 copy engine(s)
Run time limit on kernels:                 No
Integrated GPU sharing Host Memory:        No
Support host page-locked memory mapping:   Yes
Alignment requirement for Surfaces:        Yes
Device has ECC support:                    Enabled
Device supports Unified Addressing (UVA):  Yes
Device supports Compute Preemption:       No
Supports Cooperative Kernel Launch:       No
Supports MultiDevice Co-op Kernel Launch: No
Device PCI Domain ID / Bus ID / location ID: 0 / 129 / 0
```

Compute Mode:

```
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from Tesla K40t (GPU0) -> Tesla K40t (GPU1) : No
> Peer access from Tesla K40t (GPU1) -> Tesla K40t (GPU0) : No
```

```
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.0, CUDA Runtime Version = 10.0, NumDev
Result = PASS
```

# Roteiro

## 1 Módulo 1: Introdução

- Motivação
- **Primeiros Passos**

## 2 Modelo de Paralelismo

- Módulo 2: Hierarquia, Organização e Identificação
- Módulo 3: Atribuição
- Módulo 4: Escalonamento
- Módulo 5: Hierarquia de Memória

## 3 Módulo 6: Métricas e Otimização de Desempenho

## 4 Módulo 7: Multiplicação de Matrizes

## 5 Considerações Finais

# Um primeiro programa CUDA

Os códigos apresentados foram retirados e adaptados do livro de [Sanders and Kandrot, 2010], podendo ser baixados de:

<https://developer.nvidia.com/cuda-example>

# Primeiros Passos

## hello\_world.cu

```
/*
 * Copyright 1993-2010 NVIDIA Corporation. All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual property and
 * proprietary rights in and to this software and related documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement (EULA)
 * associated with this source code for terms and conditions that govern
 * your use of this NVIDIA software.
 */

#include <stdio.h>

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

# Primeiros Passos

```
$ cd primeirospassos  
$ nvcc hello_world.cu -o hello_world  
$ srun -p treinamento_gpu ./hello_world  
Hello, World!
```



# Primeiros Passos: chamada a *kernel*

## `simple_kernel.cu`

```
#include <stdio.h>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- `__global__`: qualificador CUDA que informa ao compilador que esta função será executada na GPU (*device*), e não na CPU (*host*).
- `kernel<<<1,1>>>()`: os argumentos entre `<<< >>>` definem a configuração da execução na GPU. Neste exemplo a função `kernel` nada faz na GPU.

```
$ nvcc simple_kernel.cu -o simple_kernel
$ srun -p treinamento_gpu ./simple_kernel
Hello, World!
```

# Primeiros Passos: passando parâmetros

## simple\_kernel\_params.cu

```
#include <stdio.h>

// Convenience function for checking CUDA runtime API results
// can be wrapped around any runtime API call. No-op in release builds.
inline
cudaError_t checkCuda(cudaError_t result)
{
    #if defined(DEBUG) || defined(_DEBUG)
        if (result != cudaSuccess) {
            fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
            assert(result == cudaSuccess);
        }
    #endif
    return result;
}

__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    checkCuda( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );
}
```

## Primeiros Passos: passando parâmetros (cont.)

```
checkCuda( cudaMemcpy( &c, dev_c, sizeof(int),
                      cudaMemcpyDeviceToHost ) );
printf( "2 + 7 = %d\n", c );
checkCuda( cudaFree( dev_c ) );

return 0;
}
```

- **cudaMalloc**: função da API CUDA que aloca memória do GPU.
- **cudaMemcpy**: função da API CUDA que faz a transferência de dados entre o GPU e o CPU.
- **cudaMemcpyDeviceToHost**: parâmetro que define o sentido da transferência.
- **cudaFree**: função da API CUDA que libera a memória alocada.

## Primeiros Passos: passando parâmetros (cont.)

```
$ nvcc simple_kernel_params.cu -o simple_kernel_params
```

```
$ srun -p treinamento_gpu ./simple_kernel_params
```

```
2 + 7 = 9
```

# Soma de vetores

```
$ cd vecSum
```

# Soma de vetores na CPU

## add\_cpu.cu

```
void add( int N, float *a, float *b, float *c ) {  
    int tid = 0;    // this is CPU zero, so we start at zero  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += 1;    // we have one CPU, so we increment by one  
    }  
}
```

## add\_cpu.cu

```
add( a, b, c );
```

# Soma de vetores na GPU

## add\_gpu.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {  
    int tid = threadIdx.x;    // handles the data at its thread id  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
  
}
```

## add\_gpu.cu

```
add<<<1,N>>>( N, dev_a, dev_b, dev_c );
```

# Soma de vetores na GPU

## add\_gpu.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {  
    int tid = threadIdx.x;    // handles the data at its thread id  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

## add\_gpu.cu

```
add<<<1,N>>>( N, dev_a, dev_b, dev_c );
```

- **threadIdx.x**: palavra-chave que referencia os índices das *threads*;
- **kernel**<<<1,N>>>(): os argumentos entre <<< >>> definem a configuração/geometria de execução na GPU. Neste exemplo o *kernel* **add** soma dois vetores de N elementos (uma *thread* para cada elemento dos vetores).



# Soma de vetores na CPU

```
$ nvcc add_cpu.cu -o add_cpu
$ srun -p treinamento_gpu ./add_cpu 10
```

0 +	0 =	0
-1 +	1 =	0
-2 +	4 =	2
-3 +	9 =	6
-4 +	16 =	12
-5 +	25 =	20
-6 +	36 =	30
-7 +	49 =	42
-8 +	64 =	56
-9 +	81 =	72

# Soma de vetores na GPU

```
$ nvcc add_gpu.cu -o add_gpu
$ srun -p treinamento_gpu ./add_gpu 10
```

0 +	0 =	0
-1 +	1 =	0
-2 +	4 =	2
-3 +	9 =	6
-4 +	16 =	12
-5 +	25 =	20
-6 +	36 =	30
-7 +	49 =	42
-8 +	64 =	56
-9 +	81 =	72

# Soma de vetores na CPU: tempo de execução

```
$ nvcc add_gpu.cu -o add_gpu
$ srun -p treinamento_gpu ./add_cpu 512
```

```
-502 + 252004 = 251502
-503 + 253009 = 252506
-504 + 254016 = 253512
-505 + 255025 = 254520
-506 + 256036 = 255530
-507 + 257049 = 256542
-508 + 258064 = 257556
-509 + 259081 = 258572
-510 + 260100 = 259590
-511 + 261121 = 260610
```

```
Execution Time (microseconds):      4.00
```

O compilador `nvcc` tem como opção padrão de otimização `-O0`.

# Soma de vetores na CPU: tempo de execução

```
$ nvcc -O2 add_cpu.cu -o add_cpu
$ srun -p treinamento_gpu ./add_cpu 512
```

```
-502 + 252004 = 251502
-503 + 253009 = 252506
-504 + 254016 = 253512
-505 + 255025 = 254520
-506 + 256036 = 255530
-507 + 257049 = 256542
-508 + 258064 = 257556
-509 + 259081 = 258572
-510 + 260100 = 259590
-511 + 261121 = 260610
```

```
Execution Time (microseconds):      1.00
```

# Soma de vetores na GPU: profiling

```
$ srun -p treinamento_gpu nvprof ./add_gpu 512
```

```
-502 + 252004 = 251502  
-503 + 253009 = 252506  
-504 + 254016 = 253512  
-505 + 255025 = 254520  
-506 + 256036 = 255530  
-507 + 257049 = 256542  
-508 + 258064 = 257556  
-509 + 259081 = 258572  
-510 + 260100 = 259590  
-511 + 261121 = 260610
```

Device: Tesla C2050

==== Profiling result:

Time (%)	Time	Calls	Avg	Min	Max	Name
39.64	2.62us	2	1.31us	1.18us	1.44us	[CUDA memcpy HtoD]
34.81	2.30us	1	2.30us	2.30us	2.30us	[CUDA memcpy DtoH]
25.55	1.69us	1	1.69us	1.69us	1.69us	add(int, float*, float*, float*)

## Soma de vetores: GPU × CPU

- Na CPU: **1.00** microssegundos;
- Na GPU: **1.69** microssegundos;

## Soma de vetores: GPU × CPU

- Na CPU: **1.00** microssegundos;
- Na GPU: **1.69** microssegundos;
- Na CPU é mais rápido este tamanho de problema, sobretudo se levarmos em conta o **custo de transferência** de dados entre a GPU e a CPU.

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 **Modelo de Paralelismo**
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais



## Threads

- Hierarquia, Organização e Identificação

## Threads

- Hierarquia, Organização e Identificação
- Atribuição

## Threads

- Hierarquia, Organização e Identificação
- Atribuição
- Escalonamento

## Threads

- Hierarquia, Organização e Identificação
- Atribuição
- Escalonamento
- Hierarquia de Memória

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 **Modelo de Paralelismo**
  - **Módulo 2: Hierarquia, Organização e Identificação**
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais

# Hierarquia de Threads

- *Threads*
- Blocos de threads (*Blocks*)
- Grade de blocos (*Grid*)

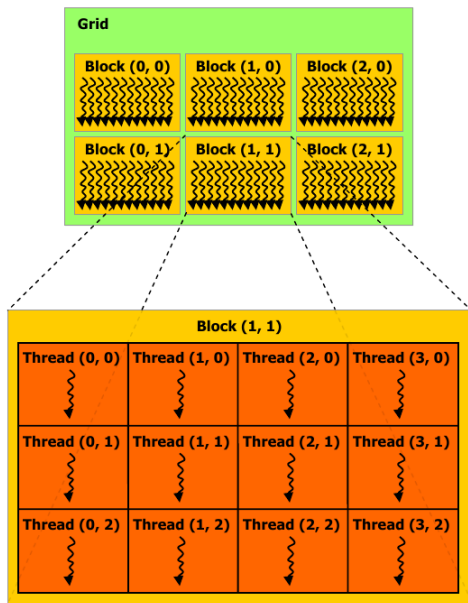
# Organização das Threads

- As *threads* são distribuídas em blocos;
- Os blocos são distribuídos em uma grade.

# Organização das *threads*

## Exemplo 1

- Grade de duas dimensões (**2D-Grid**), contendo blocos de threads com duas dimensões (**2D-Block**)
- Grade:  $3 \times 2$  blocos
- Bloco:  $4 \times 3$  threads
- Total de 72 threads

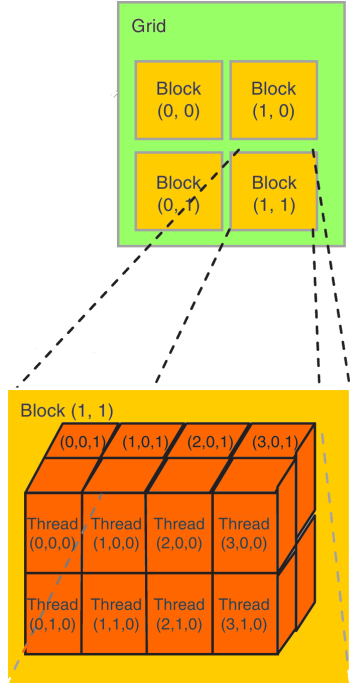




# Organização das *threads*

## Exemplo 2

- Grade de duas dimensões (**2D-Grid**), contendo blocos de threads com três dimensões (**3D-Block**)
- Grade:  $2 \times 2$  blocos
- Bloco:  $4 \times 2 \times 2$  threads
- Total de 64 threads



# Identificação das Threads

## No bloco

- Thread (  $tx$ ,  $ty$ ,  $tz$  );

## Na grade

- Bloco (  $bx$ ,  $by$  );

# Identificação das Threads

## No bloco

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:

## Na grade

- Bloco (  $bx, by$  );
- Palavra-chave CUDA `blockIdx.dim`:

# Identificação das Threads

## No bloco

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:
- (`threadIdx.x, threadIdx.y, threadIdx.z`)

## Na grade

- Bloco (  $bx, by$  );
- Palavra-chave CUDA `blockIdx.dim`:
- (`blockIdx.x, blockIdx.y`);

# Identificação das Threads

## No bloco

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:
- (`threadIdx.x, threadIdx.y, threadIdx.z`)
- Tam. máx. cada *dim*: ( 512, 512, 64 );
- Número máximo de threads: 512

## Na grade

- Bloco (  $bx, by$  );
- Palavra-chave CUDA `blockIdx.dim`:
- (`blockIdx.x, blockIdx.y`);
- Tam. máx. cada *dim*: ( 65535, 65535 );
-

# Identificação das Threads

## No bloco (dados referentes à geração Tesla)

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:
- (`threadIdx.x, threadIdx.y, threadIdx.z`)
- Tam. máx. cada *dim*: ( 512, 512, 64 );
- Número máximo de threads: 512

## Na grade (dados referentes à geração Tesla)

- Bloco (  $bx, by$  );
- Palavra-chave CUDA `blockIdx.dim`:
- (`blockIdx.x, blockIdx.y`);
- Tam. máx. cada *dim*: ( 65535, 65535 );
-

# Identificação das Threads

## No bloco (dados referentes à geração Fermi)

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:
- (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`)
- Tam. máx. cada *dim*: ( 1024, 1024, 64 );
- Número máximo de threads: 1024

## Na grade (dados referentes à geração Fermi)

- Bloco (  $bx, by, bz$  );
- Palavra-chave CUDA `blockIdx.dim`:
- (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`);
- Tam. máx. cada *dim*: ( 65535, 65535, 65535 );
-

# Identificação das Threads

## No bloco (dados referentes à geração Kepler)

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:
- (`threadIdx.x, threadIdx.y, threadIdx.z`);
- Tam. máx. cada *dim*: ( 1024, 1024, 64 );
- Número máximo de threads: 1024

## Na grade (dados referentes à geração Kepler)

- Bloco (  $bx, by, bz$  );
- Palavra-chave CUDA `blockIdx.dim`:
- (`blockIdx.x, blockIdx.y, blockIdx.z`);
- Tam. máx. cada *dim*: (  $2^{31}-1, 2^{31}-1, 2^{31}-1$  );
-



# Identificação das Threads

## No bloco (dados referentes à geração Maxwell)

- Thread (  $tx, ty, tz$  );
- Palavra-chave CUDA `threadIdx.dim`:
- (`threadIdx.x, threadIdx.y, threadIdx.z`);
- Tam. máx. cada *dim*: ( 1024, 1024, 64 );
- Número máximo de threads: 1024

## Na grade (dados referentes à geração Maxwell)

- Bloco (  $bx, by, bz$  );
- Palavra-chave CUDA `blockIdx.dim`:
- (`blockIdx.x, blockIdx.y, blockIdx.z`);
- Tam. máx. cada *dim*: (  $2^{31}-1, 2^{31}-1, 2^{31}-1$  );
-

# Recursos da GPU

## Capacidade de Computação (*Compute Capability*)

- Geração Tesla (1.x): 1.0, 1.1, 1.2, 1.3
- Geração Fermi (2.x): 2.0, 2.1
- Geração Kepler (3.x): 3.0, 3.2, 3.5, 3.7
- Geração Maxwell (5.x): 5.0, 5.2
- Geração Pascal (6.x): 6.0, 6.1
- Geração Volta (7.x): 7.0
- Geração Turing (7.x): 7.5

# Recursos da GPU

Versão	Placa (GPU)
1.0	GeForce 8800 GTX (G80)
1.1	GeForce 8400GS/GT (G84)
1.2	GeForce 210 (GT215)
1.3	GeForce GTX 285 (GT200b)
2.0	GeForce GTX 480 (GF100), GeForce GTX 580 (GF110)
2.1	GeForce GTX 460 (GF100)
3.0	Tesla K10 (GK104), GeForce GTX 680 (GK104)
3.5	Tesla K20, Tesla K40 (GK110), GeForce GTX TITAN (GK110)
3.7	Tesla K80
5.0	GeForce GTX 750, GeForce GTX 960M, Quadro K2200
5.2	GeForce GTX TITAN X, Quadro M6000
5.3	Tegra Jetson TX1

Ver lista completa em:

<http://developer.nvidia.com/cuda-gpus>

e

<https://en.wikipedia.org/wiki/CUDA>

# Recursos da GPU

Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade	65535						$2^{31}-1$	
Número de dimensões do bloco de threads	3							
Tam. máx. das dimensões x e y no bloco	512				1024			
Tam. máx. da dimensão z no bloco	64							
Núm. máx. threads no bloco	512				1024			

# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads:

$i = \text{threadIdx.x}$



# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads:

$i = \text{threadIdx}.x$



N blocos com 1 thread:

$i = \text{blockIdx}.x$



# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads:

$i = \text{threadIdx.x}$



N blocos com 1 thread:

$i = \text{blockIdx.x}$



- **blockIdx.x**: palavra-chave CUDA que referencia os índices dos blocos;

# Soma de vetores: 1 bloco com $N$ threads

## add\_gpu.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {  
    int tid = threadIdx.x;    // handles the data at its thread id  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

## add\_gpu.cu

```
add<<<1,N>>>( N, dev_a, dev_b, dev_c );
```



# Soma de vetores: $N$ blocos com 1 *thread*

## add\_gpu\_blocks.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {  
    int tid = blockIdx.x;    // handles the data at its thread id  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

## add\_gpu\_blocks.cu

```
add<<<N,1>>>( N, dev_a, dev_b, dev_c );
```

# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads

$i = \text{threadIdx}.x$



N blocos com 1 thread

$i = \text{blockIdx}.x$



# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads

$i = \text{threadIdx}.x$



N blocos com 1 thread

$i = \text{blockIdx}.x$



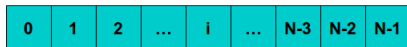
Na soma de vetores, qual a “melhor” estratégia?

# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads

$i = \text{threadIdx}.x$



N blocos com 1 thread

$i = \text{blockIdx}.x$



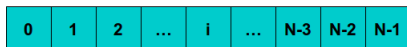
Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade								
Núm. máx. threads no bloco								

# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads: **limitado número de elementos.**

$i = \text{threadIdx}.x$



N blocos com 1 thread

$i = \text{blockIdx}.x$



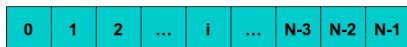
Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade								
Núm. máx. threads no bloco	512				1024			

# Organização e Identificação das *threads*

## Exemplo com vetores

1 bloco com N threads: **limitado número de elementos.**

$i = \text{threadIdx}.x$



N blocos com 1 thread: **maior número de elementos.**

$i = \text{blockIdx}.x$



Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade	65535						$2^{31}-1$	
Núm. máx. threads no bloco	512				1024			



# Soma de vetores: GPU × CPU

Comparação de desempenho: N=65535

- Na CPU: **246** microssegundos;
- Na GPU: **691** microssegundos;
- Na GPU o tempo foi quase **3 vezes maior**;





# Soma de vetores na GPU

## Comparação de desempenho: $N=512$

- 1 bloco com  $N=512$  threads: **1.69** microssegundos;

# Soma de vetores na GPU

## Comparação de desempenho: $N=512$

- 1 bloco com  $N=512$  threads: **1.69** microssegundos;
- $N=512$  blocos com 1 thread: **6.99** microssegundos;

# Soma de vetores na GPU

## Comparação de desempenho: $N=512$

- 1 bloco com  $N=512$  threads: **1.69** microssegundos;
  - $N=512$  blocos com 1 thread: **6.99** microssegundos;
- Cerca de **4** vezes **mais lenta** a execução;

# Soma de vetores na GPU

## Comparação de desempenho: $N=512$

- 1 bloco com  $N=512$  threads: **1.69** microssegundos;
  - $N=512$  blocos com 1 thread: **6.99** microssegundos;
- Cerca de **4** vezes **mais lenta** a execução;
- Tanto este caso, quanto aquele com  $N=65535$  blocos, veremos que se trata de uma questão de melhorar a **atribuição** das *threads*.

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 **Modelo de Paralelismo**
  - Módulo 2: Hierarquia, Organização e Identificação
  - **Módulo 3: Atribuição**
  - Módulo 4: Escalonamento
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais

# Atribuição das Threads

- 1 core CUDA é 1 *Streaming Processor* (SP)



# Atribuição das Threads

- 1 core CUDA é 1 *Streaming Processor* (SP)
- 1 *Streaming Multiprocessor* (SM) é constituído por um número de SP





# Atribuição das Threads

- 1 core CUDA é 1 *Streaming Processor* (SP)
- 1 *Streaming Multiprocessor* (SM) é constituído por um número de SP
- Número de SP por SM depende da *Compute Capability* da GPU



# Atribuição das Threads

- GPU 1.x possui 8 SP por SM



# Atribuição das Threads

- GPU 1.x possui 8 SP por SM
- GPU 2.0 possui 32 SP por SM



# Atribuição das Threads

- GPU 1.x possui 8 SP por SM
- GPU 2.0 possui 32 SP por SM
- GPU 2.1 possui 48 SP por SM



# Atribuição das Threads

- GPU 1.x possui 8 SP por SM
- GPU 2.0 possui 32 SP por SM
- GPU 2.1 possui 48 SP por SM
- GPU 3.x possui 192 SP por SM



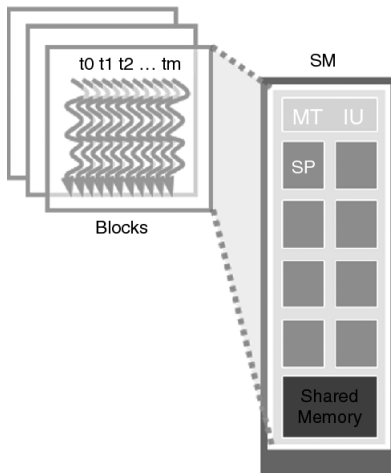
# Atribuição das Threads

- GPU 1.x possui 8 SP por SM
- GPU 2.0 possui 32 SP por SM
- GPU 2.1 possui 48 SP por SM
- GPU 3.x possui 192 SP por SM
- GPU 5.x possui 128 SP por SM



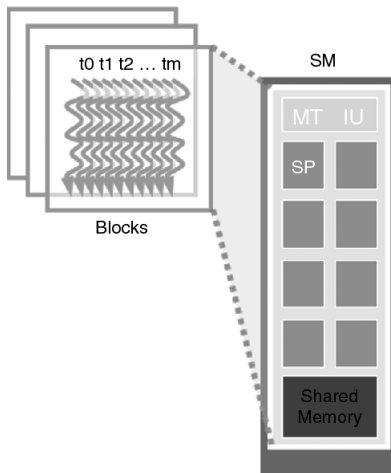
# Atribuição das Threads

- As threads são atribuídas aos SMs em blocos



# Atribuição das Threads

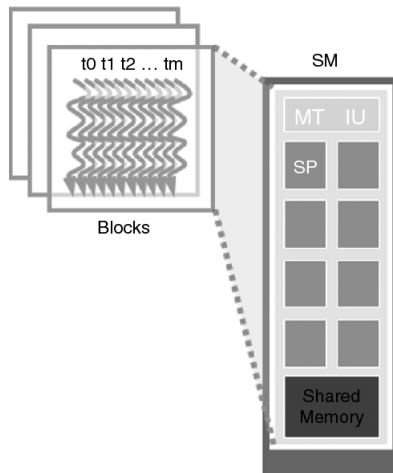
- As threads são atribuídas aos SMs em blocos
- Máximo de 8 blocos de *threads* podem ser atribuídos no SM





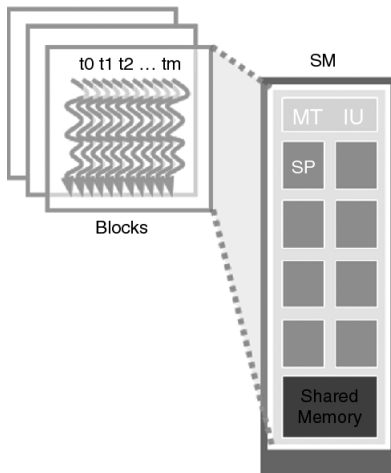
# Atribuição das Threads

- As threads são atribuídas aos SMs em blocos
- Máximo de 8 blocos de *threads* podem ser atribuídos no SM
- **OU** até 768, 1024 e 1536 threads para GPUs com *Compute Capability* 1.0-1.1, 1.2-1.3 e 2.x, respectivamente



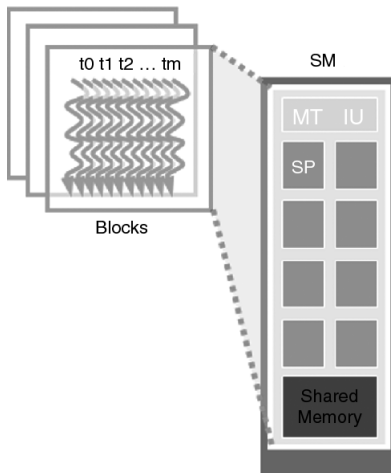
# Atribuição das Threads

- *Compute Capability 3.x:*



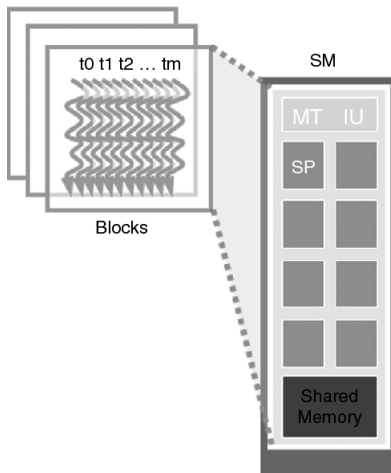
# Atribuição das Threads

- *Compute Capability 3.x*:
- Máximo de 16 blocos de *threads* podem ser atribuídos no SM



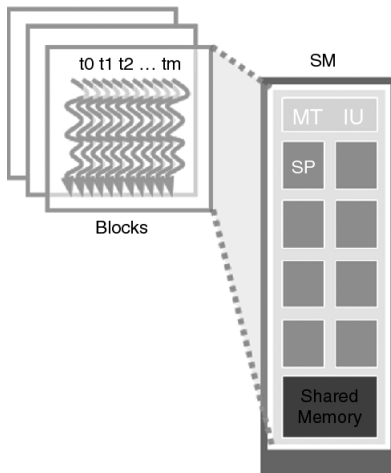
# Atribuição das Threads

- *Compute Capability 3.x*:
- Máximo de 16 blocos de *threads* podem ser atribuídos no SM
- **OU** até 2048 threads



# Atribuição das Threads

- *Compute Capability 5.x*:
- Máximo de **32** blocos de *threads* podem ser atribuídos no SM
- **OU** até 2048 threads



# Atribuição das Threads

## Exemplo: Compute Capability 1.0

- Máximo de threads por SM: 768

# Atribuição das Threads

## Exemplo: Compute Capability 1.0

- Máximo de threads por SM: 768
- 3 blocos de 256 threads, 8 blocos de 96 threads, etc...

# Atribuição das Threads

## Exemplo: Compute Capability 1.0

- Máximo de threads por SM: 768
- 3 blocos de 256 threads, 8 blocos de 96 threads, etc...
- Possui 16 SM: até 12.288 threads atribuídas para serem executadas



# Atribuição das Threads

## Exemplo: Compute Capability 1.3

- Máximo de threads por SM: 1024

# Atribuição das Threads

## Exemplo: Compute Capability 1.3

- Máximo de threads por SM: 1024
- 4 blocos de 256 threads, 8 blocos de 128 threads, etc...

# Atribuição das Threads

## Exemplo: Compute Capability 1.3

- Máximo de threads por SM: 1024
- 4 blocos de 256 threads, 8 blocos de 128 threads, etc...
- 16 blocos de 64 threads: não é possível, pela limitação de 8 blocos por SM

# Atribuição das Threads

## Exemplo: Compute Capability 1.3

- Máximo de threads por SM: 1024
- 4 blocos de 256 threads, 8 blocos de 128 threads, etc...
- 16 blocos de 64 threads: não é possível, pela limitação de 8 blocos por SM
- Possui 30 SM: até 30.720 threads atribuídas para serem executadas

# Soma de vetores na GPU

## Comparação de desempenho

- 1 bloco  $N=512$  threads: **1.69** microssegundos;
- $N=512$  blocos com 1 thread: **6.99** microssegundos;  
Cerca de **4** vezes **mais lenta** a execução;
- Este é um problema de **atribuição** das *threads*.

# Soma de vetores na GPU

## Comparação de desempenho

- 1 bloco  $N=512$  threads: **1.69** microssegundos;
  - $N=512$  blocos com 1 thread: **6.99** microssegundos;
- Cerca de **4** vezes **mais lenta** a execução;
- Este é um problema de **atribuição** das *threads*.

$N$  blocos com 1 *thread*: poucas *threads* residentes no SM.

$i = \text{blockIdx.x}$



# Soma de vetores na GPU

## Comparação de desempenho

- 1 bloco  $N=512$  threads: **1.69** microssegundos;
  - $N=512$  blocos com 1 thread: **6.99** microssegundos;
- Cerca de **4** vezes **mais lenta** a execução;
- Este é um problema de **atribuição** das *threads*.

$N$  blocos com 1 *thread*: poucas *threads* residentes no SM.

$i = \text{blockIdx.x}$



1 bloco com  $N$  *threads*: mais *threads* residentes no SM, que permite um maior **escalonamento** da execução paralela.

$i = \text{threadIdx.x}$



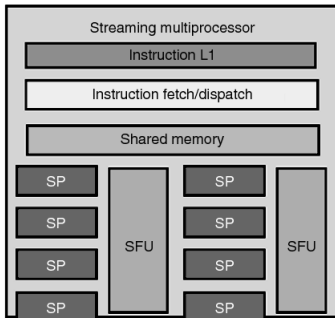
# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 **Modelo de Paralelismo**
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - **Módulo 4: Escalonamento**
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais



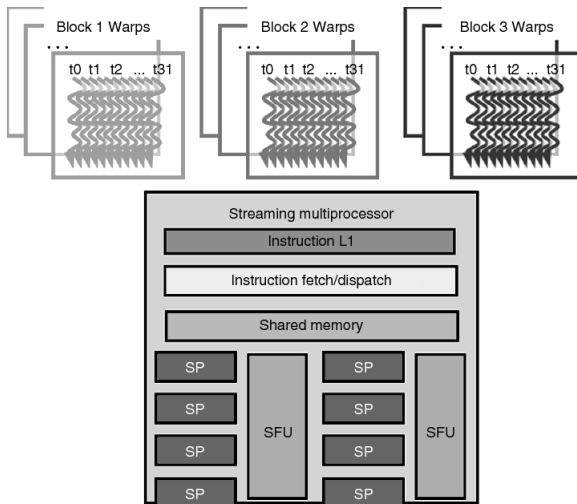
# Escalonamento das Threads

- As threads de um bloco atribuído a um SM, são escalonadas em grupos de 32 threads, denominados *warps*;



# Escalonamento das Threads

- As threads de um bloco atribuído a um SM, são escalonadas em grupos de 32 threads, denominados *warps*;



# Escalonamento das Threads

## Exemplo

- blocos de 256 threads atribuídos ao SM

# Escalonamento das Threads

## Exemplo

- blocos de 256 threads atribuídos ao SM
- $256/32 = 8$  warps por bloco

# Escalonamento das Threads

## Exemplo

- blocos de 256 threads atribuídos ao SM
- $256/32 = 8$  warps por bloco
- **Compute Capability 1.0**: no máximo podem residir no SM 3 blocos de 256 threads, ou 24 warps = 768 threads

# Escalonamento das Threads

## Exemplo

- blocos de 256 threads atribuídos ao SM
- $256/32 = 8$  warps por bloco
- **Compute Capability 1.0**: no máximo podem residir no SM 3 blocos de 256 threads, ou 24 warps = 768 threads
- **Compute Capability 1.3**: no máximo podem residir no SM 4 blocos de 256 threads, ou 32 warps = 1024 threads

# Escalonamento das Threads

- Acessos a memória global: latência de 200 a 800 ciclos.

# Escalonamento das Threads

- Acessos a memória global: latência de 200 a 800 ciclos.
- O número de warps residentes pode ser maior do que o número de SP no SM;

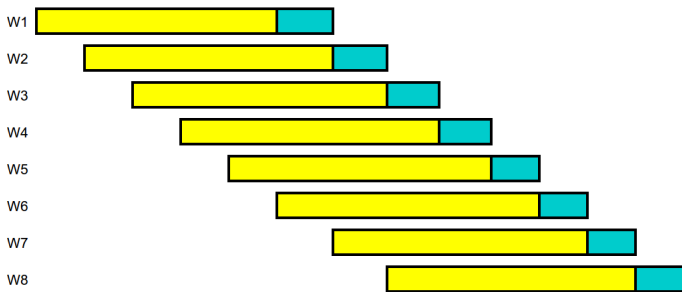


# Escalonamento das Threads

- Acessos a memória global: latência de 200 a 800 ciclos.
- O número de warps residentes pode ser maior do que o número de SP no SM;
- Permite que se obtenha uma *ocultação de operações de longa latência*;

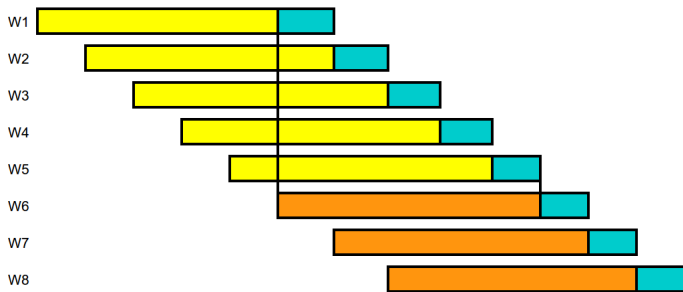
# Escalonamento das Threads

- Acessos a memória global: latência de 200 a 800 ciclos.
- O número de warps residentes pode ser maior do que o número de SP no SM;
- Permite que se obtenha uma *ocultação de operações de longa latência*;



# Escalonamento das Threads

- Acessos a memória global: latência de 200 a 800 ciclos.
- O número de warps residentes pode ser maior do que o número de SP no SM;
- Permite que se obtenha uma *ocultação de operações de longa latência*;



# Escalonamento das Threads

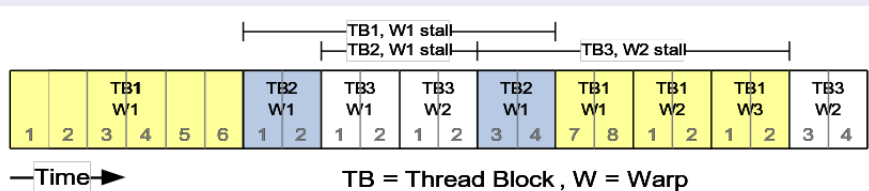
## Comentários

- Nas figuras do exemplo hipotético dado, a razão entre ciclos de acesso e ciclos de operações é 5.
- A partir do sexto warp, há **custo zero** na latência para escalonamento;
- Em um outro caso, por exemplo, para 200 ciclos de um acesso, pode haver 4 instruções com custo de 4 ciclos cada (16 ciclos).
- Neste caso, a razão entre ciclos de acesso e ciclos de instruções é, arredondando para cima, 13.
- São então necessários 14 warps para ocultar a latência de leitura do dado.
- Quanto menor a razão entre ciclos de acesso e ciclos de instruções, menor será o número de warps necessário para ocultar a latência;

# Escalonamento das Threads

## Comentários

- Além da latência de acesso, o grande número de warps pode ocultar o custo da própria dependência de instruções em um código:



## Definição

Ocupação do SM, ou *Occupancy*: é definida como a razão entre o número de *warps* ativos pelo número máximo de *warps* suportado no SM.

# Recursos da GPU: revisão

Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade	65535						$2^{31}-1$	
Número de dimensões do bloco de threads	3							
Tam. máx. das dimensões x e y no bloco	512				1024			
Tam. máx. da dimensão z no bloco	64							
Núm. máx. threads no bloco	512				1024			

# Recursos da GPU: atualização

Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade	65535						$2^{31}-1$	
Número de dimensões do bloco de threads	3							
Tam. máx. das dimensões x e y no bloco	512				1024			
Tam. máx. da dimensão z no bloco	64							
Núm. máx. threads no bloco	512				1024			
Tamanho do warp	32							
Núm. máx. de blocos por SM	8						16	
Núm. máx. warps por SM	24		32		48		64	
Núm. máx. threads por SM	768		1024		1536		2048	



# Escalonamento das Threads

Qual o tamanho adequado do bloco?

Compute Capability	Num. máx. threads	threads no SM			
		64	256	512	1024
1.0-1.1	768	512(8)	768(3)	512(1)	0(0)
1.2-1.3	1024	512(8)	1024(4)	1024(2)	0(0)
2.0-2.1	1536	512(8)	1536(6)	1536(3)	1024(1)
3.0-3.5	2048	1024(16)	2048(8)	2048(4)	2048(2)

# Escalonamento das Threads

Qual o tamanho adequado do bloco?

