

Introdução à Computação Paralela em Sistemas Heterogêneos

Programação para GPUs com a Plataforma CUDA

Prof. Rogério Aparecido Gonçalves¹

¹*Universidade Tecnológica Federal do Paraná (UTFPR)*
Departamento de Computação (DACOM)
Campo Mourão - Paraná - Brasil

rogerioag@utfpr.edu.br

Minicurso

IV Semana de Informática (SEINFO17)

Agenda

- 1 Introdução
- 2 Arquiteturas das GPUs
- 3 Modelo de Programação e Execução
- 4 Memória Mapeada no Host e Memória Unificada (UVA)
- 5 Paralelismo Dinâmico
- 6 Perfilamento e Depuração

Introdução

Material:

Slides: <https://github.com/rogerioag/minicurso-cuda-seinfo17>

Código dos Exemplos

- **Computação Paralela** é uma área da Ciência da Computação que estuda mecanismos e técnicas para a execução paralela de código de aplicações.
- Paralelismo x Concorrência
 - Software
 - Hardware
- **Sistemas Heterogêneos:** Plataformas que possuem Elementos de Processamento diferentes.
- CPUs *multicore* no host + Dispositivos Aceleradores.
- Dispositivos Aceleradores:
 - Placas Gráficas: **GPUs**
 - Arranjos de Coprocessadores: **Xeon Phi**
 - Computação Reconfigurável: **FPGAs**

Aceleradores



Multicore

Host



GPUs



Coprocessadores



FPGAs

Dispositivos Aceleradores

Classificação de Flynn (Flynn 1972) para organização de computadores:

- **SISD (*Single Instruction, Single Data*):** Classe que representa as arquiteturas que trabalham com um único fluxo de instruções e um único fluxo de dados. Não expressam nenhum paralelismo, remete ao *Modelo de von Neumann*.
- **SIMD (*Single Instruction, Multiple Data*):** Máquinas paralelas que executam exatamente o mesmo fluxo de instruções (mesmo programa) em cada uma das suas unidades de execução paralela, considerando fluxos de dados distintos.
- **MIMD (*Multiple Instruction, Multiple Data*):** Máquinas paralelas que executam fluxos independentes e separados de instruções em suas unidades de processamento. Pode-se ter programas diferentes, independentes que processam entradas diferentes, múltiplos fluxos de dados.

Outras classificações considerando-se o nível de abstração foram definidas:

- **SPMD (*Single Program, Multiple Data*):** O modelo SPMD (*Single Program, Multiple Data*) foi definido por (Darema 2001), este modelo é mais geral que o modelo SIMD (*Single Instruction, Multiple Data*) e que o modelo *data-parallel*, pois o SPMD usa aplicações com paralelismo de dados e paralelismo de threads.
- **MSIMD (*Multiple SIMD*):** Máquinas que possuem um memória global de tamanho ilimitado, e P processadores SIMD independentes (Bridges 1990) (Jurkiewicz e Danilewski 2011).
- **SIMT (*Single Instruction, Multiple Threads*):** Modelo utilizado para gerenciar a execução de várias *threads* na arquitetura, todas as *threads* em execução, executam a mesma instrução. Termo utilizado pela NVIDIA para a plataforma CUDA.

Arquiteturas das GPUs

- As GPUs são **Unidades de Processamento Gráfico** (Placas de Vídeo).
- A NVIDIA e a AMD são as principais fabricantes.
 - Podem atuar em conjunto com CPUs Intel ou AMD
 - Conectadas no barramento *PCI Express*
- Paralelismo do tipo SIMD, porém a NVIDIA tem utilizado a denominação SIMT
- Funcionam como dispositivos aceleradores: Programa principal executa em CPU (*host*) e lança a execução de funções **kernel** na GPU (*device*)
- Para a execução de um *kernel* é criado um arranjo de *threads* no qual cada *thread* executa uma cópia do código da função *kernel*
- As GPUs tem sua própria memória organizada em uma hierarquia: Memória Global, Memória Compartilhada...
- As transferências de dados entre a Memória Principal (*host*) e a Memória das GPUs ocorre pelo barramento *PCI Express*.

Arquitetura Kepler I

- Até 15 *Streaming Multiprocessors* (SMX), cada um com 192 núcleos
 - Chegando a 2880 cores
- Introduziu o **Paralelismo Dinâmico**
- Lançamento de execução de *kernels* simultâneos com Hyper-Q



Figura 2: Arquitetura Kepler

Arquitetura Kepler II

- Características de cada SMX:
 - 192 Núcleos de precisão simples
 - 64 Unidades de precisão dupla
 - 32 Unidades de Funções Especiais (SFUs)
 - 32 Unidades de *Load/Store*
 - Escalonamento de 4 warps concorrentes

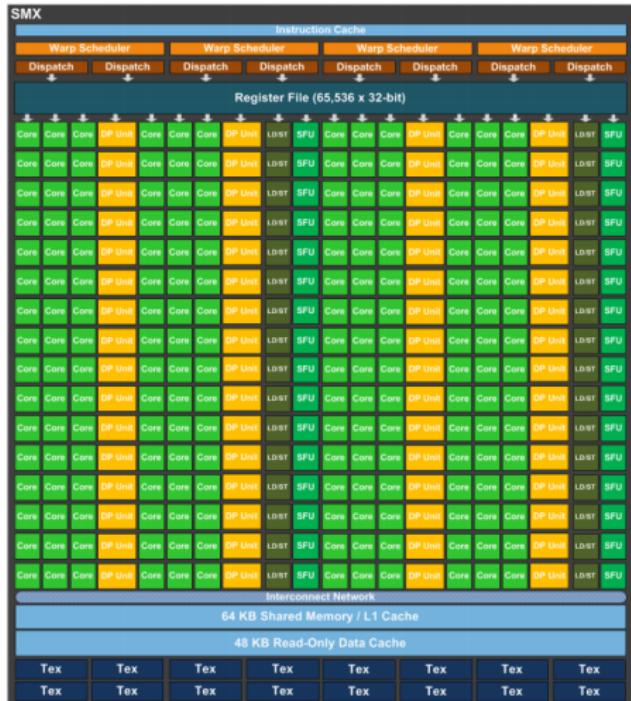


Figura 3: Detalhes do SMX

Arquitetura Pascal I

- Até 60 SMs com 64 cores cada
 - Total de 3840 cores
- 16GB de RAM
- Interconexão NVIDIA NVLink
 - Comunicação entre CPU e GPU, Sistema de Memória e entre GPUs



Figura 4: Arquitetura Pascal

Arquitetura Pascal II

- Cada SM possui:
 - 64 *single precision* FP32 CUDA cores
 - Menos *cores* que as arquiteturas anteriores, porém com maior número de SMs
 - *Threads* compartilham menos recursos como registradores e *cache*
 - Maior concorrência.



Figura 5: Arquitetura Pascal

Comparativo de Recursos

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs¹	5040	6840	10600
Peak FP64 GFLOPs¹	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²
Manufacturing Process	28-nm	28-nm	16-nm FinFET

¹ The GFLOPS in this chart are based on GPU Boost Clocks.

Compute Capabilities I

- A *compute capability* descreve a arquitetura do dispositivo.
 - Número de registradores
 - Quantidade de Memória: (Global, Textura...)
 - Quantidade de *cores* por SM
 - Paralelismo máximo
 - ...

Compute Capabilities II

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

Figura 7: Compute Capabilities (NVIDIA 2016)

Prática: Recuperando Informações do Dispositivo I

- **Exemplo:** `deviceQuery`
- Aplicação exemplo que acompanha o *Software Development Kit (SDK)* do CUDA
- Lista as informações dos dispositivos instalados no *host*

Saída no Terminal

Terminal

```
rogerio@ragserver:~/deviceQuery$ ./deviceQuery
./deviceQuery Starting...
CUDA Device Query (Runtime API) version (CUDART static linking)
Detected 1 CUDA Capable device(s)
Device 0: Tesla K40c
CUDA Driver Version / Runtime Version 7.5 / 7.5
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory: 11520 MBytes (12079136768 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 (xyz) 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
(continua...)
```

Saída no Terminal

Terminal

```
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (xyz): (1024 1024 64)
Max dimension size of a grid size (xyz): (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 PCI Domain ID / Bus ID / location ID: 0 / 1 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device
    simultaneously) >
deviceQuery CUDA Driver = CUDART CUDA Driver Version = 7.5 CUDA Runtime
Version = 7.5 NumDevs = 1 Device0 = Tesla K40c
Result = PASS
```

Modelo de Programação e Execução

Modelo de Programação CUDA

- Plataforma de Computação Paralela **CUDA**
- Permite utilizar a GPU para computação de propósito geral (**GPGPU**)
- O CUDA C/C++ é uma extensão para C/C++ que permite a programação para dispositivos heterogêneos
- API para gerenciamento de dispositivos, memória, transferências entre o *host* e o *device*
- Terminologia:
 - **Host:** CPU e sua Memória Principal (*Host Memory*)
 - **Device:** GPU e sua Memória Global (*Device Memory*)