Compute Capability	Num. máx. threads	threads no SM			
		64	256	512	1024
1.0-1.1	768	512(8)	768(3)	512(1)	0(0)
1.2-1.3	1024	512(8)	1024(4)	1024(2)	0(0)
2.0-2.1	1536	512(8)	1536(6)	1536(3)	1024(1)
3.0-3.5	2048	1024(16)	2048(8)	2048(4)	2048(2)

Compute Capability	Num. máx. warps	warps no SM			
		64	256	512	1024
1.0-1.1	24	16	24	16	0
1.2-1.3	32	16	32	32	0
2.0-2.1	48	16	48	48	32
3.0-3.5	64	32	64	64	64

# Escalonamento das Threads

Qual o tamanho adequado do bloco?

Compute Capability	Num. máx. warps	warps no SM			
		64	256	512	1024
1.0-1.1	24	16	24	16	0
1.2-1.3	32	16	32	32	0
2.0-2.1	48	16	48	48	32
3.0-3.5	64	32	64	64	64

# Escalonamento das Threads

Qual o tamanho adequado do bloco?

Compute Capability	Num. máx. warps	warps no SM			
		64	256	512	1024
1.0-1.1	24	16	24	16	0
1.2-1.3	32	16	32	32	0
2.0-2.1	48	16	48	48	32
3.0-3.5	64	32	64	64	64

Compute Capability	Num. máx. warps	Occupancy			
		64	256	512	1024
1.0-1.1	24	66%	100%	66%	0%
1.2-1.3	32	50%	100%	100%	0%
2.0-2.1	48	33%	100%	100%	66%
3.0-3.5	64	50%	100%	100%	100%

# Occupancy

Ver planilha *CUDA Occupancy Calculator*:  
<https://tinyurl.com/y6thnsys>

# Soma de vetores na GPU

## Estratégia paralela

- Uma nova estratégia a empregar;

# Soma de vetores na GPU

## Estratégia paralela

- Uma nova estratégia a empregar;
- Combina a capacidade de escalonamento de `threadIdx.x`, com a maior capacidade de identificação de elementos de `blockIdx.x`;

# Soma de vetores na GPU

## Estratégia paralela

- Uma nova estratégia a empregar;
- Combina a capacidade de escalonamento de `threadIdx.x`, com a maior capacidade de identificação de elementos de `blockIdx.x`;
- `tid = blockIdx.x * blockDim.x + threadIdx.x`



# Soma de vetores na GPU

## Estratégia paralela

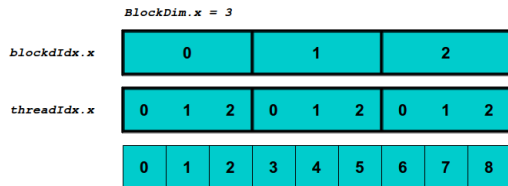
- Uma nova estratégia a empregar;
- Combina a capacidade de escalonamento de `threadIdx.x`, com a maior capacidade de identificação de elementos de `blockIdx.x`;
- $\text{tid} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

`threadIdx.x`: índices das *threads* no bloco;

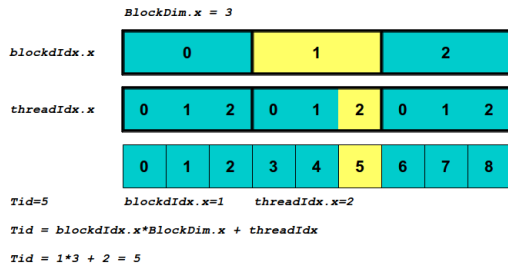
`blockIdx.x`: índices dos blocos na grade;

`blockDim.x`: tamanho dos blocos (*threads* no bloco);

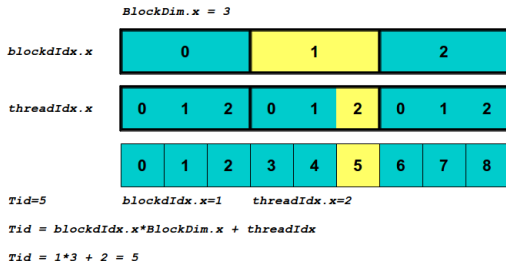
# Soma de vetores na GPU



# Soma de vetores na GPU



# Soma de vetores na GPU



## add\_gpu\_grid.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {  
    int tid = blockIdx.x*blockDim.x + threadIdx.x;    // this thread handles the data at its  
        thread id  
    if (tid < N)  
        c[tid] = a[tid] + b[tid];  
}
```

## add\_gpu\_grid.cu

```
add<<< (N + blocksize-1)/blocksize, blocksize >>>( N, dev_a, dev_b, dev_c );
```

# Soma de vetores na GPU

128 blocos com 512 threads:  $N=65536$

```
$ nvcc add_gpu_grid.cu -o add_gpu_grid
$ srun -p treinamento_gpu nvprof ./add_gpu_grid `echo 128*512 | bc` 512 1
```

```
%
===== NVPROF is profiling add_gpu_grid...
===== Command: add_gpu_grid 65536 512 1
```

Device: Tesla C2050

===== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
52.56	127.42us	1	127.42us	127.42us	127.42us	[CUDA memcpy DtoH]
43.67	105.88us	2	52.94us	52.57us	53.31us	[CUDA memcpy HtoD]
3.77	9.14us	1	9.14us	9.14us	9.14us	add(int, float*, float*, float*)

```
%
```



# Soma de vetores na GPU

$N=65535$  blocos com 1 *thread*

```
$ srun -p treinamento_gpu nvprof ./add_gpu_blocks 65535
```

```
=====  
===== NVPROF is profiling add_gpu_blocks...  
===== Command: add_gpu_blocks 65535 1
```

```
Device: Tesla C2050
```

```
===== Profiling result:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
74.70	691.15us	1	691.15us	691.15us	691.15us	add(int, float*, float*, float*)
13.85	128.12us	1	128.12us	128.12us	128.12us	[CUDA memcpy DtoH]
11.45	105.95us	2	52.97us	52.35us	53.60us	[CUDA memcpy HtoD]

## Na CPU

```
$ srun -p treinamento_gpu ./add_cpu 65535
```

```
Execution Time (microseconds):          246.00
```

# Soma de vetores

Comparação de desempenho GPU  $\times$  CPU

Comparação de desempenho GPU  $\times$  GPU



# Soma de vetores

## Comparação de desempenho GPU $\times$ CPU

- Na CPU: **246** microssegundos;

## Comparação de desempenho GPU $\times$ GPU

# Soma de vetores

## Comparação de desempenho GPU × CPU

- Na CPU: **246** microssegundos;
- Na GPU: **9.14** microssegundos:  
Cerca de **27** vezes **mais rápida** a execução;

## Comparação de desempenho GPU × GPU

# Soma de vetores

## Comparação de desempenho GPU × CPU

- Na CPU: **246** microssegundos;
- Na GPU: **9.14** microssegundos:  
Cerca de **27** vezes **mais rápida** a execução;

## Comparação de desempenho GPU × GPU

- N=65535 blocos de 1 *thread*: **691** microssegundos;

# Soma de vetores

## Comparação de desempenho GPU × CPU

- Na CPU: **246** microssegundos;
- Na GPU: **9.14** microssegundos:  
Cerca de **27** vezes **mais rápida** a execução;

## Comparação de desempenho GPU × GPU

- N=65535 blocos de 1 *thread*: **691** microssegundos;
- 128 blocos com 512 *threads*: **9.14** microssegundos:  
Cerca de **75** vezes **mais rápida** a execução;

# Soma de vetores maiores na GPU

65535 blocos com 512 threads:  $N=33.553.920$

```
$ nvcc add_gpu_grid.cu -o add_gpu_grid
$ srun -p treinamento_gpu nvprof ./add_gpu_grid `echo 65535*512 | bc` 512 1

%
===== NVPROF is profiling add_gpu_grid...
===== Command: add_gpu_grid 33.553.920 512 1

Device: Tesla C2050
===== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
49.24      62.21ms      1      62.21ms  62.21ms  62.21ms  [CUDA memcpy DtoH]
47.36      59.84ms      2      29.92ms  29.92ms  29.92ms  [CUDA memcpy HtoD]
 3.40      4.30ms      1      4.30ms   4.30ms   4.30ms  add(int, float*, float*, float*)
```

- No limite da dimensão máxima da grade:

$$\text{gridDim.x} \leq 2^{16} - 1 = 65535$$

- Uma nova estratégia consiste percorrer com um laço todos os elementos do vetor.

## Soma de vetores maiores na GPU (cont.)

Neste caso, a cada iteração de um laço, o índice `tid` é incrementado pelos índices dos blocos de uma grade de menor dimensão.

# Soma de vetores maiores na GPU usando *loop*

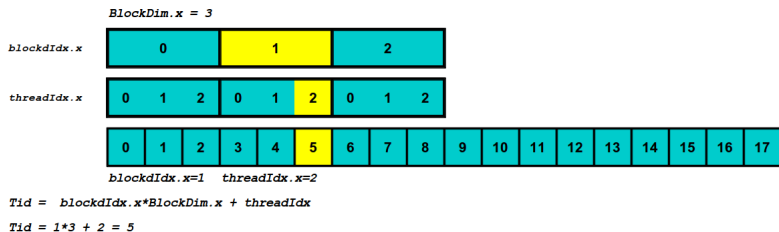
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

# Soma de vetores maiores na GPU usando *loop*



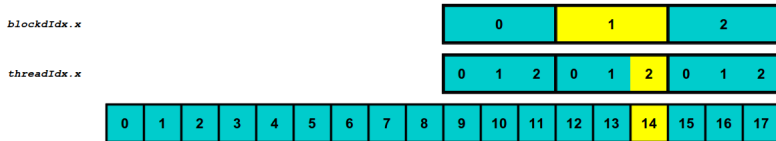


# Soma de vetores maiores na GPU usando *loop*



# Soma de vetores maiores na GPU usando *loop*

`BlockDim.x = 3 GridDim.x = 3`

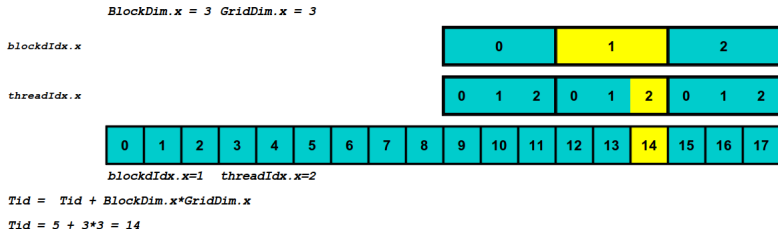


`blockIdx.x=1 threadIdx.x=2`

`Tid = Tid + BlockDim.x*GridDim.x`

`Tid = 5 + 3*3 = 14`

# Soma de vetores maiores na GPU usando *loop*



## add\_gpu\_grid\_loop.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {  
    int tid = blockIdx.x*blockDim.x + threadIdx.x;    // handles the data at its thread id  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += blockDim.x * gridDim.x;  
    }  
}
```

# Soma de vetores maiores na GPU usando *loop* (cont.)

## add\_gpu\_grid\_loop.cu

```
add<<< GridSize, BlockSize >>>( N, dev_a, dev_b, dev_c );
```

- **blockDim.x**: tamanho dos blocos (*threads* no bloco);
- **gridDim.x**: tamanho da grade (blocos na grade);

# Soma de vetores maiores na GPU

64 × grade de 1024 blocos com 512 threads:  $N=33.554.432$

```
$ nvcc add_gpu_grid_loop.cu -o add_gpu_grid_loop
$ srun -p treinamento_gpu nvprof ./add_gpu_grid_loop `echo 64*1024*512 | bc` 1024 512 1

%
===== NVPROF is profiling add_gpu_grid_loop...
===== Command: add_gpu_grid_loop 33554432 1024 512 1
Device: Tesla C2050
===== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
50.39      64.72ms      1      64.72ms  64.72ms  64.72ms  [CUDA memcpy DtoH]
46.01      59.09ms      2      29.55ms  29.52ms  29.57ms  [CUDA memcpy HtoD]
 3.60      4.62ms      1      4.62ms   4.62ms   4.62ms  add(int, float*, float*, float*)

%
```

$\text{gridDim.x} * \text{blockDim.x} = 1024 * 512$ : 64 passos no laço

$\text{gridDim.x} * \text{blockDim.x} = 2048 * 512$ : 32 passos no laço

$\text{gridDim.x} * \text{blockDim.x} = 4096 * 512$ : 16 passos no laço

# Soma de vetores na CPU

$$N = 64 \times 1024 \times 512 = 33.554.432$$

```
$ srun -p treinamento_gpu ./add_cpu `echo 64*1024*512 | bc`
```

```
Execution Time (microseconds): 83836.00
```

# Soma de vetores: GPU × CPU

## Comparação de desempenho

- Na CPU: **83 836** microssegundos;
- Na GPU: **4 620** microssegundos;
- Na GPU cerca de **18** vezes **mais rápida** a execução.

## Soma de vetores: GPU $\times$ CPU - Observação

### Comparação de desempenho com transferência

- Na CPU: **83 836** microssegundos;
- Na GPU: **4 620 + 123 810 = 128 430** microssegundos;
- Na GPU cerca de **50% mais lenta** a execução.



# Soma de matrizes

```
$ cd matSum
```

# Soma de matrizes na CPU

## matSum\_cpu.cu

```
void matSum(float* S, float* A, float* B, unsigned int N) {
    int i, j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            int tid = i*N + j;
            S[tid] = A[tid] + B[tid];
        }
    }
}
```

## matSum\_cpu.cu

```
matSum( S, A, B, N );
```

# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

### matSum\_gpu.cu

```
__global__ void matSum(float* S, float* A, float* B, int N) {  
    int i = blockIdx.y*blockDim.y + threadIdx.y;  
    int j = blockIdx.x*blockDim.x + threadIdx.x;  
    int tid = i*N + j;  
    if (tid < N*N) {  
        S[tid] = A[tid] + B[tid];  
    }  
}
```

# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

### matSum\_gpu.cu

```
__global__ void matSum(float* S, float* A, float* B, int N) {  
    int i = blockIdx.y*blockDim.y + threadIdx.y;  
    int j = blockIdx.x*blockDim.x + threadIdx.x;  
    int tid = i*N + j;  
    if (tid < N*N) {  
        S[tid] = A[tid] + B[tid];  
    }  
}
```

**threadIdx.y**: índices verticais das *threads* no bloco 2D;

**blockIdx.y**: índices verticais dos blocos na grade 2D;

**blockDim.y**: tamanho vertical dos blocos 2D;

# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

### matSum\_gpu.cu

```
__global__ void matSum(float* S, float* A, float* B, int N) {  
    int i = blockIdx.y*blockDim.y + threadIdx.y;  
    int j = blockIdx.x*blockDim.x + threadIdx.x;  
    int tid = i*N + j;  
    if (tid < N*N) {  
        S[tid] = A[tid] + B[tid];  
    }  
}
```

**threadIdx.y**: índices verticais das *threads* no bloco 2D;

**blockIdx.y**: índices verticais dos blocos na grade 2D;

**blockDim.y**: tamanho vertical dos blocos 2D;

**threadIdx.x**: índices horizontais das *threads* no bloco 2D;

**blockIdx.x**: índices horizontais dos blocos na grade 2D;

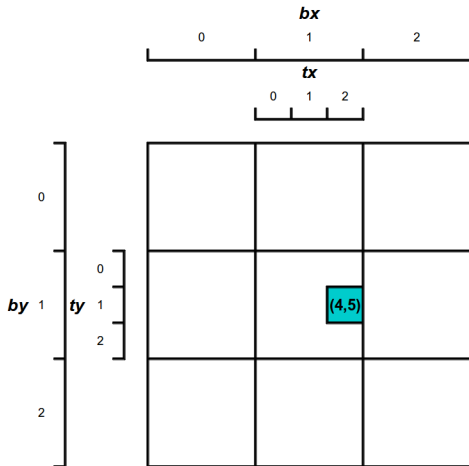
**blockDim.x**: tamanho horizontal dos blocos 2D;

# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

$$i = by*(blocos\ na\ horizontal) + ty$$

$$j = bx*(blocos\ na\ vertical) + tx$$



# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

$$i = by * (\text{blocos na horizontal}) + ty$$

$$i = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y};$$

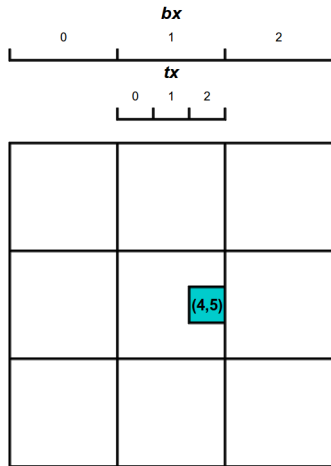
$$j = bx * (\text{blocos na vertical}) + tx$$

$$j = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$

by

ty

2



# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

$$i = by * (\text{blocos na horizontal}) + ty$$

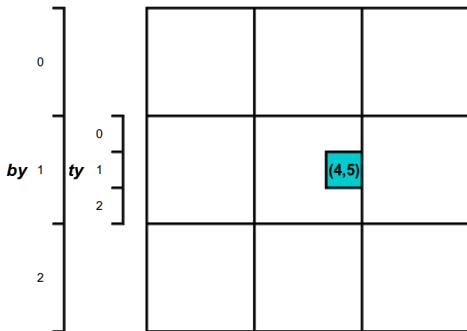
$$i = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y};$$

$$i = 1 * 3 + 1 = 4;$$

$$j = bx * (\text{blocos na vertical}) + tx$$

$$j = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$

$$j = 1 * 3 + 2 = 5;$$





# Soma de matrizes na GPU

## Mapeamento dos índices dos elementos

$$i = by * (\text{blocos na horizontal}) + ty$$

$$i = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y};$$

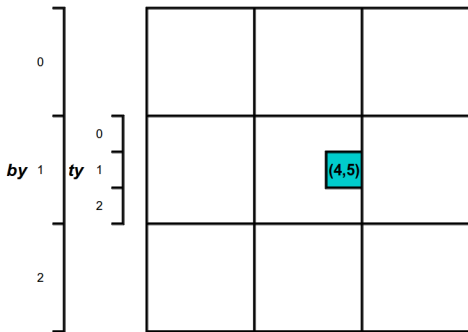
$$i = 1 * 3 + 1 = 4;$$

$$j = bx * (\text{blocos na vertical}) + tx$$

$$j = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$

$$j = 1 * 3 + 2 = 5;$$

$$tid = i * N_x + j$$



# Soma de matrizes na GPU

Chamada ao *kernel* CUDA

## matSum\_gpu.cu

```
int GridSize = (N + BlockSize-1) / BlockSize;
dim3 gridDim(GridSize, GridSize);
dim3 blockDim(BlockSize, BlockSize);

matSum<<< gridDim, blockDim >>>(dev_S, dev_A, dev_B, N);
```

**dim3** declara uma variável como vetor do tipo inteiro, definindo o tamanho das dimensões da grade ou do bloco de *threads*. **dim3** é baseado no tipo **uint3**, sendo cada componente do vetor inicializado com valor 1.

# Soma de matrizes na GPU

## matSum\_gpu.cu

```
__global__ void matSum(float* S, float* A, float* B, int N) {  
    int i = blockIdx.y*blockDim.y + threadIdx.y;  
    int j = blockIdx.x*blockDim.x + threadIdx.x;  
    int tid = i*N + j;  
    if (tid < N*N) {  
        S[tid] = A[tid] + B[tid];  
    }  
}
```

## matSum\_gpu.cu

```
int GridSize = (N + BlockSize-1) / BlockSize;  
dim3 gridDim(GridSize, GridSize);  
dim3 blockDim(BlockSize, BlockSize);  
  
matSum<<< gridDim, blockDim >>>(dev_S, dev_A, dev_B, N);
```

# Soma de matrizes na GPU

## Profiling

```
$ cd matSum
$ nvcc matSum_gpu.cu -o matSum_gpu
$ srun -p treinamento_gpu nvprof ./matSum_gpu 6400 16 0
```

```
%
==== NVPROF is profiling matSum_gpu...
==== Command: matSum_gpu 6400 16 1
Allocate host memory for matrices A and B...
Initialize host matrices...
Allocate device matrices (linearized)...
Execute the kernel...
```

Device: Tesla C2050

==== Profiling result:

Time (%)	Time	Calls	Avg	Min	Max	Name
48.85	79.37ms	1	79.37ms	79.37ms	79.37ms	[CUDA memcpy DtoH]
47.07	76.48ms	2	38.24ms	37.78ms	38.70ms	[CUDA memcpy HtoD]
4.08	6.62ms	1	6.62ms	6.62ms	6.62ms	matSum(float*, float*, float*, int)

```
%
```

# Soma de matrizes na GPU

## Profiling mais detalhado

```
$ srun -p treinamento_gpu nvprof --print-gpu-trace ./matSum_gpu 6400 16 1
```

```
%  
=====  
===== NVPROF is profiling matSum_gpu...  
===== Command: matSum_gpu 6400 16 1  
Allocate host memory for matrices A and B...  
Initialize host matrices...  
Allocate device matrices (linearized)...  
Execute the kernel...
```

```
Device: Tesla C2050
```

```
===== Profiling result:
```

Start	Duration	Grid Size	Block Size	Size	Throughput	Name
1.22s	65.78ms	-	-	163.84MB	2.49GB/s	[CUDA memcpy HtoD]
1.28s	65.78ms	-	-	163.84MB	2.49GB/s	[CUDA memcpy HtoD]
1.35s	6.62ms	(400 400 1)	(16 16 1)	-	-	matSum(float*, float*, f
1.35s	92.94ms	-	-	163.84MB	1.76GB/s	[CUDA memcpy DtoH]

```
%
```

# Soma de matrizes na CPU

```
$ nvcc -O2 matSum_cpu.cu -o matSum_cpu
$ srun -p treinamento_gpu ./matSum_cpu 6400
Allocate memory for matrices A and B...
Initialize matrices...
Sum matrices...

Execution Time (microseconds): 118157.00
```

# Soma de matrizes: GPU × CPU

## Comparação de desempenho

- Na CPU: **118 157** microssegundos;
- Na GPU: **6 620** microssegundos;
- Na GPU cerca de **18** vezes **mais rápida** a execução.

# Soma de matrizes na GPU: Tesla C2050

Compute Capability 2.0

## Variando o tamanho dos blocos

```
$ srun -p treinamento_gpu nvprof ./matSum_gpu 6400 16 0
```

Time(%)	Time	Calls	Avg	Min	Max	Name
4.08	6.62ms	1	6.62ms	6.62ms	6.62ms	matSum(float*, f

```
$ srun -p treinamento_gpu nvprof ./matSum_gpu 6400 32 0
```

Time(%)	Time	Calls	Avg	Min	Max	Name
5.30	8.76ms	1	8.76ms	8.76ms	8.76ms	matSum(float*, f

```
$ srun -p treinamento_gpu nvprof ./matSum_gpu 6400 8 0
```

Time(%)	Time	Calls	Avg	Min	Max	Name
6.97	11.43ms	1	11.43ms	11.43ms	11.43ms	matSum(float*, f



# Soma de matrizes na GPU: Tesla C2050

Compute Capability 2.0

## Variando o tamanho dos blocos

- Admite até 1536 *threads* residentes no SM;

# Soma de matrizes na GPU: Tesla C2050

Compute Capability 2.0

## Variando o tamanho dos blocos

- Admite até 1536 *threads* residentes no SM;
- Bloco  $16 \times 16$ , ou  $1536/256 = 6$  blocos de 256 *threads*, resultando em **100% de ocupância**: **6 620** microssegundos;

# Soma de matrizes na GPU: Tesla C2050

Compute Capability 2.0

## Variando o tamanho dos blocos

- Admite até 1536 *threads* residentes no SM;
- Bloco  $16 \times 16$ , ou  $1536/256 = 6$  blocos de 256 *threads*, resultando em **100% de ocupância**: **6 620** microssegundos;
- Bloco  $32 \times 32$ , ou  $1536/1024 = 1$  bloco de 1024 *threads*, resultando em **66% de ocupância**: **8 760** microssegundos;

# Soma de matrizes na GPU: Tesla C2050

Compute Capability 2.0

## Variando o tamanho dos blocos

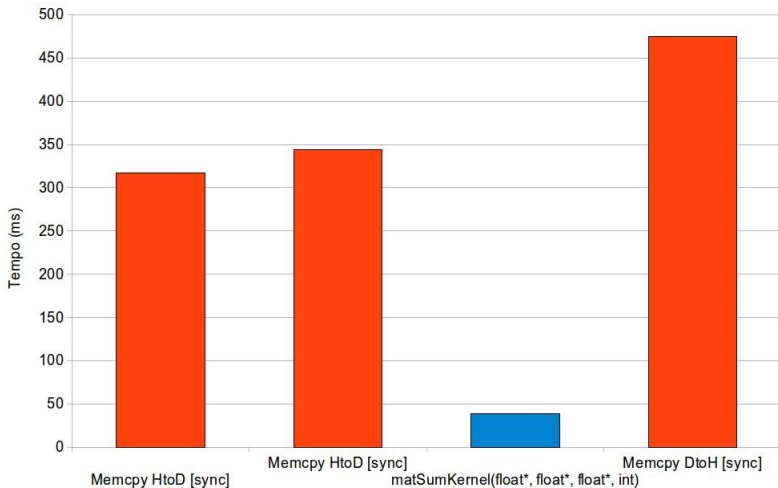
- Admite até 1536 *threads* residentes no SM;
- Bloco  $16 \times 16$ , ou  $1536/256 = 6$  blocos de 256 *threads*, resultando em **100% de ocupância**: **6 620** microssegundos;
- Bloco  $32 \times 32$ , ou  $1536/1024 = 1$  bloco de 1024 *threads*, resultando em **66% de ocupância**: **8 760** microssegundos;
- Bloco  $8 \times 8$ , ou  $1536/64 = 24$  blocos de 64 *threads*. Destes, somente 8 blocos (512 *threads*) podem residir no SM, resultando em **33% de ocupância**: **11 430** microssegundos;

# Soma de matrizes na GPU

## Comentários

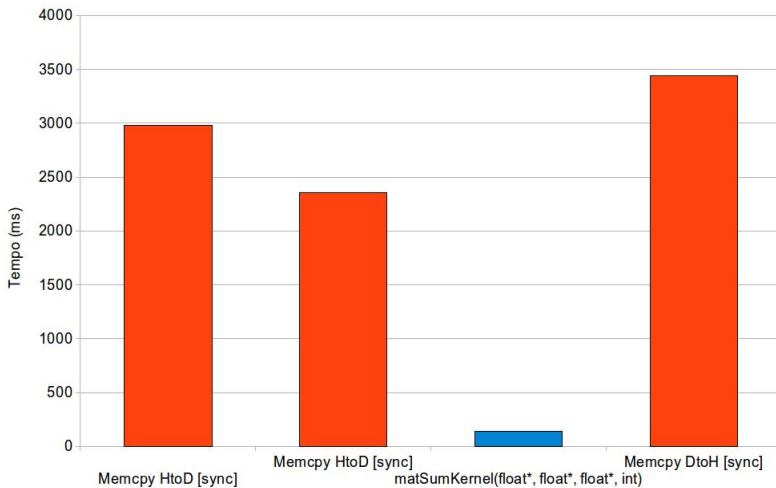
- Na prática a soma de vetores e matrizes em GPU não são bons exemplos de aplicações com potencial ganho de desempenho;
- O aumento do tamanho dos dados cresce na mesma ordem de grandeza que o aumento de processamento;
- No caso das matrizes, para cada  $N^2$  dados transferidos, são realizadas  $N^2$  operações;
- Dado que a transferência dos dados da CPU para GPU é uma operação custosa, o possível ganho de desempenho no processamento, é perdido com o tempo de transferência.

## SOMA DE MATRIZES



# Exemplos N=1280 / GeForce GTX 480

## SOMA DE MATRIZES



## Definição

Ocupação do SM, ou *Occupancy*: é definida como a razão entre o número de *warps* ativos pelo número máximo de *warps* suportado no SM.



# Escalonamento das Threads

## Definição

Ocupação do SM, ou *Occupancy*: é definida como a razão entre o número de *warps* ativos pelo número máximo de *warps* suportado no SM.

## Importante

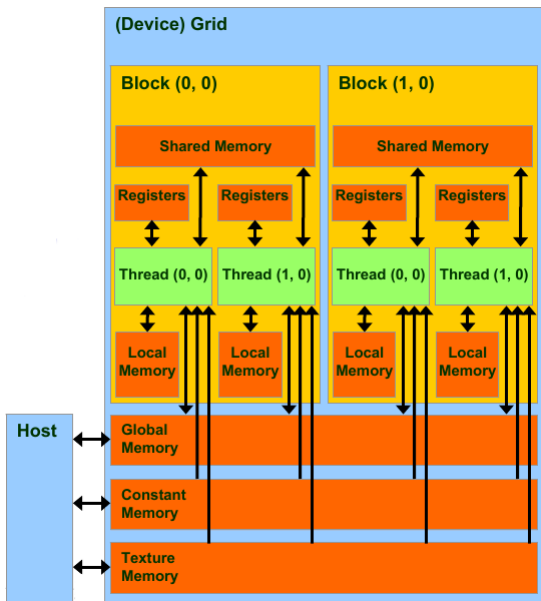
Outro fator que também deve ser considerado para fins de desempenho, é o gerenciamento do uso dos diferentes tipos de memória presentes na GPU, tais como *registradores* e *memória compartilhada*.

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 **Modelo de Paralelismo**
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - **Módulo 5: Hieraquia de Memória**
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais

# Hierarquia de Memória das Threads

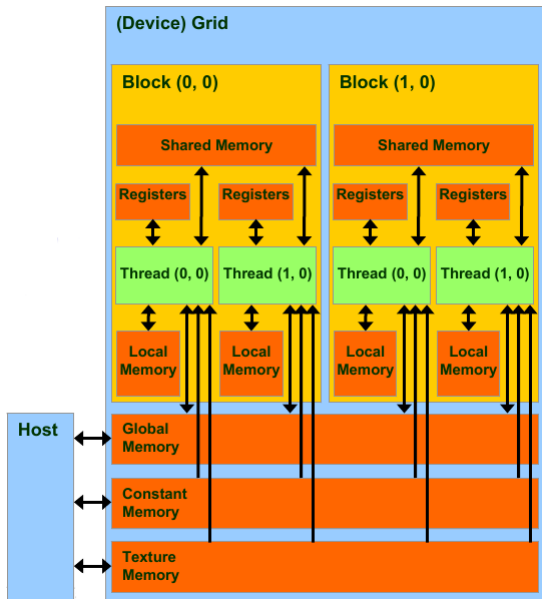
## Hierarquia de Memória



# Hierarquia de Memória das Threads

## Hierarquia de Memória

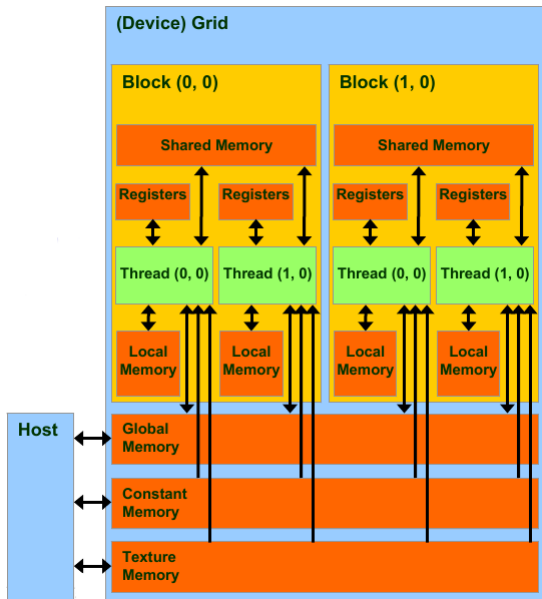
- Grade:
  - ▶ Global
  - ▶ Constante
  - ▶ Textura



# Hierarquia de Memória das Threads

## Hierarquia de Memória

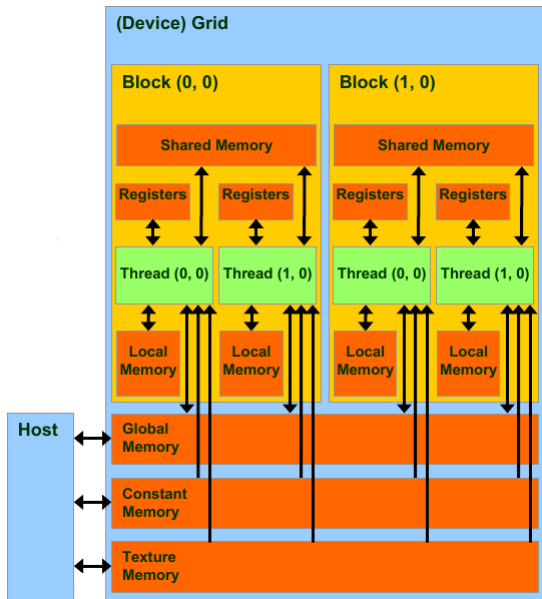
- Grade:
  - ▶ Global
  - ▶ Constante
  - ▶ Textura
- Bloco:
  - ▶ Compartilhada



# Hierarquia de Memória das Threads

## Hierarquia de Memória

- Grade:
  - ▶ Global
  - ▶ Constante
  - ▶ Textura
- Bloco:
  - ▶ Compartilhada
- Threads:
  - ▶ Registradores
  - ▶ Local



# Hierarquia de Memória das Threads

Tipo	Escopo	Acesso	Velocidade	Tempo de Vida	Obs.
Registrador	Thread	R/W	Rápido	Kernel	
Local	Thread	R/W	Lento	Kernel	
Compartilhada	Bloco	R/W	Rápido	Kernel	
Global	Grade	R/W	Lento	Aplicação	<i>Not Cached</i>
Constante	Grade	R/O	Rápido	Aplicação	<i>Cached</i>
Textura	Grade	R/O	Rápido	Aplicação	<i>Spatially Cached</i>

# Observação importante

## ECC Memory

- As GPUs da linha Tesla e Quadro da NVIDIA, possuem suporte a tecnologia de proteção de dados de memória **ECC (Error-correcting code memory)**;
- Este é um tipo de armazenamento de dados que pode detectar e corrigir os tipos mais comuns de corrupção de dados;
- A corrupção de dados não pode ser tolerada, em hipótese alguma, em aplicações tais como computação científica ou financeira;
- A proteção ECC pode ser configurada para estar ativada (on) ou desativada (off) na GPU. Com a ECC ativada, 12.5% da memória global é reservada para os bits da ECC. Por exemplo, uma placa Tesla com 6Gb, efetivamente terá disponível 5.25Gb.

*FONTE:* Wikipedia



# Tipos de Funções

Declaração	Executada no	Chamada pelo
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

# Limite de uso dos registradores

## Compute Capability 1.0

- Cada SM possui 8192 registradores ;

# Limite de uso dos registradores

## Compute Capability 1.0

- Cada SM possui 8192 registradores ;
- Suportar cada SM contendo o número máximo de 768 threads;

# Limite de uso dos registradores

## Compute Capability 1.0

- Cada SM possui 8192 registradores ;
- Suportar cada SM contendo o número máximo de 768 threads;
- Neste caso, cada thread pode utilizar até  $\lfloor 8192/768 \rfloor = 10$  registradores;

# Limite de uso dos registradores

## Compute Capability 1.0

- Cada SM possui 8192 registradores ;
- Suportar cada SM contendo o número máximo de 768 threads;
- Neste caso, cada thread pode utilizar até  $\lfloor 8192/768 \rfloor = 10$  registradores;
- Se cada thread usar 11 registradores, haverá redução em nível de blocos;

# Limite de uso dos registradores

## Compute Capability 1.0

- Cada SM possui 8192 registradores ;
- Suportar cada SM contendo o número máximo de 768 threads;
- Neste caso, cada thread pode utilizar até  $\lfloor 8192/768 \rfloor = 10$  registradores;
- Se cada thread usar 11 registradores, haverá redução em nível de blocos;
- Para 3 blocos de 256 threads usando 11 registradores são atribuídos somente 2 blocos (512 threads);

# Limite de uso dos registradores

## Compute Capability 1.0

- Cada SM possui 8192 registradores ;
- Suportar cada SM contendo o número máximo de 768 threads;
- Neste caso, cada thread pode utilizar até  $\lfloor 8192/768 \rfloor = 10$  registradores;
- Se cada thread usar 11 registradores, haverá redução em nível de blocos;
- Para 3 blocos de 256 threads usando 11 registradores são atribuídos somente 2 blocos (512 threads);
- A ocupação do SM cai de 100% para 67%.

# Limite de uso da memória compartilhada

## Compute Capability 1.0

- Cada SM possui 16 Kbytes de memória compartilhada;



# Limite de uso da memória compartilhada

## Compute Capability 1.0

- Cada SM possui 16 Kbytes de memória compartilhada;
- Cada SM suporta até 8 blocos;

# Limite de uso da memória compartilhada

## Compute Capability 1.0

- Cada SM possui 16 Kbytes de memória compartilhada;
- Cada SM suporta até 8 blocos;
- Cada bloco pode utilizar até  $16\text{Kbytes}/8=2\text{Kbytes}$  de memória compartilhada;

# Recursos da GPU: revisão

Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade	65535						$2^{31}-1$	
Número de dimensões do bloco de threads	3							
Tam. máx. das dimensões x e y no bloco	512				1024			
Tam. máx. da dimensão z no bloco	64							
Núm. máx. threads no bloco	512				1024			
Tamanho do warp	32							
Núm. máx. de blocos por SM	8						16	
Núm. máx. warps por SM	24		32		48		64	
Núm. máx. threads por SM	768		1024		1536		2048	

# Recursos da GPU: atualização

Especificação	Compute Capability							
	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Número de dimensões da grade de blocos	2				3			
Tam. máx. de cada dimensão na grade	65535						2 <sup>31</sup> -1	
Número de dimensões do bloco de threads	3							
Tam. máx. das dimensões x e y no bloco	512				1024			
Tam. máx. da dimensão z no bloco	64							
Núm. máx. threads no bloco	512				1024			
Tamanho do warp	32							
Núm. máx. de blocos por SM	8						16	
Núm. máx. warps por SM	24		32		48		64	
Núm. máx. threads por SM	768		1024		1536		2048	
Núm. registradores por SM	8192		16384		32768		65536	
Mem. compartilhada por SM	16Kb				48Kb			

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 Modelo de Paralelismo
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais

# Métricas de desempenho

## GPU × CPU em soma de matrizes

- Na CPU: **118 157** microssegundos;
- Na GPU: **6 620** microssegundos;
- Na GPU cerca de **18** vezes **mais rápida** a execução.

# Métricas de desempenho

## GPU × CPU em soma de matrizes

- Na CPU: **118 157** microssegundos;
- Na GPU: **6 620** microssegundos;
- Na GPU cerca de **18** vezes **mais rápida** a execução.
- Pode ser melhor?

# Métricas de desempenho

## GPU × CPU em soma de matrizes

- Na CPU: **118 157** microssegundos;
- Na GPU: **6 620** microssegundos;
- Na GPU cerca de **18** vezes **mais rápida** a execução.
- Pode ser melhor?
- Tempo de execução é uma métrica suficiente para avaliar o desempenho paralelo?



# Métricas de desempenho

## GPU em soma de matrizes

- São realizadas  $N \times N$  adições na soma de matrizes;
- Para  $N = 6\,400$ , são realizadas 40 960 000 adições, ou operações de ponto flutuante (*#flop*);
- O tempo de execução (*elapsed\_time*) do kernel foi de 6 620 microssegundos;
- Portanto, a razão entre número de operações e tempo de execução é dada por:

$$\frac{\#flop}{elapsed\_time} = \frac{40\,960\,000\ flop}{6620 \times 10^{-6} s}$$
$$\frac{\#flop}{elapsed\_time} = 6.2 \cdot 10^6\ flop/s = 6.2\ Gflop/s$$

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz)

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz)
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz)
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz)
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;
- $FLOPS \approx 1.15 \times 448 \times 2 = 1030$ ;

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz)
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;
- $FLOPS \approx 1.15 \times 448 \times 2 = 1030$ ;
- Desempenho teórico em precisão simples de **1030** Gflop/s;

# Desempenho máximo teórico: $R_{peak}$

## Tesla C2050 (Fermi)

- 14 SMs com 32 SPs cada: 448 *cores*;
- Frequência de clock de 1.150 GHz ( $f=1.150$  GHz)
- Executa 2 operações de ponto flutuante de precisão simples por ciclo de clock ( $\#flop/cycle=2$ );
- $FLOPS \approx f \times \#cores \times flop/cycle$ ;
- $FLOPS \approx 1.15 \times 448 \times 2 = 1030$ ;
- Desempenho teórico em precisão simples de **1030** Gflop/s;
- Portanto, o desempenho alcançado em soma de matrizes (**6.2** Gflop/s) ficou muito aquém ao pico teórico desta GPU.



# Largura de banda teórica: $BW_{theoretical}$

## Tesla C2050 (Fermi)

- Largura do barramento de memória: 384-bit

# Largura de banda teórica: $BW_{theoretical}$

## Tesla C2050 (Fermi)

- Largura do barramento de memória: 384-bit , ou 48 bytes (384/8);

# Largura de banda teórica: $BW_{theoretical}$

## Tesla C2050 (Fermi)

- Largura do barramento de memória: 384-bit , ou 48 bytes (384/8);
- Frequência de *clock* de memória: 1.5 GHz;

# Largura de banda teórica: $BW_{theoretical}$

## Tesla C2050 (Fermi)

- Largura do barramento de memória: 384-bit , ou 48 bytes (384/8);
- Frequência de *clock* de memória: 1.5 GHz;
- Memória DDR (*double data rate*), que faz dobrar o *clock* para 3.0 GHz;

# Largura de banda teórica: $BW_{theoretical}$

## Tesla C2050 (Fermi)

- Largura do barramento de memória: 384-bit , ou 48 bytes (384/8);
- Frequência de *clock* de memória: 1.5 GHz;
- Memória DDR (*double data rate*), que faz dobrar o *clock* para 3.0 GHz;
- $BW_{theoretical} = (1.5 \times 10^6 \times 2) \times (384/8)/10^9 = 144 \text{ Gbytes/s} = 144 \text{ Gb/s}$ .

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, são carregados da memória global 36 bilhões (144 Gbytes/s / 4 bytes) dados de ponto flutuante por segundo;



# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, são carregados da memória global 36 bilhões (144 Gbytes/s / 4 bytes) dados de ponto flutuante por segundo;
- A razão entre cálculo e acesso a memória global é denominada ***intensidade aritmética***;

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, são carregados da memória global 36 bilhões (144 Gbytes/s / 4 bytes) dados de ponto flutuante por segundo;
- A razão entre cálculo e acesso a memória global é denominada ***intensidade aritmética***;
- Na soma de matrizes, para cada três acessos à memória global, é realizada uma operação de adição;

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, são carregados da memória global 36 bilhões (144 Gbytes/s / 4 bytes) dados de ponto flutuante por segundo;
- A razão entre cálculo e acesso a memória global é denominada ***intensidade aritmética***;
- Na soma de matrizes, para cada três acessos à memória global, é realizada uma operação de adição;
- Portanto, a intensidade aritmética de soma de matrizes é igual a ***1/3***;

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, são carregados da memória global 36 bilhões (144 Gbytes/s / 4 bytes) dados de ponto flutuante por segundo;
- A razão entre cálculo e acesso a memória global é denominada ***intensidade aritmética***;
- Na soma de matrizes, para cada três acessos à memória global, é realizada uma operação de adição;
- Portanto, a intensidade aritmética de soma de matrizes é igual a ***1/3***;
- São realizadas então, no máximo, 12 ( $36 \times \frac{1}{3}$ ) bilhões de operações de ponto flutuante por segundo (***12*** Gflop/s);

# Desempenho máximo teórico $R_{peak}$

Largura de banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda teórica de 144 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, são carregados da memória global 36 bilhões (144 Gbytes/s / 4 bytes) dados de ponto flutuante por segundo;
- A razão entre cálculo e acesso a memória global é denominada **intensidade aritmética**;
- Na soma de matrizes, para cada três acessos à memória global, é realizada uma operação de adição;
- Portanto, a intensidade aritmética de soma de matrizes é igual a **1/3**;
- São realizadas então, no máximo, 12 ( $36 \times \frac{1}{3}$ ) bilhões de operações de ponto flutuante por segundo (**12 Gflop/s**);
- Pico teórico de **1030 Gflop/s**.

# Largura de banda efetiva: $BW_{effective}$

Exemplo dado de soma de matrizes

## Tesla C2050 (Fermi)

- Dimensão da matriz quadrada:  $N = 6400$ ;

# Largura de banda efetiva: $BW_{effective}$

Exemplo dado de soma de matrizes

## Tesla C2050 (Fermi)

- Dimensão da matriz quadrada:  $N = 6400$ ;
- Número de bytes lidos:

$$\begin{aligned}R_B &= (6400 \times 6400 \times 4\text{bytes}) \times 2 \\ &= 163\,840\,000 \times 2 \\ &= 327\,680\,000 \text{ bytes}\end{aligned}$$

# Largura de banda efetiva: $BW_{effective}$

Exemplo dado de soma de matrizes

## Tesla C2050 (Fermi)

- Dimensão da matriz quadrada:  $N = 6400$ ;
- Número de bytes lidos:

$$\begin{aligned}R_B &= (6400 \times 6400 \times 4\text{bytes}) \times 2 \\ &= 163\,840\,000 \times 2 \\ &= 327\,680\,000 \text{ bytes}\end{aligned}$$

- Número de bytes escritos:

$$\begin{aligned}R_W &= (6400 \times 6400 \times 4\text{bytes}) \times 1 \\ &= 163\,840\,000\end{aligned}$$



# Largura de banda efetiva: $BW_{effective}$

Exemplo dado de soma de matrizes

## Tesla C2050 (Fermi)

- Dimensão da matriz quadrada:  $N = 6400$ ;
- Número de bytes lidos:

$$\begin{aligned}R_B &= (6400 \times 6400 \times 4\text{bytes}) \times 2 \\ &= 163\,840\,000 \times 2 \\ &= 327\,680\,000 \text{ bytes}\end{aligned}$$

- Número de bytes escritos:

$$\begin{aligned}R_W &= (6400 \times 6400 \times 4\text{bytes}) \times 1 \\ &= 163\,840\,000\end{aligned}$$

- $Elapsed\_time = 6620 \times 10^{-6} s$

# Largura de banda efetiva: $BW_{effective}$

Exemplo dado de soma de matrizes

## Tesla C2050 (Fermi)

- Largura de banda efetiva:

$$\begin{aligned} BW_{effective} &= \frac{R_B + R_W}{Elapsed\_time \times 10^9} Gb/s \\ &= \frac{327\,680\,000 + 163\,840\,000}{(6620 \times 10^{-6}) \times 10^9} Gb/s \\ &= 74 Gb/s \end{aligned}$$

# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;

# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;

# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, a cada segundo, só podem ser carregados da memória global 18.5 bilhões (74 Gbytes/s/ 4 bytes) de dados de ponto flutuante por segundo;

# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, a cada segundo, só podem ser carregados da memória global 18.5 bilhões (74 Gbytes/s/ 4 bytes) de dados de ponto flutuante por segundo;
- A intensidade aritmética de soma de matrizes é igual a **1/3**;

# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, a cada segundo, só podem ser carregados da memória global 18.5 bilhões (74 Gbytes/s/ 4 bytes) de dados de ponto flutuante por segundo;
- A intensidade aritmética de soma de matrizes é igual a **1/3**;
- São realizadas então, no máximo, 6.2 ( $18.5 \times 1/3$ ) bilhões de operações de ponto flutuante por segundo (**6.2 Gflop/s**);

# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, a cada segundo, só podem ser carregados da memória global 18.5 bilhões (74 Gbytes/s/ 4 bytes) de dados de ponto flutuante por segundo;
- A intensidade aritmética de soma de matrizes é igual a **1/3**;
- São realizadas então, no máximo, 6.2 ( $18.5 \times 1/3$ ) bilhões de operações de ponto flutuante por segundo (**6.2 Gflop/s**);
- Desempenho alcançado de **6.2 Gflop/s**;



# Desempenho máximo teórico $R_{peak}$

Largura de **efetiva** banda como restrição

## Tesla C2050 (Fermi)

- Largura de banda efetiva de 74 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- Logo, a cada segundo, só podem ser carregados da memória global 18.5 bilhões (74 Gbytes/s/ 4 bytes) de dados de ponto flutuante por segundo;
- A intensidade aritmética de soma de matrizes é igual a **1/3**;
- São realizadas então, no máximo, 6.2 (18.5  $\times$  1/3) bilhões de operações de ponto flutuante por segundo (**6.2 Gflop/s**);
- Desempenho alcançado de **6.2 Gflop/s**;
- Pico teórico de **1030 Gflop/s**.

# Desempenho Computacional

## Largura de banda como restrição

### Acesso a memória global

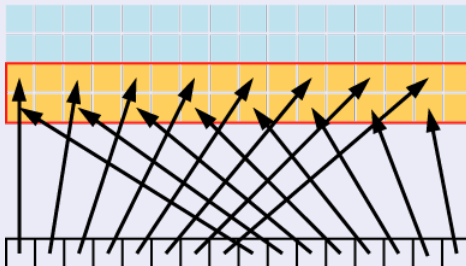
- Em linguagem C, os índices de matrizes são orientados por linha (*row-major*);
- Ou seja, os dados são acessados em endereços contíguos de memória nas linhas da matriz;
- O acesso aos dados que segue este padrão, denomina-se acesso agrupado (*coalesced access pattern*);
- Qualquer outro tipo de acesso,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

## Largura de banda como restrição

### Acesso não agrupado a memória global

- Exemplo: 16 *threads* adjacentes de um *warp* acessando endereços espaçados de 2 posições (*stride-2*) da memória global:



# Desempenho Computacional

## Largura de banda como restrição