Modelo Clássico de Execução

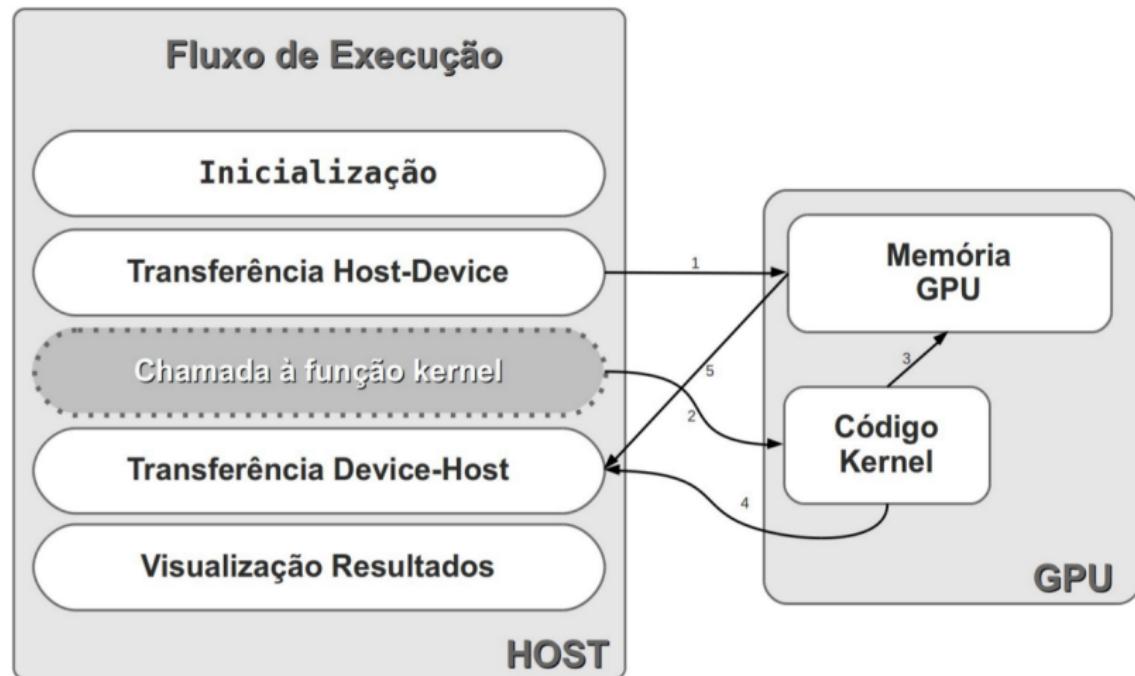


Figura 8: Modelo Clássico de Execução

Fluxo de Execução entre Host e Device

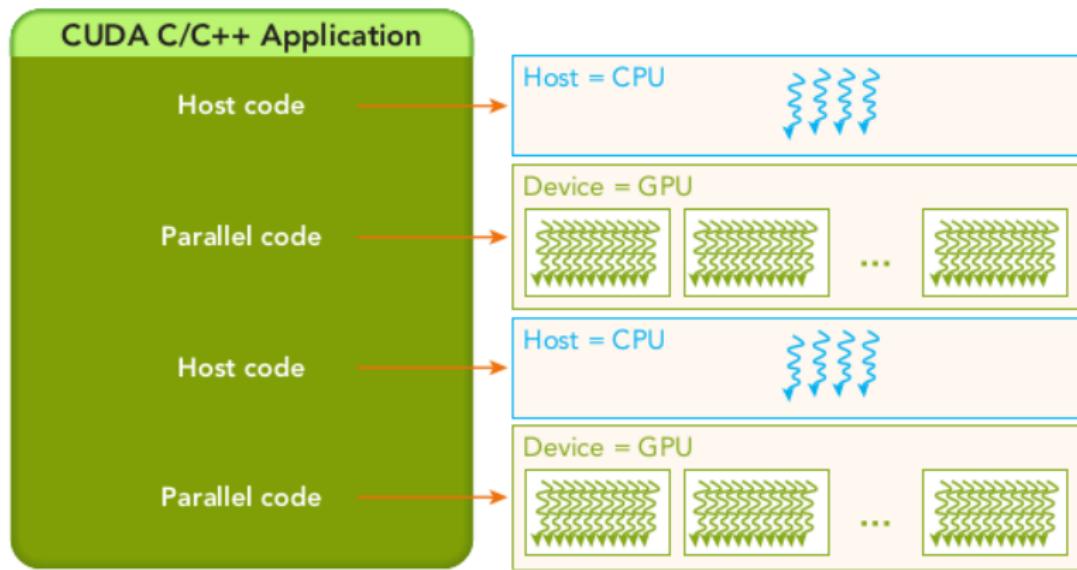


Figura 9: Fluxo de Execução

Plataforma

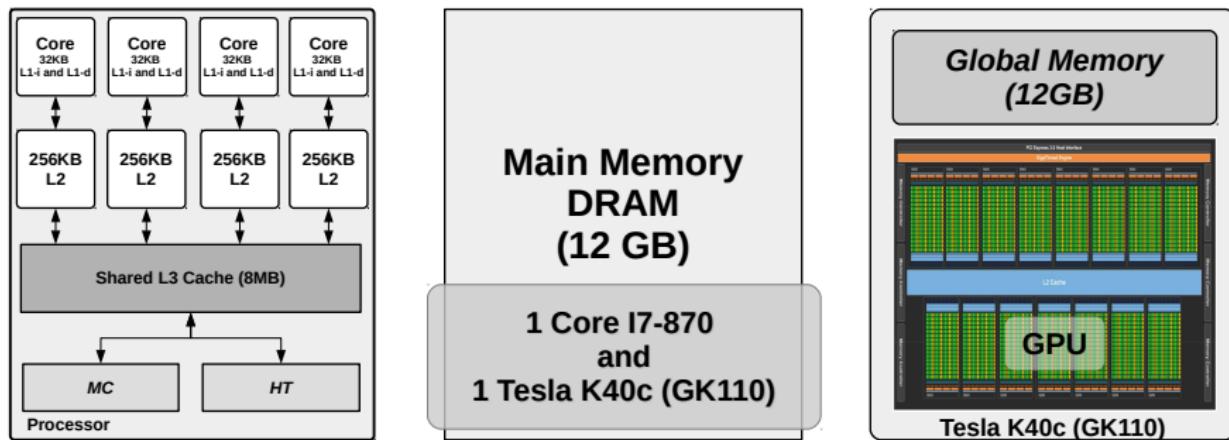


Figura 10: Plataforma i7 com K40c

Hello World!!! I

- **Exemplo:** hello-world-1

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World!!!\n");
6     return 0;
7 }
```

Código 1: Hello World em C

- O compilador NVIDIA nvcc é capaz de compilar código somente para o *host*
- Internamente ao detectar código que será executado no *host* ele chama o compilador do sistema.

Terminal

```
$ nvcc hello-world-1.c -o hello.exe
$ ./hello.exe
Hello World!!!
$
```

Hello World!!! I

- **Exemplo:** hello-world-2

```
1 #include <cuda_runtime.h>
2 #include <stdio.h>
3
4 __global__ void helloKernel()
5 {
6 }
7
8 int main(int argc,
9     char **argv)
10 {
11     helloKernel<<<1,1>>>();
12     printf("Host:
13         Hello World!!!\n");
14
15 }
```

- A chamada à função:

helloKernel<<<1,1>>>() lança a execução do *kernel* na GPU

Terminal

```
$ make
Compiling...
nvcc hello-world-1.c -o hello-world-1.exe
nvcc hello-world-2.cu -o hello-world-2.exe
nvcc hello-world-3.cu -o hello-world-3.exe
$ ./hello-world-2.exe
Host: Hello World!!!
$
```

Código 2: Hello World CUDA

Hello World!!! I

- **Exemplo:** hello-world-3

```
1 #include <cuda_runtime.h>
2 #include <stdio.h>
3
4 __global__ void helloKernel()
5 {
6     int blockIdx = blockIdx.x
7         + blockIdx.y * gridDim.x
8         + blockIdx.z * gridDim.x * gridDim.y;
9
10    int threadIdx = threadIdx.x
11        + threadIdx.y * blockDim.x
12        + threadIdx.z * blockDim.x *
13            blockDim.y
14        + blockIdx * blockDim.x * blockDim.y *
15            blockDim.z;
16
17    printf("GPU: gridDim:(%2d, %2d, %2d)
18          blockDim:(%2d, %2d, %2d)
19          blockIdx:(%2d, %2d, %2d) "
20      "threadIdx:(%2d, %2d, %2d) ->
21          Thread[%2d]: %s\n",
22          gridDim.x, gridDim.y,
23          gridDim.z, blockDim.x, blockDim.y,
24          blockDim.z, blockIdx.x,
25          blockIdx.y,
26          blockIdx.z, threadIdx.x, threadIdx.y,
27          threadIdx.z, threadIdx,
28          "Hello World!!!!\n");
29 }
```

```
1 int main(int argc, char **argv)
2 {
3
4     // Define the GPU id to work.
5     cudaSetDevice(0);
6
7     // Hello from host.
8     printf("Host: Hello World!!!!\n");
9
10    // Hello from GPU.
11    helloKernel<<<1,1>>>();
12
13    // Reset device.
14    cudaDeviceReset();
15
16    return (0);
17 }
```

Código 4: Hello World CUDA

Hello World!!! II

Terminal

```
$ make
Compiling...
nvcc hello-world-1.c -o hello-world-1.exe
nvcc hello-world-2.cu -o hello-world-2.exe
nvcc hello-world-3.cu -o hello-world-3.exe
$ ./hello-world-3.exe
Host: Hello World!!!
GPU: blockDim:( 1 1 1) blockDim:( 1 1 1) blockIdx:( 0 0 0) threadIdx:( 0 0 0)
      -> Thread[ 0]: Hello World!!!
$
```

- Neste caso teremos parte do código executando na CPU e a função *kernel* terá sua execução lançada na GPU.
- O nvcc separa as partes do código que serão executadas no host e no device
 - O código do *kernel* `helloKernel()` será compilado pelo nvcc

- As funções do host como a `main(...)` será compilada pelo compilador do sistema: `gcc`, `clang`, `icc`

Gerenciamento de Memória I

- Do lado `host` temos a Memória Principal da máquina, e a GPU tem uma quantidade de memória separada.
 - Ponteiros de alocações feitas na Memória Principal não são 'desreferenciados' do lado `device` e vice-versa.
- No gerenciamento de memória podemos utilizar as funções de alocação e dealocação existentes em C para o `host`: `malloc()`, `free()` e `memcpy()`
- A API CUDA fornece funções para o gerenciamento de memória do `device`: `cudaMalloc()`, `cudaFree()` e `cudaMemcpy()`
- Alocação de Memória, considerando `size_t bytes = n * sizeof(float)`:

```
1 float * host_pointer = (float *)malloc(bytes);
2 cudaMalloc(&dev_pointer, bytes);
```

- Estruturas alocadas com `malloc` e `cudaMalloc` precisam ser explicitamente copiadas entre as memórias.

Gerenciamento de Memória II

- Transferências **Síncronas** entre as memórias do host e do device são feitas usando:
 - `cudaMemcpy(dev_pointer, host_pointer, num_bytes,
cudaMemcpyHostToDevice)`
 - `cudaMemcpy(host_pointer, device_pointer, bytes,
cudaMemcpyDeviceToHost)`
- Bloqueiam a CPU até que a cópia esteja completa.
- Transferências **Assíncronas** entre as memórias do host e do device são feitas usando:
 - `cudaMemcpyAsync(dev_pointer, host_pointer, num_bytes,
cudaMemcpyHostToDevice)`
 - `cudaMemcpyAsync(host_pointer, device_pointer, bytes,
cudaMemcpyDeviceToHost)`
- Não bloqueiam a CPU.

Organização das Threads I

- Para o lançamento da execução de um determinado *kernel* o programador deve especificar a configuração do arranjo de *threads* que será criado para a execução.
- A configuração é especificada na ativação da função *kernel* entre `<<< #Blocos, #Threads >>>` e cada uma das $\#Blocos \times \#Threads$ *threads* irá executar uma cópia do código do *kernel*.
- A especificação da configuração do arranjo de *threads* pode variar conforme o domínio do problema.

```
1 // Chamada a função kernel.  
2 funcKernel<<<N, 1>>>(parametros);
```

Código 5: Chamada com N blocos de 1 thread cada

```
1 // Chamada a função kernel.  
2 funcKernel<<<1, N>>>(parametros);
```

Código 6: Chamada com 1 bloco de N threads cada

Exemplo Soma de Vetores usando Blocos

- A ativação de **vecAdd** cria **blocos** (block)
- O conjunto de **blocos** é chamado de **grade** (grid)
- O índice de bloco **blockIdx.x** é usado para referenciar os elementos dos **arrays**

```
1 __global__ void vecAdd(float *a, float *b,
2                         float *c, int n)
3 {
4     int id = blockIdx.x;
5     if (id < n)
6         c[id] = a[id] + b[id];
}
```

Código 7: kernel vecAdd utilizando o índice de bloco