```
$ cd codigos/blogforall/cuda-cpp/coalescing-global
```

# Desempenho Computacional

## Largura de banda como restrição

stride.cu

```
template <typename T>
__global__ void strideCopy(T* a, int stride)
{
    int i = (blockDim.x * blockIdx.x + threadIdx.x) * stride;
    a[i] = a[i] + 1;
}
```

```
$ make -f Makefile_stride
$ srun -p treinamento_gpu ./stride
Device: Tesla C2050
Transfer size (MB): 4
Single Precision
```

Stride, Bandwidth (GB/s):

```
1, 97.200623
2, 52.988552
3, 34.232506
4, 24.555544
```

# Desempenho Computacional (cont.)

## Largura de banda como restrição

5, 19.182077  
6, 15.517349  
7, 12.686492  
8, 10.604454  
9, 9.493071  
10, 8.372965  
11, 7.699415  
12, 6.822772  
13, 6.507535  
14, 6.030781  
15, 5.625816  
16, 5.312367  
17, 5.019979  
18, 4.709428  
19, 4.590188  
20, 4.295164  
21, 4.162296  
22, 3.959769  
23, 3.775636  
24, 3.618364  
25, 3.541829

# Desempenho Computacional (cont.)

Largura de banda como restrição

**26, 3.429638**

**27, 3.267504**

**28, 3.190485**

**29, 3.067033**

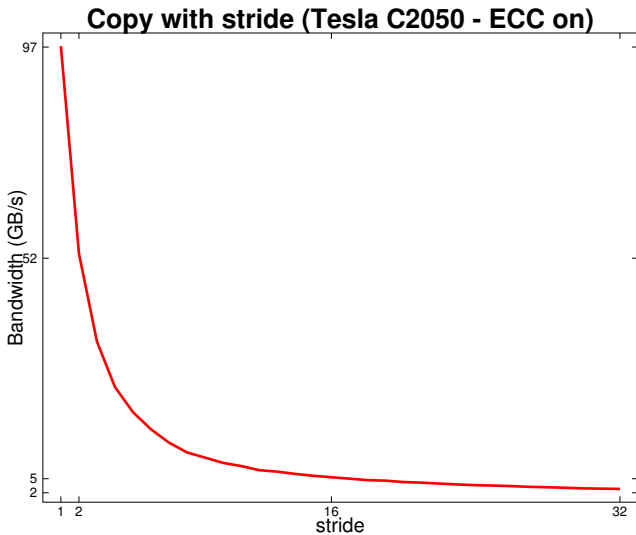
**30, 2.936720**

**31, 2.887870**

**32, 2.829943**

# Desempenho Computacional (cont.)

Largura de banda como restrição





# Produto Escalar

```
$ cd codigos/dot
```

# Produto Escalar em GPU

dot\_gpu\_glmem.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) c[tid] = a[tid] * b[tid];
}

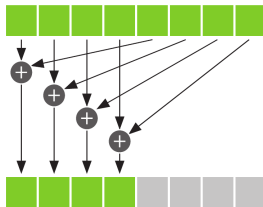
__global__ void dot_reduction( float *c, float *r, int N ) {
    // for reductions, vector size N must be a power of 2
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int i = N/2;
    while (i != 0) {
        if ( tid < i ) c[tid] += c[tid + i];

        i /= 2;
    }
    *r = c[0];
}
```

# Produto Escalar em GPU

dot\_gpu\_glmem.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    if (tid < N) c[tid] = a[tid] * b[tid];  
}  
  
__global__ void dot_reduction( float *c, float *r, int N ) {  
    // for reductions, vector size N must be a power of 2  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int i = N/2;  
    while (i != 0) {  
        if ( tid < i ) c[tid] += c[tid + i];  
  
        i /= 2;  
    }  
    *r = c[0];  
}
```



# Produto Escalar em GPU

dot\_gpu\_glmem.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) c[tid] = a[tid] * b[tid];
}

__global__ void dot_reduction( float *c, float *r, int N ) {
    // for reductions, vector size N must be a power of 2
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int i = N/2;
    while (i != 0) {
        if ( tid < i ) c[tid] += c[tid + i];

        i /= 2;
    }
    *r = c[0];
}
```

```
$ nvcc dot_gpu_glmem.cu -o dot_gpu_glmem
$ srun -p treinamento_gpu ./dot_gpu_glmem `echo 2^24 | bc` 512 1
Size of vector: 16777216
Does GPU value 1.42819e+18 = 3.14824e+21?
$ srun -p treinamento_gpu ./dot_gpu_glmem `echo 2^24 | bc` 512 1
Size of vector: 16777216
Does GPU value 2.40462e+18 = 3.14824e+21?
```

# Produto Escalar em GPU

dot\_gpu\_glmem.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) c[tid] = a[tid] * b[tid];
}

__global__ void dot_reduction( float *c, float *r, int N ) {
    // for reductions, vector size N must be a power of 2
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int i = N/2;
    while (i != 0) {
        if ( tid < i ) c[tid] += c[tid + i];
        //AQUI deveria haver uma barreira de sincronização
        i /= 2;
    }
    *r = c[0];
}
```

- O código não é *thread-safe*;

# Produto Escalar em GPU

dot\_gpu\_glmem.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) c[tid] = a[tid] * b[tid];
}

__global__ void dot_reduction( float *c, float *r, int N ) {
    // for reductions, vector size N must be a power of 2
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int i = N/2;
    while (i != 0) {
        if ( tid < i ) c[tid] += c[tid + i];
        //AQUI deveria haver uma barreira de sincronização
        i /= 2;
    }
    *r = c[0];
}
```

- O código não é *thread-safe*;
- Deveria haver uma barreira de *sincronização* a fim de garantir que todas as *threads* realizem a soma parcial, antes do escalar *i* ser atualizado;

# Produto Escalar em GPU

dot\_gpu\_glmem.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) c[tid] = a[tid] * b[tid];
}

__global__ void dot_reduction( float *c, float *r, int N ) {
    // for reductions, vector size N must be a power of 2
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int i = N/2;
    while (i != 0) {
        if ( tid < i ) c[tid] += c[tid + i];
        //AQUI deveria haver uma barreira de sincronização
        i /= 2;
    }
    *r = c[0];
}
```

- O código não é *thread-safe*;
- Deveria haver uma barreira de *sincronização* a fim de garantir que todas as *threads* realizem a soma parcial, antes do escalar *i* ser atualizado;
- Entretanto, não existe em CUDA um método para sincronizar as *threads* de todos os blocos, tal como é feito internamente *em cada bloco* com `__syncthreads()`.

# Produto Escalar em GPU

dot\_gpu\_glmem\_sync.cu

```
__global__ void dot_prod( float *a, float *b, float *c, int N ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) c[tid] = a[tid] * b[tid];
}

__global__ void dot_reduction( float *c, int i ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if ( tid < i ) c[tid] += c[tid + i];
}

int main(int argc, char* argv[])
{
    ...