```
1 size_t bytes = n * sizeof(float);
2
3 // Alocacao de memoria para host e device.
4 h_a = (float *)malloc(bytes);
5 h_b = (float *)malloc(bytes);
6 h_c = (float *)malloc(bytes);
7 cudaMalloc(&d_a, bytes);
8 cudaMalloc(&d_b, bytes);
9 cudaMalloc(&d_c, bytes);
10
11 init_array();
12
13 // Copia dos arrays para a GPU.
14 cudaMemcpy(d_a, h_a, bytes,
15             cudaMemcpyHostToDevice);
15 cudaMemcpy(d_b, h_b, bytes,
16             cudaMemcpyHostToDevice);
16
17 vecAdd <<<n,1>>> (d_a, d_b, d_c, n);
18
19 // Copia do resultado.
20 cudaMemcpy(h_c, d_c, bytes,
21             cudaMemcpyDeviceToHost);
21
22 print_array(); check_result();
23
24 // Dealocacao de Memoria.
25 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
26 free(h_a); free(h_b); free(h_c);
```

Código 8: Chamada com N blocos de

Exemplo Soma de Vetores usando Threads

- A ativação de **vecAdd** cria **threads**
- As **threads** são agrupadas em blocos
- O índice da *thread* **threadIdx.x** é usado para referenciar os elementos dos *arrays*

```
1 __global__ void vecAdd(float *a, float *b,
2                         float *c, int n)
3 {
4     int id = threadIdx.x;
5     if (id < n)
6         c[id] = a[id] + b[id];
}
```

Código 9: kernel vecAdd utilizando o índice das threads

```
1 size_t bytes = n * sizeof(float);
2
3 // Alocacao de memoria para host e device.
4 h_a = (float *)malloc(bytes);
5 h_b = (float *)malloc(bytes);
6 h_c = (float *)malloc(bytes);
7 cudaMalloc(&d_a, bytes);
8 cudaMalloc(&d_b, bytes);
9 cudaMalloc(&d_c, bytes);
10
11 init_array();
12
13 // Copia dos arrays para a GPU.
14 cudaMemcpy(d_a, h_a, bytes,
15            cudaMemcpyHostToDevice);
15 cudaMemcpy(d_b, h_b, bytes,
16            cudaMemcpyHostToDevice);
16
17 vecAdd <<<1,n>>> (d_a, d_b, d_c, n);
18
19 // Copia do resultado.
20 cudaMemcpy(h_c, d_c, bytes,
21            cudaMemcpyDeviceToHost);
21
22 print_array(); check_result();
23
24 // Dealocacao de Memoria.
25 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
26 free(h_a); free(h_b); free(h_c);
```

Código 10: Chamada com N blocos

Organização das Threads I

- Ao lançar a execução de um *kernel* é criado um arranjo de *threads* organizado em um **grid** de **blocos** de **threads**.
- **Variáveis embutidas:**

- `threadIdx`
- `blockIdx`
- `blockDim`
- `gridDim`

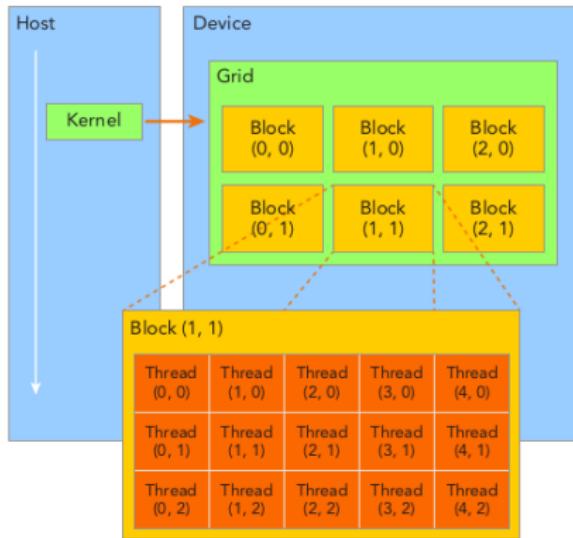


Figura 11: Organização do arranjo de threads

Exemplo de arranjo (4x4)

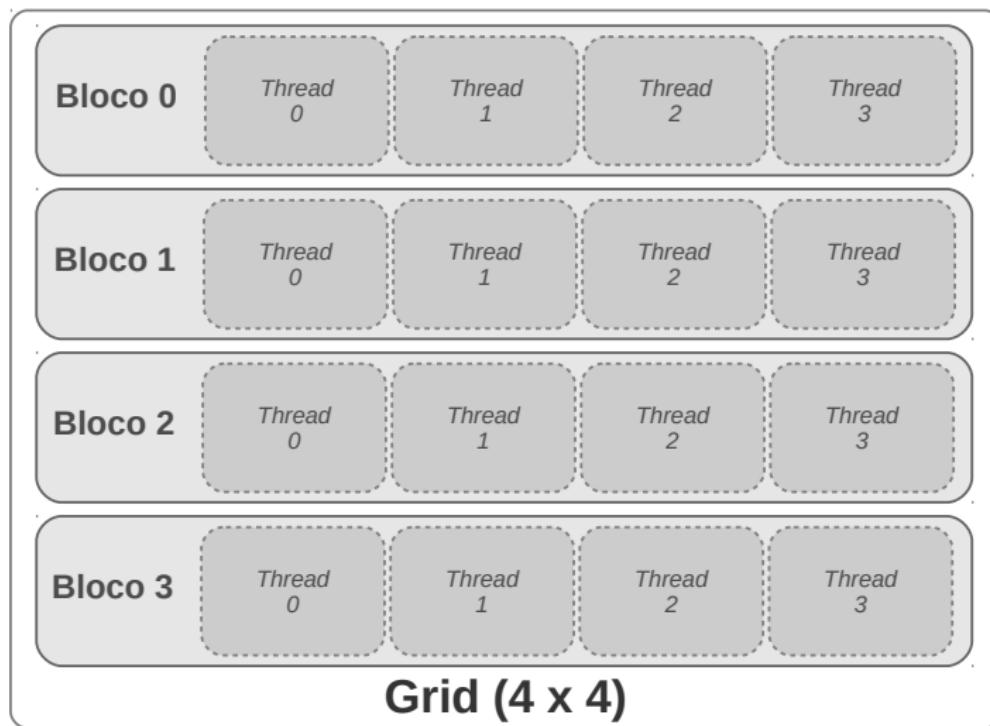


Figura 12: Arranjo (4x4)

Tradução do **id** das *threads* para o arranjo (4x4)

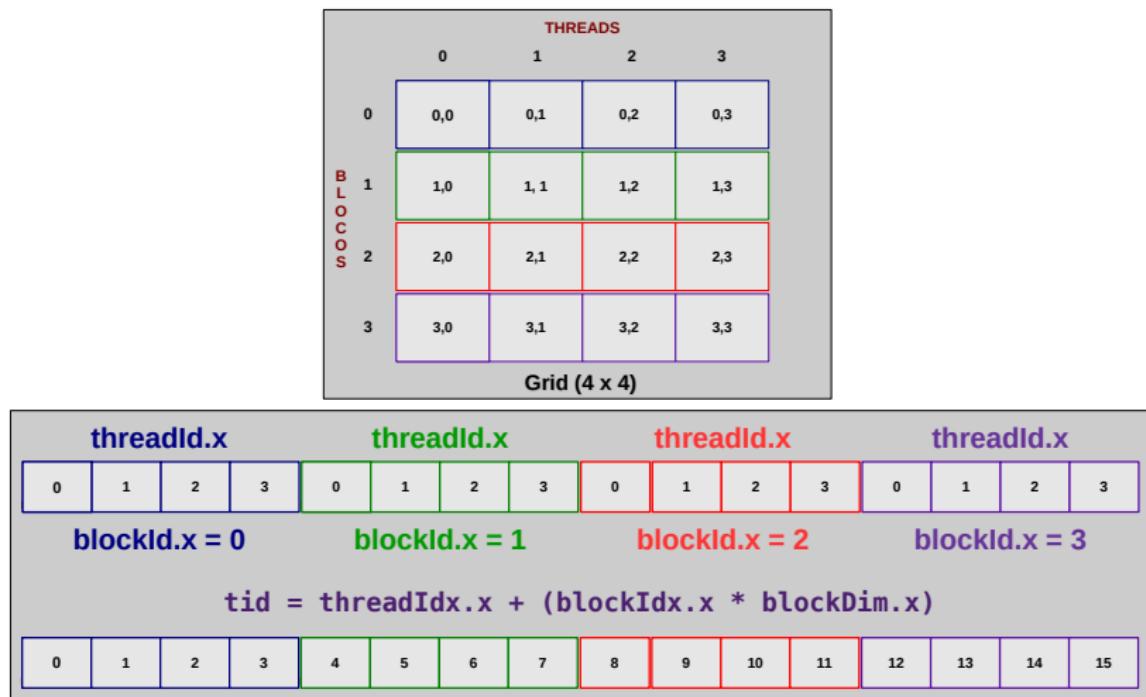


Figura 13: Tradução do **id** das *threads*

Mapeamento para o **Hardware**

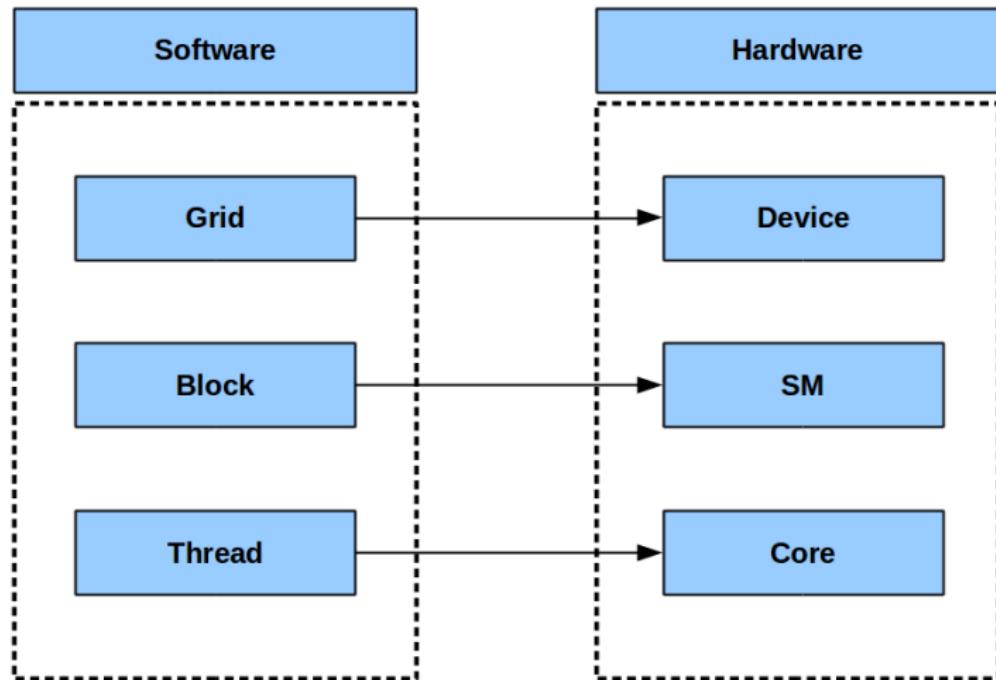


Figura 14: Mapeamento para o Hardware

Visão Lógica e Visão do Hardware

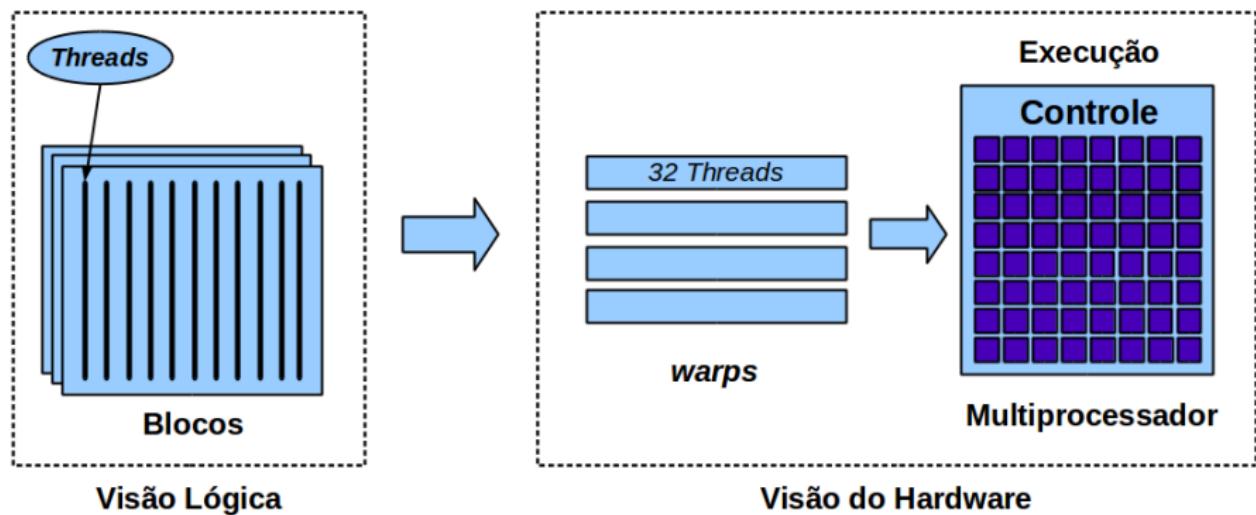


Figura 15: Mapeamento para o Hardware

Como definir a configuração do arranjo de **threads** I

- A definição do arranjo pode acompanhar a complexidade do problema que está sendo resolvido.
- Em algoritmos que trabalham sobre blocos de dados é possível configurar o arranjo de *threads* para que cada uma trabalhe sobre um desses blocos.