    // for reductions, vector size N must be a power of 2
    int i = N/2;
    printf("i: %d\n",i);
    while (i != 0) {
        dot_reduction<<<GridSize,BlockSize>>>( dev_c, i );
        checkCuda( cudaDeviceSynchronize() );
        i /= 2;
        printf("i: %d\n",i);
    }

    checkCuda( cudaMemcpy( c, dev_c, N*sizeof(float), cudaMemcpyDeviceToHost ) );
    r = c[0];

    ...
}
```



# Produto Escalar em GPU

dot\_gpu\_glmem\_sync.cu

```
// for reductions, vector size N must be a power of 2
int i = N/2;
printf("i: %d\n",i);
while (i != 0) {
    dot_reduction<<<GridSize,BlockSize>>( dev_c, i );
    checkCuda( cudaDeviceSynchronize() ); //bloqueia a CPU
    i /= 2;
    printf("i: %d\n",i);
}

checkCuda( cudaMemcpy( c, dev_c, N*sizeof(float), cudaMemcpyDeviceToHost ) );
r = c[0];
}
```

- O laço de redução foi transferido para o programa principal;

# Produto Escalar em GPU

dot\_gpu\_glmem\_sync.cu

```
// for reductions, vector size N must be a power of 2
int i = N/2;
printf("i: %d\n",i);
while (i != 0) {
    dot_reduction<<<GridSize,BlockSize>>( dev_c, i );
    checkCuda( cudaDeviceSynchronize() ); //bloqueia a CPU
    i /= 2;
    printf("i: %d\n",i);
}

checkCuda( cudaMemcpy( c, dev_c, N*sizeof(float), cudaMemcpyDeviceToHost ) );
r = c[0];
}
```

- O laço de redução foi transferido para o programa principal;
- `cudaDeviceSynchronize()` bloqueia a execução em CPU até que o *kernel* seja finalizado.

# Produto Escalar em GPU

```
$ nvcc dot_gpu_glmem_sync.cu -o dot_gpu_glmem_sync
$ srun -p treinamento_gpu ./dot_gpu_glmem_sync `echo 2^24 | bc` 512 1
Size of vector: 16777216
i: 8388608
i: 4194304
i: 2097152
i: 1048576
i: 524288
i: 262144
i: 131072
i: 65536
i: 32768
i: 16384
i: 8192
i: 4096
i: 2048
i: 1024
i: 512
i: 256
i: 128
i: 64
i: 32
```

# Produto Escalar em GPU (cont.)

i: 16

i: 8

i: 4

i: 2

i: 1

i: 0

Does GPU value  $3.14824e+21 = 3.14824e+21$ ?

# Produto Escalar em GPU

```
$ srun -p treinamento_gpu nvprof --print-gpu-trace ./dot_gpu_glmem_sync `echo 2^24 | bc` 512 1
===== NVPROF is profiling dot_gpu_glmem_sync...
===== Command: dot_gpu_glmem_sync 16777216 512 1
Size of vector: 16777216

===== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
38.50      39.87ms      24      1.66ms    1.62ms    1.90ms    dot_reduction(float*, int)
29.96      31.03ms      1      31.03ms   31.03ms   31.03ms    [CUDA memcpy DtoH]
29.48      30.53ms      2      15.26ms   15.09ms   15.44ms    [CUDA memcpy HtoD]
 2.06      2.14ms      1      2.14ms    2.14ms    2.14ms    dot_prod(float*, float*, float*, int)
```

# Produto Escalar em GPU

```
$ srun -p treinamento_gpu nvprof --print-gpu-trace ./dot_gpu_glmem_sync `echo 2^24 | bc` 512 1
Duration      Grid Size    Block Size   Name
15.07ms       -            -            [CUDA memcpy HtoD]
14.86ms       -            -            [CUDA memcpy HtoD]
2.14ms        (32768 1 1) (512 1 1)   dot_prod(float*, float*, float*, int)
1.91ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.77ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.72ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.69ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.67ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.63ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.65ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.65ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.65ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.66ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.66ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.66ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.66ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.66ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.65ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.65ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
1.64ms        (32768 1 1) (512 1 1)   dot_reduction(float*, int)
32.81ms       -            -            [CUDA memcpy DtoH]
```

# Produto Escalar em GPU: memória compartilhada

B0				B1			
T0	T1	T2	T3	T0	T1	T2	T3

# Produto Escalar em GPU: memória compartilhada

Inicialização das *threads*

B0				B1			
T0 temp=0	T1 temp=0	T2 temp=0	T3 temp=0	T0 temp=0	T1 temp=0	T2 temp=0	T3 temp=0



# Produto Escalar em GPU: memória compartilhada

Atualização das *threads*, com os valores de  $a[i] * b[i]$ , com  $i = 0, \dots, N - 1$

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

B0				B1			
T0 temp += a[0]*b[0]	T1 temp += a[1]*b[1]	T2 temp += a[2]*b[2]	T3 temp += a[3]*b[3]	T0 temp += a[4]*b[4]	T1 temp += a[5]*b[5]	T2 temp += a[6]*b[6]	T3 temp += a[7]*b[7]

# Produto Escalar em GPU: memória compartilhada

Atualização das *threads*, com os valores de  $a[i] * b[i]$ , com  $i = 0, \dots, N - 1$

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

B0				B1			
T0 temp += a[8]*b[8]	T1 temp += a[9]*b[9]	T2 temp += a[10]*b[10]	T3 temp += a[11]*b[11]	T0 temp += a[12]*b[12]	T1 temp += a[13]*b[13]	T2 temp += a[14]*b[14]	T3 temp += a[15]*b[15]

# Produto Escalar em GPU: memória compartilhada

Operação de redução em cada bloco

B0				B1			
T0	T1	T2	T3	T0	T1	T2	T3

# Produto Escalar em GPU: memória compartilhada

Operação de redução em cada bloco

B0				B1			
<b>T0+=T2</b>	<b>T1+=T3</b>	T2	T3	<b>T0+=T2</b>	<b>T1+=T3</b>	T2	T3

# Produto Escalar em GPU: memória compartilhada

Operação de redução em cada bloco

B0				B1			
<b>T0</b>	<b>T1</b>	T2	T3	<b>T0</b>	<b>T1</b>	T2	T3

# Produto Escalar em GPU: memória compartilhada

Operação de redução em cada bloco

B0				B1			
<b>T0+=T1</b>	T1	T2	T3	<b>T0+=T1</b>	T1	T2	T3

# Produto Escalar em GPU: memória compartilhada

Operação de redução em cada bloco

B0				B1			
T0	T1	T2	T3	T0	T1	T2	T3

# Produto Escalar em GPU: memória compartilhada

Resultado armazenado no array  $C$ , com número de elementos igual ao número de blocos na grade

B0				B1			
T0	T1	T2	T3	T0	T1	T2	T3

$C(0)=T0$

$C(1)=T0$



# Produto Escalar em GPU: memória compartilhada

dot\_gpu.cu

```
__global__ void dot( float *a, float *b, float *c, int N ) {
    __shared__ float cache[BlockSize];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();

    // for reductions, BlockSize must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
}
```

# Produto Escalar em GPU: memória compartilhada (cont.)

```
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}
```

```
$ nvcc dot_gpu.cu -o dot_gpu
$ srun -p treinamento_gpu ./dot_gpu `echo 2^24 | bc` 1
Does GPU value 3.14824e+21 = 3.14824e+21?
```

```
$ srun -p treinamento_gpu nvprof ./dot_gpu `echo 2^24 | bc` 1
===== NVPROF is profiling dot_gpu...
===== Command: dot_gpu 16777216 1
Does GPU value 3.14824e+21 = 3.14824e+21?
===== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
95.10      31.53ms      2      15.76ms  15.73ms  15.79ms  [CUDA memcpy HtoD]
 4.89      1.62ms      1      1.62ms   1.62ms   1.62ms   dot(float*, float*, float*, int)
 0.01      2.02us      1      2.02us   2.02us   2.02us   [CUDA memcpy DtoH]
```

# Produto Escalar em GPU: shared × global

## dot\_gpu.cu

```
$ srun -p treinamento_gpu nvprof ./dot_gpu `echo 2^24 | bc` 1
===== NVPROF is profiling dot_gpu...
===== Command: dot_gpu 16777216 1

===== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
95.10      31.53ms      2      15.76ms  15.73ms  15.79ms  [CUDA memcpy HtoD]
 4.89      1.62ms      1      1.62ms   1.62ms   1.62ms  dot(float*, float*, float*, int)
 0.01      2.02us      1      2.02us   2.02us   2.02us  [CUDA memcpy DtoH]
```

## dot\_gpu\_glmem\_sync.cu

```
$ srun -p treinamento_gpu nvprof ./dot_gpu_glmem_sync `echo 2^24 | bc` 512 1
===== NVPROF is profiling dot_gpu_glmem_sync...
===== Command: dot_gpu_glmem_sync 16777216 512 1

===== Profiling result:
Time(%)      Time      Calls      Avg      Min      Max      Name
38.50      39.87ms     24      1.66ms   1.62ms   1.90ms  dot_reduction(float*, int)
29.96      31.03ms      1     31.03ms  31.03ms  31.03ms  [CUDA memcpy DtoH]
29.48      30.53ms      2     15.26ms  15.09ms  15.44ms  [CUDA memcpy HtoD]
 2.06      2.14ms      1      2.14ms   2.14ms   2.14ms  dot_prod(float*, float*, float*, int)
```

# Produto Escalar em GPU: memória compartilhada

- Acesso extremamente rápido: (*on-chip memory*);

# Produto Escalar em GPU: memória compartilhada

- Acesso extremamente rápido: (*on-chip memory*);
- Redução no *tráfego de acesso* à memória global.

# Produto Escalar em GPU: memória compartilhada

- Acesso extremamente rápido: (*on-chip memory*);
- Redução no *tráfego de acesso* à memória global.
- **Cooperação**: cada *thread* de um bloco carrega um dado da memória global, e todas outras *threads* do mesmo bloco podem ler este dado;

# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 Modelo de Paralelismo
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - Módulo 5: Hieraquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 **Módulo 7: Multiplicação de Matrizes**
- 5 Considerações Finais

# Soma de matrizes na GPU

- Na prática vimos que a soma de vetores e matrizes em GPU não são bons exemplos de aplicações com potencial ganho de desempenho;
- O aumento do tamanho dos dados cresce na mesma ordem de grandeza que o aumento de processamento;
- No caso das matrizes, para cada  $N^2$  dados transferidos, são realizadas  $N^2$  operações;
- Dado que a transferência dos dados da CPU para GPU é uma operação custosa, o possível ganho de desempenho no processamento, é perdido com o tempo de transferência.



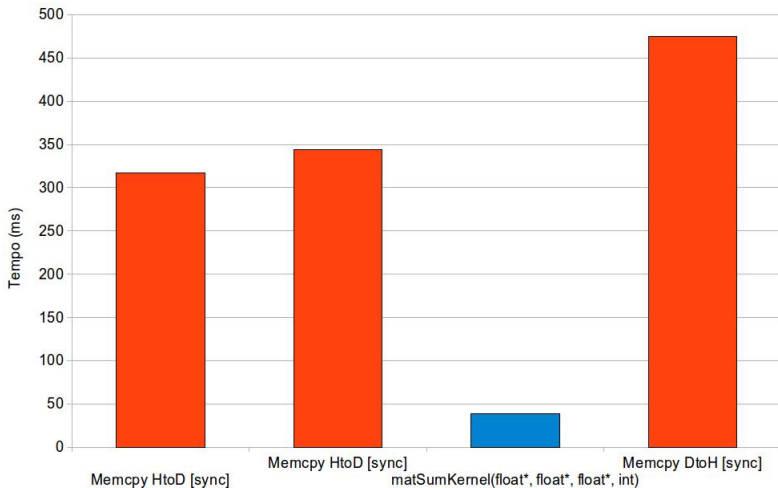
# Multiplicação de matrizes na GPU

- Por este raciocínio, uma aplicação com bom potencial de ganho de desempenho é a multiplicação de matrizes;
- Enquanto que a quantidade de dados cresce quadraticamente, a quantidade de processamento cresce na ordem cúbica.
- Logo, para cada  $N^2$  dados transferidos, são realizadas  $N^3$  operações;
- Para um  $N$  suficientemente grande, o custo de transferência dos dados torna-se pouco importante com relação ao tempo de processamento.

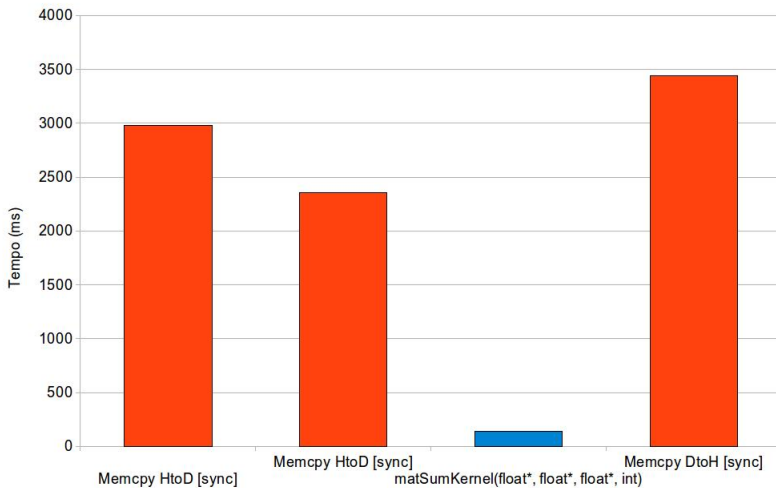
# Multiplicação de matrizes na GPU

Os códigos apresentados nesta seção foram retirados do livro de [Kirk and Hwu, 2012].

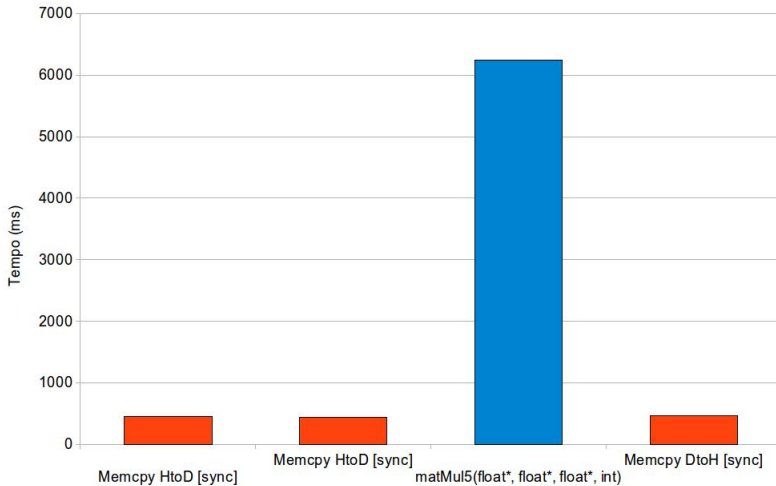
## SOMA DE MATRIZES



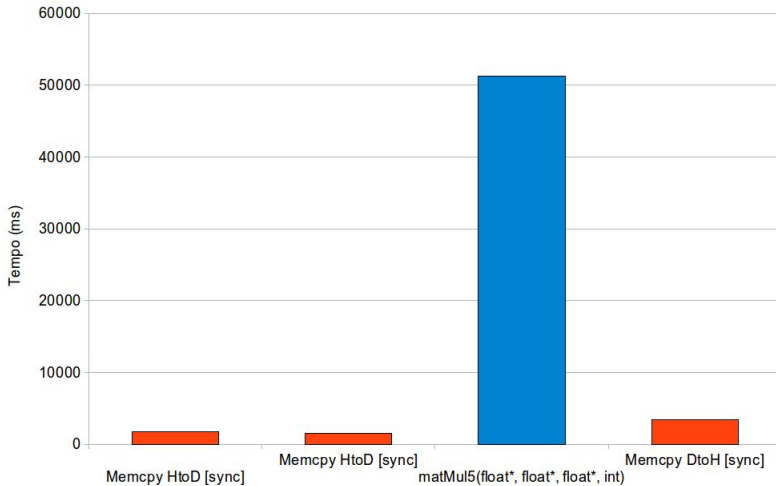
## SOMA DE MATRIZES



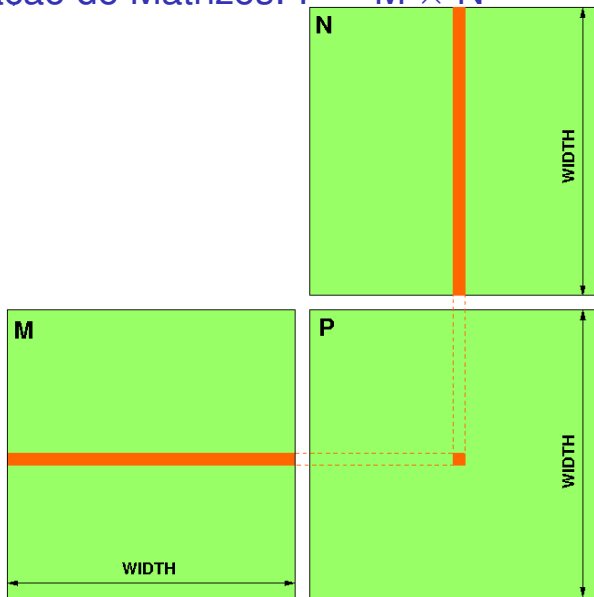
## MULTIPLICAÇÃO DE MATRIZES



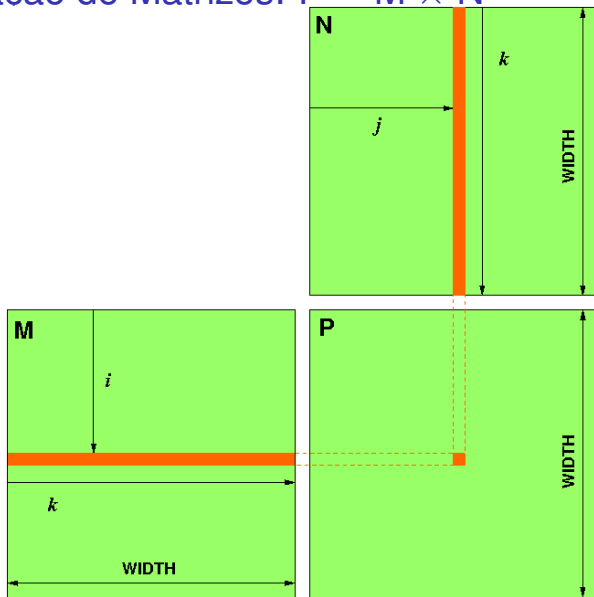
## MULTIPLICAÇÃO DE MATRIZES



# Multiplicação de Matrizes: $P = M \times N$



# Multiplicação de Matrizes: $P = M \times N$





# Multiplicação de Matrizes: $P = M \times N$

## Paralelismo de Dados

- Matrizes quadradas  $M$  e  $N$  de dimensão **WIDTH**;
- Cada elemento  $P_{ij}$  da matriz  $P$  é resultado do **produto escalar** entre a  $i$ -ésima linha da matriz  $M$  e a  $j$ -ésima coluna de  $N$ ;
- Há **WIDTH<sup>2</sup> independentes** produtos escalares a serem calculados;
- Cada *thread* executa um produto escalar, ou seja, calcula um elemento da matriz  $P$ .

# Código em CPU

## matMul\_cpu.cu

```
// Computes the matrix product using line matrices:
void matMul(float* P, float* M, float* N, unsigned int Width) {
    for (unsigned int i = 0; i < Width; ++i) {
        for (unsigned int j = 0; j < Width; ++j) {
            P[i * Width + j] = 0.0;
            for (unsigned int k = 0; k < Width; ++k) {
                P[i * Width + j] += M[i * Width + k] * N[k * Width + j];
            }
        }
    }
}
```

## matMul\_cpu.cu

```
matMul( P, M, N, Width );
```

# Código em CPU

```
$ nvcc -O2 matMul_cpu -o matMul_cpu
$ srun -p treinamento_gpu ./matMul_cpu 640
Allocate memory for matrices M and N...
Initialize matrices...
Multiply matrices...