```
1 int blockSize, gridSize;
2
3 // Numero de threads em cada bloco de threads.
4 blockSize = 1024;
5
6 // Numero de blocos de threads no grid.
7 gridSize = (int)ceil((float)n/blockSize);
8
9 // Chamada a funcao kernel.
10 funcKernel<<<gridSize, blockSize>>>(parametros);
```

Código 11: Chamada com $n/1024$ blocos de 1024 threads cada

Trabalhando com tamanhos de dados arbitrários

- Pode ser que o problema não seja múltiplo da dimensão de blocos
`blockDim.x`
- Para evitar acessos a posições fora do espaço dos arrays

```
1 int blockSize, gridSize;
2
3 __global__ void vecAdd(float *a, float *b, float *c, int n)
4 {
5     // Get our global thread ID
6     int id = blockIdx.x * blockDim.x + threadIdx.x;
7     if (id < n)
8         c[id] = a[id] + b[id];
9 }
```

Código 12: Chamada com $n/1024$ blocos de 1024 threads cada

```
1 // Chamada a função kernel.
2 vecAdd<<<(N + M+1) / M, M>>>(d_a, d_b, d_c, N);
```

Código 13: Chamada com ajuste para tamanho arbitrário

Dimensões de *Kernels*

- A configuração de **grid** e **bloco** pode ir além do que vimos até agora.
- É uma combinação de 6 dimensões. `grid(x,y,z)` e `bloco(x,y,z)`

```
1 /* Definicao do arranjo de threads em blocos do grid. */
2 dim3 grid(32, 64, 16);
3 dim3 block(32, 32, 1);
4
5 /* Chamada do kernel */
6 myKernel<<<grid, block>>>(parametros);
```

Código 14: Chamada usando mais dimensões

Cálculo das Dimensões de *Kernels*: Restrições I

- O total de *threads* criadas em uma configuração será $gx \times gy \times gz \times bx \times by \times bz$.
- A multiplicação das dimensões dos blocos não pode ultrapassar 512 ou 1024 dependendo da arquitetura. Este valor é o número máximo de *threads* por blocos ($bx \times by \times bz \leq 1024$).
- A multiplicação das dimensões do bloco ($bx \times by \times bz$) tem que ser divisível por 32, pois as *threads* são organizadas durante a execução em grupos de 32, em *warps*.
 - Problema de **kernels divergentes**: *threads* (*lanes*) inativos dentro do *warp*, degradação do desempenho.
- O total de *threads* criadas em uma configuração deve ser menor que o tamanho de N , esta condição se dá pelo fato que as *threads* não podem realizar trabalho com espaços de memórias não alocados.

Exemplo: checkDimensions

- Imprime o valor das variáveis embutidas:

```
1 __global__ void checkIndex(int funcId)
2 {
3     printf("gridDim:(%2d, %2d, %2d) blockDim:(%2d, %2d, %2d)
4         blockIdx:(%2d, %2d, %2d) "
5         "threadIdx:(%2d, %2d, %2d) -> id: %2d\n",
6             gridDim.x, gridDim.y,
7             gridDim.z, blockDim.x, blockDim.y, blockDim.z, blockIdx.x,
8             blockIdx.y, blockIdx.z, threadIdx.x, threadIdx.y,
9             threadIdx.z, getGlobalIdFunc[funcId]());
10 }
```

Código 15: Imprimindo variáveis embutidas

```
1 dim3 grid(gx, gy, gz);
2 dim3 block(bx, by, bz);
3
4 int funcId = calculateFunctionId(grid, block);
5 cudaSetDevice(gpuId);
6
7 checkIndex<<<grid, block>>>(funcId);
```

Cálculo das Dimensões de Kernels

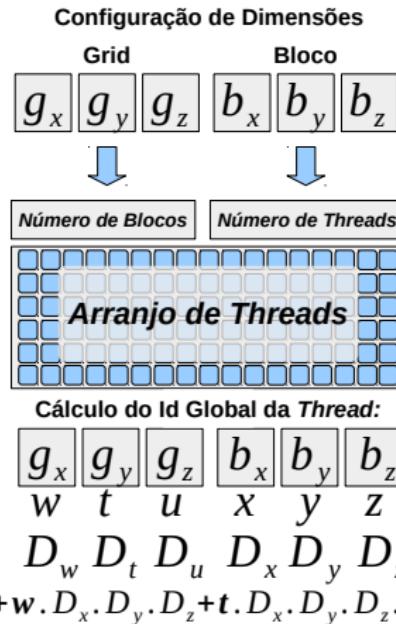


Figura 16: Cálculo das Dimensões

Exemplo: sincos

- **Exemplo:** sincos
- Cálculo de uma tabela de *seno e cosseno*.