Execution Time (microseconds): 612705.00
```

# Multiplicação de matrizes na GPU

## Mapeamento dos índices dos elementos

$$i = by * (\text{blocos na horizontal}) + ty$$

$$i = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y};$$

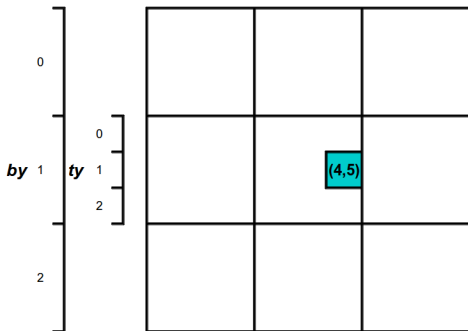
$$i = 1 * 3 + 1 = 4;$$

$$j = bx * (\text{blocos na vertical}) + tx$$

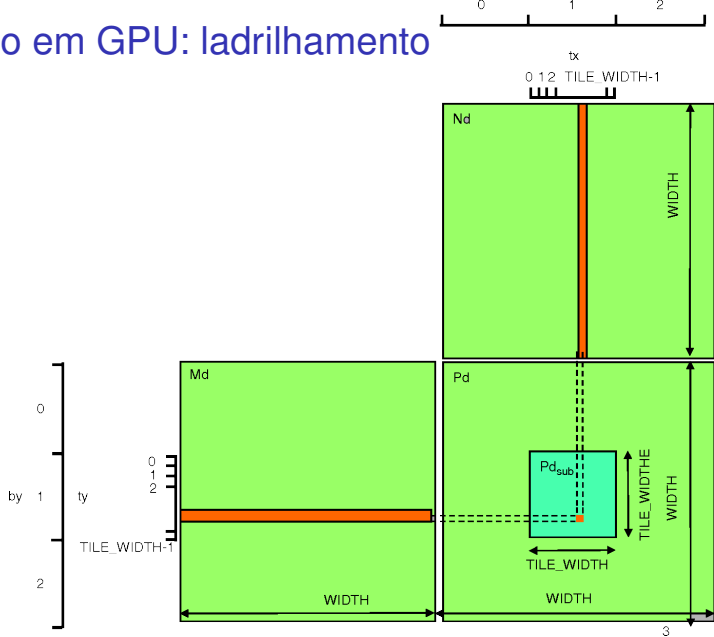
$$j = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$

$$j = 1 * 3 + 2 = 5;$$

$$tid = i * \text{Width} + j$$



# Código em GPU: ladrilhamento

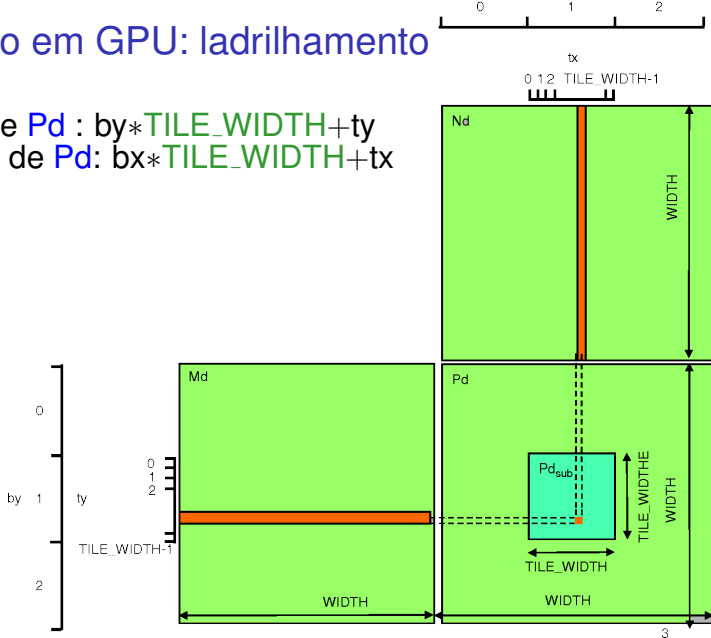


## Código em GPU: ladrilhamento

- A matriz  $P_d$  é particionada em blocos,
- Sub-matrizes quadradas de tamanho `TILE_WIDTH`;
- Uma sub-matriz forma um **ladrilho** (*tile* em inglês);
- Cada *thread* continua calculando um elemento da matriz  $P_d$ ;
- Identificar um elemento de cada sub-matriz, consiste agora na identificação da *thread*  $(tx,ty)$  pertencente ao bloco  $(bx,by)$ ;

# Código em GPU: ladrilhamento

linha de  $P_d$  :  $by * TILE\_WIDTH + ty$   
coluna de  $P_d$  :  $bx * TILE\_WIDTH + tx$



# Código em GPU

## matMul\_gpu.cu

```
__global__ void matMul(float* Pd, float* Md, float* Nd, int Width) {
    float Pvalue = 0.0;

    int j = blockIdx.x * Tile_Width + threadIdx.x;
    int i = blockIdx.y * Tile_Width + threadIdx.y;

    for (int k = 0; k < Width; ++k) {
        Pvalue += Md[i * Width + k] * Nd[k * Width + j];
    }

    Pd[i * Width + j] = Pvalue;
}
```

## matMul\_gpu.cu

```
int GridSize = (Width + Tile_Width-1) / Tile_Width;
dim3 gridDim(GridSize, GridSize);
dim3 blockDim(Tile_Width, Tile_Width);

matMul<<< gridDim, blockDim >>>(Pd, Md, Nd, Width);
```



# Código em GPU

```
$ nvcc matMul_gpu -o matMul_gpu
$ srun -p treinamento_gpu nvprof ./matMul_gpu 640 1
===== NVPROF is profiling matMul_gpu...
===== Command: matMul_gpu 640 1
Allocate host memory for matrices M and N...
Initialize host matrices...
Allocate device matrices (linearized)...
Execute the kernel...
Device: Tesla C2050
===== Profiling result:
Time(%)      Time      Calls   Name
 84.73      8.20ms      1  matMul(float*, float*, float*, int)
  8.25     798.01us      1  [CUDA memcpy DtoH]
  7.02     679.48us      2  [CUDA memcpy HtoD]
```

# Multiplicação de Matrizes: GPU × CPU

## Comparação de desempenho

- Na CPU: **612** milissegundos;
- Na GPU: **8.2** milissegundos;
- Na GPU cerca de **74** vezes **mais rápida** a execução.

# Código em GPU: acesso agrupado

## matMul\_gpu.cu

```
__global__ void matMul(float* Pd, float* Md, float* Nd, int Width) {  
    float Pvalue = 0.0;  
  
    int j = blockIdx.x * Tile_Width + threadIdx.x;  
    int i = blockIdx.y * Tile_Width + threadIdx.y;  
  
    for (int k = 0; k < Width; ++k) {  
        Pvalue += Md[i * Width + k] * Nd[k * Width + j];  
    }  
  
    Pd[i * Width + j] = Pvalue;  
}
```

# Código em GPU: acesso não agrupado

## matMul\_gpu\_uncoalesced.cu

```
__global__ void matMul(float* Pd, float* Md, float* Nd, int Width) {  
    float Pvalue = 0.0;  
  
    int j = blockIdx.x * Tile_Width + threadIdx.x;  
    int i = blockIdx.y * Tile_Width + threadIdx.y;  
  
    for (int k = 0; k < Width; ++k) {  
        Pvalue += Md[j * Width + k] * Nd[k * Width + i];  
    }  
  
    Pd[j * Width + i] = Pvalue;  
}
```

## Código em GPU: acesso não agrupado

```
$ nvcc matMul_gpu_uncoalesced.cu -o matMul_gpu_uncoalesced
$ srun -p treinamento_gpu nvprof ./matMul_gpu_uncoalesced 640 1
===== NVPROF is profiling matMul_gpu_uncoalesced...
===== Command: matMul_gpu_uncoalesced 640 1
Allocate host memory for matrices M and N...
Initialize host matrices...
Allocate device matrices (linearized)...
Execute the kernel...
Device: Tesla C2050
===== Profiling result:
Time(%)      Time      Calls      Name
  95.14     29.31ms         1  matMul(float*, float*, float*, int)
   2.60     800.70us         1  [CUDA memcpy DtoH]
   2.26     696.64us         2  [CUDA memcpy HtoD]
```

# Multiplicação de Matrizes: GPU × CPU

## Comparação de desempenho

- Na CPU : **612** milissegundos;
- Na GPU (acesso agrupado) : **8.2** milissegundos;
- Na GPU (acesso não agrupado): **29.31** milissegundos;

# Multiplicação de matrizes: memória global

- Acesso é lento (*off-chip memory*);
- No kernel de multiplicação de matriz, para cada dois acessos à memória global (uma leitura de  $M_d$  e uma leitura de  $N_d$ ) ocorrem duas operações de ponto flutuante (uma adição e uma multiplicação);

matMul\_gpu.cu

```
__global__ void matMul(float* Pd, float* Md, float* Nd, int Width)
{
    ...

    for (int k = 0; k < Width; ++k) {
        Pvalue += Md[i * Width + k] * Nd[k * Width + j];
    }
}
```

## Multiplicação de matrizes: memória global

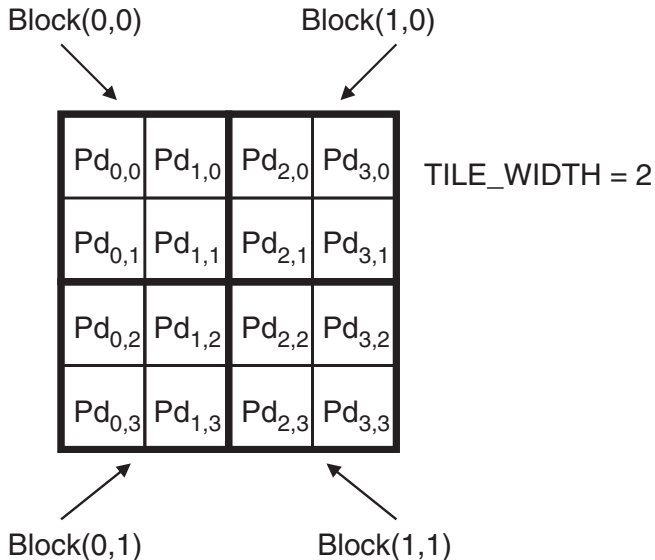
- A razão entre cálculo e acesso a memória global, a **intensidade aritmética**, é igual a 1;
- *Device* GT200: largura de banda de 141,7 Gb/s;
- Dado de ponto flutuante com precisão simples: 4 bytes;
- A cada segundo, só podem ser carregados da memória global 35,4 (141,7/4) bilhões de dados por segundo;
- Para cada dado carregado, há uma operação de ponto flutuante (intensidade aritmética=1.0);
- São realizadas então, no máximo, 35,4 bilhões de operações de ponto flutuante por segundo (**35,4 gigaflops**);
- O *device* GT200 tem pico teórico de **933,1 gigaflops**;



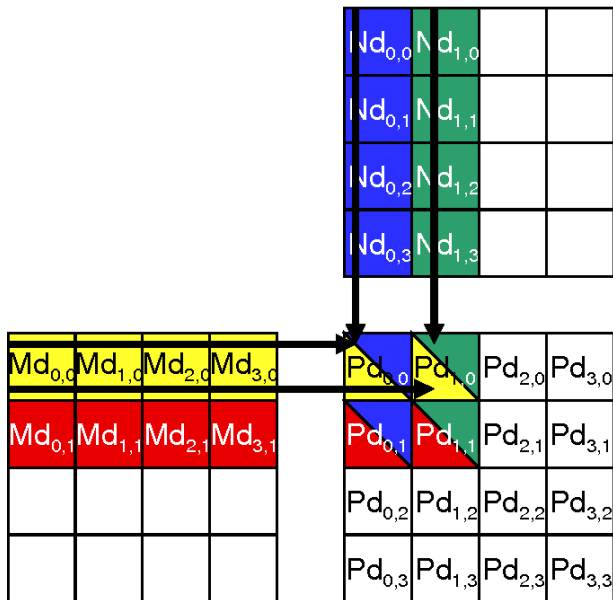
# Multiplicação de matrizes: memória compartilhada

- Acesso extremamente rápido (*on-chip memory*);
- **Cooperação**: cada thread de um bloco carrega um dado da memória global, e todas outras threads do mesmo bloco podem ler este dado;
- Ocorre conseqüente redução no *tráfego de acesso* à memória global.

# Multiplicação de matrizes: ladrilhamento



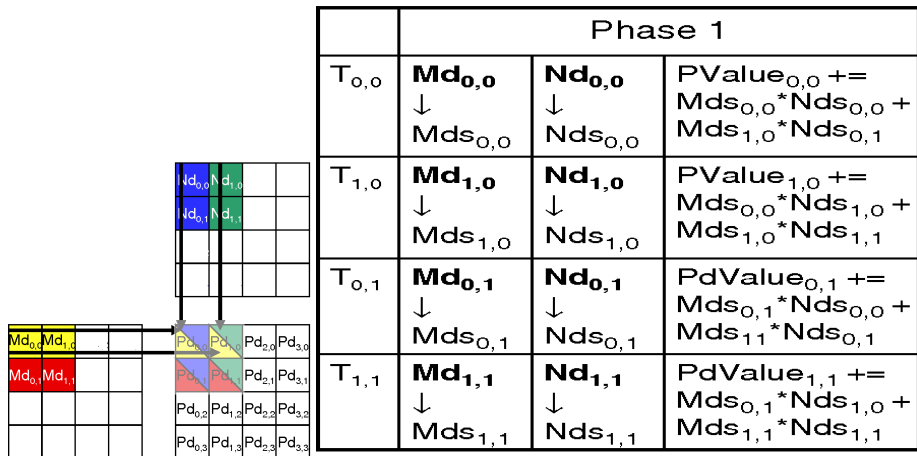
# Multiplicação de matrizes: ladrilhamento





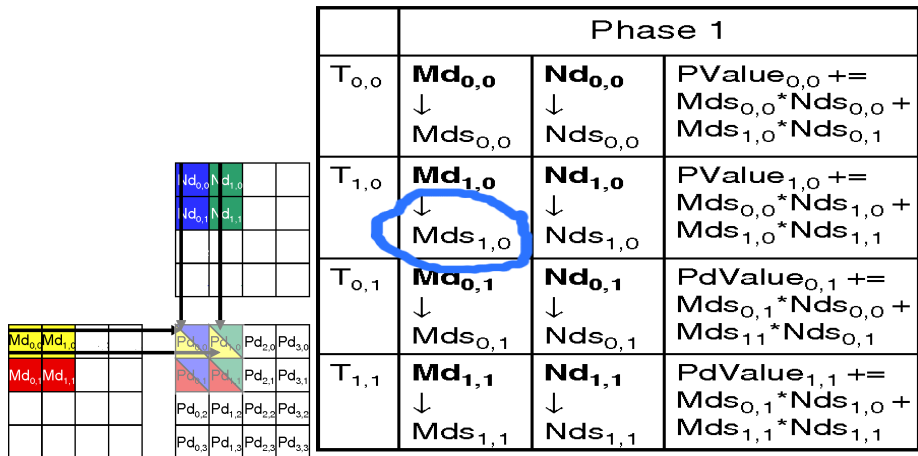
# Multiplicação de matrizes: memória compartilhada

$Md$  e  $Nd$  também são divididos em ladrilhos.



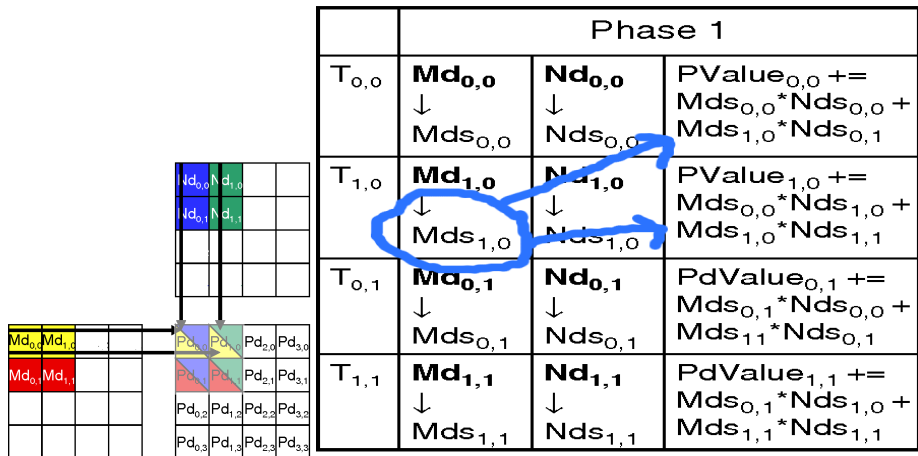
# Multiplicação de matrizes: memória compartilhada

Um elemento dos ladrilhos de  $Md$  e de  $Nd$  é lido uma só vez na memória global, para um ladrilho (bloco) em  $Pd$ .



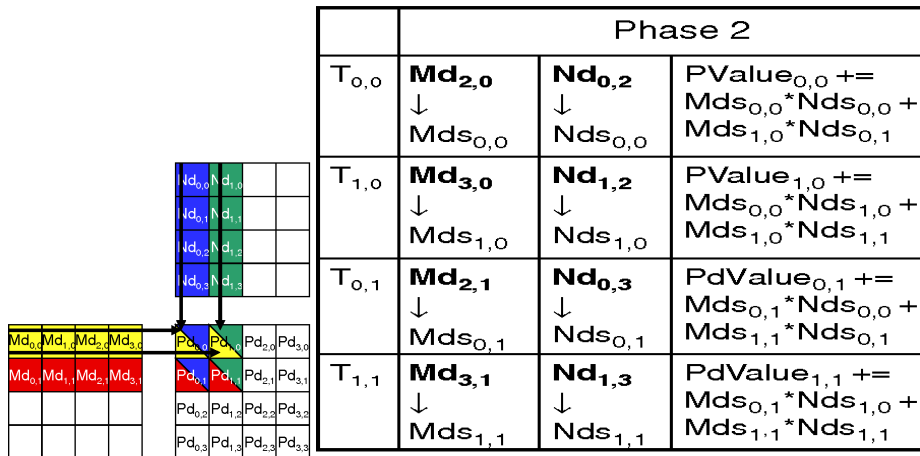
# Multiplicação de matrizes: memória compartilhada

E utilizado duas vezes para cálculo em operações de ponto flutuante.



# Multiplicação de matrizes: memória compartilhada

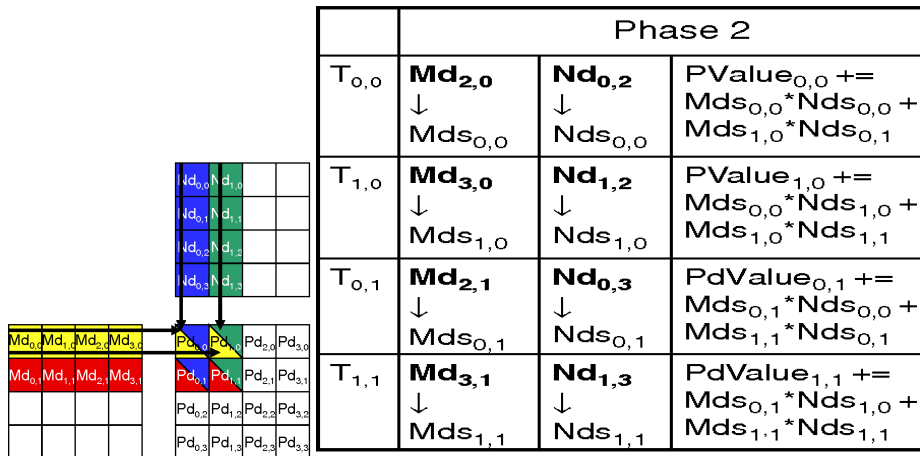
Os elementos de **Md** e **Nd** do bloco são enviados à memória compartilhada em duas etapas.





# Multiplicação de matrizes: memória compartilhada

O número de etapas é dado pela razão  $\text{Width}/\text{TILE\_WIDTH}$ ;



# Multiplicação de matrizes: memória compartilhada

- Tráfego à memória global é potencialmente reduzido à metade;
- Ladrilhos de dimensão  $4 \times 4$ : redução de  $1/4$ ;
- Ladrilhos de dimensão  $8 \times 8$ : redução de  $1/8$ ;
- Ladrilhos de dimensão  $16 \times 16$ : potencial redução de  $1/16$  no tráfego à memória global;
- Neste caso, a largura de banda pode atender uma taxa de cálculo de ponto flutuante muito maior;
- Portanto, é possível atingir agora  $(141,7/4) \times 16 = 566,8$  gigaflops.

# Memória compartilhada: sincronização de threads

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8

	Phase 1		
$T_{0,0}$			
$T_{1,0}$			
$T_{0,1}$			
$T_{1,1}$			

# Memória compartilhada: sincronização de threads

1 Leitura dos dados

2

3

4

5

6

7

8

	Phase 1		
$T_{0,0}$	<b>Md<sub>0,0</sub></b> ↓ Mds <sub>0,0</sub>	<b>Nd<sub>0,0</sub></b> ↓ Nds <sub>0,0</sub>	
$T_{1,0}$	<b>Md<sub>1,0</sub></b> ↓ Mds <sub>1,0</sub>	<b>Nd<sub>1,0</sub></b> ↓ Nds <sub>1,0</sub>	
$T_{0,1}$	<b>Md<sub>0,1</sub></b> ↓ Mds <sub>0,1</sub>	<b>Nd<sub>0,1</sub></b> ↓ Nds <sub>0,1</sub>	
$T_{1,1}$	<b>Md<sub>1,1</sub></b> ↓ Mds <sub>1,1</sub>	<b>Nd<sub>1,1</sub></b> ↓ Nds <sub>1,1</sub>	

# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3
- 4
- 5
- 6
- 7
- 8

	Phase 1	
$T_{0,0}$	<b>Md<sub>0,0</sub></b> ↓ Mds <sub>0,0</sub>	<b>Nd<sub>0,0</sub></b> ↓ Nds <sub>0,0</sub>
$T_{1,0}$	<b>Md<sub>1,0</sub></b> ↓ Mds <sub>1,0</sub>	<b>Nd<sub>1,0</sub></b> ↓ Nds <sub>1,0</sub>
$T_{0,1}$	<b>Md<sub>0,1</sub></b> ↓ Mds <sub>0,1</sub>	<b>Nd<sub>0,1</sub></b> ↓ Nds <sub>0,1</sub>
$T_{1,1}$	<b>Md<sub>1,1</sub></b> ↓ Mds <sub>1,1</sub>	<b>Nd<sub>1,1</sub></b> ↓ Nds <sub>1,1</sub>

# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4
- 5
- 6
- 7
- 8

	Phase 1		
$T_{0,0}$	$\mathbf{Md}_{0,0}$ ↓ $Mds_{0,0}$	$\mathbf{Nd}_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{1,0} * Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md}_{1,0}$ ↓ $Mds_{1,0}$	$\mathbf{Nd}_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0} +$ $Mds_{1,0} * Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md}_{0,1}$ ↓ $Mds_{0,1}$	$\mathbf{Nd}_{0,1}$ ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md}_{1,1}$ ↓ $Mds_{1,1}$	$\mathbf{Nd}_{1,1}$ ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0} +$ $Mds_{1,1} * Nds_{1,1}$

# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4 Barreira: `__syncthreads()`
- 5
- 6
- 7
- 8

	Phase 1		
$T_{0,0}$	$\mathbf{Md}_{0,0}$ ↓ $Mds_{0,0}$	$\mathbf{Nd}_{0,0}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0}$ $Mds_{1,0} * Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md}_{1,0}$ ↓ $Mds_{1,0}$	$\mathbf{Nd}_{1,0}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0}$ $Mds_{1,0} * Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md}_{0,1}$ ↓ $Mds_{0,1}$	$\mathbf{Nd}_{0,1}$ ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0}$ $Mds_{1,1} * Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md}_{1,1}$ ↓ $Mds_{1,1}$	$\mathbf{Nd}_{1,1}$ ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0}$ $Mds_{1,1} * Nds_{1,1}$

# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4 Barreira: `__syncthreads()`
- 5
- 6
- 7
- 8

	Phase 2		
$T_{0,0}$			
$T_{1,0}$			
$T_{0,1}$			
$T_{1,1}$			



# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4 Barreira: `__syncthreads()`
- 5 Leitura dos dados
- 6
- 7
- 8

	Phase 2		
$T_{0,0}$	<b>Md<sub>2,0</sub></b> ↓ Mds <sub>0,0</sub>	<b>Nd<sub>0,2</sub></b> ↓ Nds <sub>0,0</sub>	
$T_{1,0}$	<b>Md<sub>3,0</sub></b> ↓ Mds <sub>1,0</sub>	<b>Nd<sub>1,2</sub></b> ↓ Nds <sub>1,0</sub>	
$T_{0,1}$	<b>Md<sub>2,1</sub></b> ↓ Mds <sub>0,1</sub>	<b>Nd<sub>0,3</sub></b> ↓ Nds <sub>0,1</sub>	
$T_{1,1}$	<b>Md<sub>3,1</sub></b> ↓ Mds <sub>1,1</sub>	<b>Nd<sub>1,3</sub></b> ↓ Nds <sub>1,1</sub>	

# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4 Barreira: `__syncthreads()`
- 5 Leitura dos dados
- 6 Barreira: `__syncthreads()`
- 7
- 8

	Phase 2	
$T_{0,0}$	<b>Md<sub>2,0</sub></b> ↓ Mds <sub>0,0</sub>	<b>Nd<sub>0,2</sub></b> ↓ Nds <sub>0,0</sub>
$T_{1,0}$	<b>Md<sub>3,0</sub></b> ↓ Mds <sub>1,0</sub>	<b>Nd<sub>1,2</sub></b> ↓ Nds <sub>1,0</sub>
$T_{0,1}$	<b>Md<sub>2,1</sub></b> ↓ Mds <sub>0,1</sub>	<b>Nd<sub>0,3</sub></b> ↓ Nds <sub>0,1</sub>
$T_{1,1}$	<b>Md<sub>3,1</sub></b> ↓ Mds <sub>1,1</sub>	<b>Nd<sub>1,3</sub></b> ↓ Nds <sub>1,1</sub>

# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4 Barreira: `__syncthreads()`
- 5 Leitura dos dados
- 6 Barreira: `__syncthreads()`
- 7 Cálculo de Pvalue
- 8

	Phase 2		
$T_{0,0}$	$\mathbf{Md}_{2,0}$ ↓ $Mds_{0,0}$	$\mathbf{Nd}_{0,2}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0} +$ $Mds_{1,0} * Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md}_{3,0}$ ↓ $Mds_{1,0}$	$\mathbf{Nd}_{1,2}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0} +$ $Mds_{1,0} * Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md}_{2,1}$ ↓ $Mds_{0,1}$	$\mathbf{Nd}_{0,3}$ ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0} +$ $Mds_{1,1} * Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md}_{3,1}$ ↓ $Mds_{1,1}$	$\mathbf{Nd}_{1,3}$ ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0} +$ $Mds_{1,1} * Nds_{1,1}$

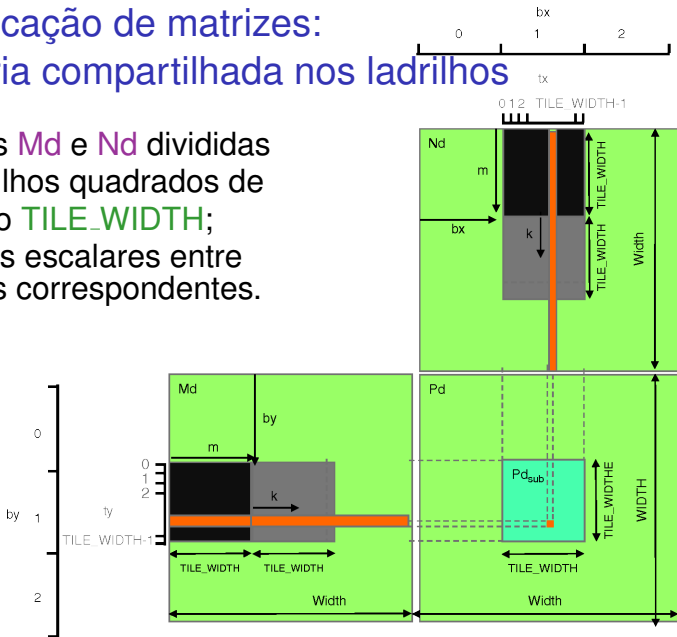
# Memória compartilhada: sincronização de threads

- 1 Leitura dos dados
- 2 Barreira: `__syncthreads()`
- 3 Cálculo de Pvalue
- 4 Barreira: `__syncthreads()`
- 5 Leitura dos dados
- 6 Barreira: `__syncthreads()`
- 7 Cálculo de Pvalue
- 8 Barreira: `__syncthreads()`

	Phase 2		
$T_{0,0}$	$\mathbf{Md}_{2,0}$ ↓ $Mds_{0,0}$	$\mathbf{Nd}_{0,2}$ ↓ $Nds_{0,0}$	$PValue_{0,0} +=$ $Mds_{0,0} * Nds_{0,0}$ $Mds_{1,0} * Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md}_{3,0}$ ↓ $Mds_{1,0}$	$\mathbf{Nd}_{1,2}$ ↓ $Nds_{1,0}$	$PValue_{1,0} +=$ $Mds_{0,0} * Nds_{1,0}$ $Mds_{1,0} * Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md}_{2,1}$ ↓ $Mds_{0,1}$	$\mathbf{Nd}_{0,3}$ ↓ $Nds_{0,1}$	$PdValue_{0,1} +=$ $Mds_{0,1} * Nds_{0,0}$ $Mds_{1,1} * Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md}_{3,1}$ ↓ $Mds_{1,1}$	$\mathbf{Nd}_{1,3}$ ↓ $Nds_{1,1}$	$PdValue_{1,1} +=$ $Mds_{0,1} * Nds_{1,0}$ $Mds_{1,1} * Nds_{1,1}$

# Multiplicação de matrizes: memória compartilhada nos ladrilhos

Matrizes  $M_d$  e  $N_d$  divididas em ladrilhos quadrados de tamanho  $TILE\_WIDTH$ ;  
Produtos escalares entre ladrilhos correspondentes.



# Multiplicação de matrizes: memória compartilhada

## matMul\_gpu\_shared.cu

```
__global__ void matMul(float* Pd, float* Md, float* Nd, int Width) {
    __shared__ float Mds[Tile_Width][Tile_Width];
    __shared__ float Nds[Tile_Width][Tile_Width];

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Identify the row and column of the M element to work on
    int Col = blockIdx.x * Tile_Width + tx;
    int Row = blockIdx.y * Tile_Width + ty;

    float Pvalue = 0;
    // Loop over the N and P tiles required to compute the M element
    for (int m = 0; m < Width/Tile_Width; ++m) {
        // Collaborative loading of N and P tiles into shared memory
        Mds[ty][tx] = Md[Row*Width + (m*Tile_Width + tx)];
        Nds[ty][tx] = Nd[Col + (m*Tile_Width + ty)*Width];
        __syncthreads();

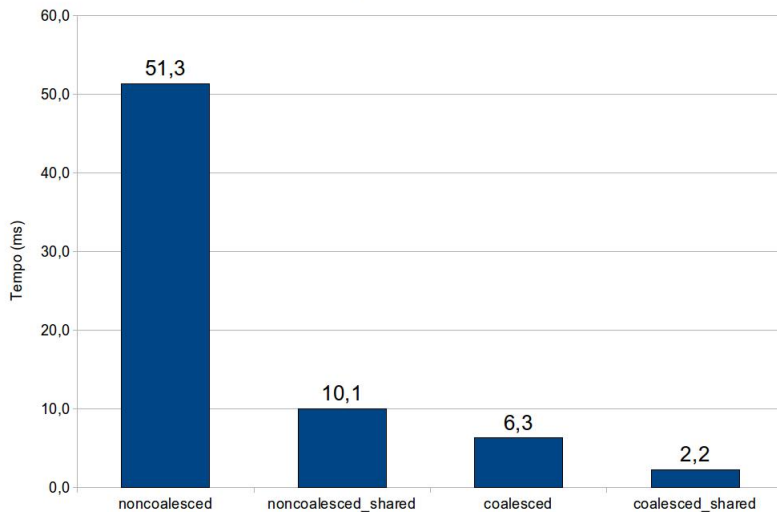
        for (int k = 0; k < Tile_Width; ++k)
```

## Multiplicação de matrizes: memória compartilhada (cont.)

```
    Pvalue += Mds[ty][k] * Nds[k][tx];  
    __syncthreads();  
}  
Pd[Row * Width + Col] = Pvalue;  
}
```

# Multiplicação de Matrizes: N=640

## MULTIPLICAÇÃO DE MATRIZES





# Roteiro

- 1 Módulo 1: Introdução
  - Motivação
  - Primeiros Passos
- 2 Modelo de Paralelismo
  - Módulo 2: Hierarquia, Organização e Identificação
  - Módulo 3: Atribuição
  - Módulo 4: Escalonamento
  - Módulo 5: Hierarquia de Memória
- 3 Módulo 6: Métricas e Otimização de Desempenho
- 4 Módulo 7: Multiplicação de Matrizes
- 5 Considerações Finais

# Considerações Finais

Ocupação dos SMs e uso dos registradores:

- Apresentação de Vasily Volkov no GTC 2010:

<http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf>

Otimizações em CUDA:

- Prof. Fernando Pereira (DCC/UFMG):

<http://homepages.dcc.ufmg.br/~fernando/classes/gpuOpt>

Programação em GPU no Ambiente Google Colaboratory:

- Prof. Ricardo Ferreira (UFV):

<https://github.com/arduinoufv/GPUcolab>

# Considerações Finais

Sites para visitar:

- <https://developer.nvidia.com/accelerated-computing>
- <https://nvidia.qwiklabs.com/>
- <http://hgpu.org>

# Agradecimentos



Laboratório  
Nacional de  
Computação  
Científica





Cook, S. (2012).

*CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.*

Applications of GPU Computing Series. Elsevier Science.



Kirk, D. B. and Hwu, W.-m. W. (2012).

*Programming Massively Parallel Processors: A Hands-on Approach.*

Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.



Sanders, J. and Kandrot, E. (2010).

*CUDA by Example: An Introduction to General-Purpose GPU Programming.*

Addison-Wesley Professional, 1st edition.



Wilt, N. (2013).

*The CUDA Handbook: A Comprehensive Guide to GPU Programming.*

Pearson Education.