```
1 void sincos_function_(DATA_TYPE* x, DATA_TYPE* y, DATA_TYPE* xy,
2   int nx, int ny, int nz)
3 {
4     int i, j, k, indice;
5     for (i = 0; i < nx; ++i)
6     {
7         for (j = 0; j < ny; ++j)
8         {
9             for (k = 0; k < nz; ++k)
10            {
11                indice = (i * ny * nz) + (j * nz) + k;
12                xy[indice] = sin(x[indice]) + cos(y[indice]);
13            }
14        }
15    }
```

Código 17: Código para CPU

Exemplo: sincos |

- **Exemplo:** sincos
- O código com os mesmos laços aninhados.

```
1 --global__ void sincos_kernel_3(DATA_TYPE* x, DATA_TYPE* y, DATA_TYPE*
2 xy, int nx, int ny, int nz)
3 {
4     int i, j, k, indice;
5     for (i = 0; i < nx; ++i)
6     {
7         for (j = 0; j < ny; ++j)
8         {
9             for (k = 0; k < nz; ++k)
10            {
11                indice = (i * ny * nz) + (j * nz) + k;
12                xy[indice] = sin(x[indice]) + cos(y[indice]);
13            }
14        }
15    }
```

Código 18: Versão do kernel com três laços

Exemplo: sincos II

- Apenas uma *thread* deve executá-lo.

```
1 dim3 grid(1,1,1);
2 dim3 block(1,1,1);
3 sincos_kernel_3<<<grid, block>>>(d_x, d_y, d_xy, nx, ny, nz, funcId);
```

Código 19: Chamada do kernel

Exemplo: sincos |

- **Exemplo:** sincos
- Migrando iterações dos laços aninhados para *threads* do arranjo
- nx iterações foram transferidas para as dimensões de *grid* e blocos de *threads*. O i ($0 \leq i < nx$), é obtido com o *id* das *threads*

```
1 __global__ void sincos_kernel_2(DATA_TYPE* x, DATA_TYPE* y, DATA_TYPE*
2     xy, int nx, int ny, int nz, int funcId)
3 {
4     int i, j, k, indice;
5     i = getGlobalIdFunc[funcId]();
6     for (j = 0; j < ny; ++j)
7     {
8         for (k = 0; k < nz; ++k)
9         {
10             indice = (i * ny * nz) + (j * nz) + k;
11             xy[indice] = sin(x[indice]) + cos(y[indice]);
12         }
13     }
}
```

Código 20: Versão do kernel com dois laços

Exemplo: sincos II

- As configurações das dimensões de **grid** e **blocos** pode ser qualquer combinação que seja igual a *nx*, desde que respeite as restrições.

```
1 sincos_kernel_2<<<grid, block>>>(d_x, d_y, d_xy, nx, ny, nz, funcId);
```

Código 21: Chamada do kernel

Exemplo: sincos |

- **Exemplo:** sincos
- Migrando iterações dos laços aninhados para *threads* do arranjo.
- $(nx * ny)$ iterações foram migradas.
- Cada thread do arranjo irá executar o laço mais interno somente, o que resulta em nz iterações.
- O índice pode ser calculado como $indice = (i * nz) + k$

```
1 __global__ void sincos_kernel_1(DATA_TYPE* x, DATA_TYPE* y, DATA_TYPE*
2     xy, int nx, int ny, int nz, int funcId)
3 {
4     int i, k, indice;
5     i = getGlobalIdFunc[funcId]();
6     for (k = 0; k < nz; k++)
7     {
8         indice = (i * nz) + k;
9         xy[indice] = sin(x[indice]) + cos(y[indice]);
10    }
11 }
```

Código 22: Versão do kernel com um laço

Exemplo: sincos II

- As configurações das dimensões de **grid** e **blocos** pode ser uma combinação que seja igual a $(nx * ny)$, desde que respeite as restrições.

```
1 sincos_kernel_1<<<grid, block>>>(d_x, d_y, d_xy, nx, ny, nz, funcId);
```

Código 23: Chamada do kernel

Exemplo: sincos |

- **Exemplo:** sincos
- Migrando iterações dos laços aninhados para *threads* do arranjo.
- Todas as ($nx * ny * nz$) iterações foram transferidas para as dimensões do arranjo de threads.
- O índice será calculado com base nas dimensões do arranjo de *threads*.

```
1 __global__ void sincos_kernel_0(DATA_TYPE* x, DATA_TYPE* y, DATA_TYPE*
2     xy, int nx, int ny, int nz, int funcId)
3 {
4     int indice;
5     indice = getGlobalIdFunc[funcId]();
6     xy[indice] = sin(x[indice]) + cos(y[indice]);
```

Código 24: Versão do kernel sem laços

Exemplo: sincos II

- As configurações das dimensões de **grid** e **blocos** pode ser uma combinação que seja igual a $(nx * ny * nz)$, desde que respeite as restrições.

```
1 sincos_kernel_0<<<grid, block>>>(d_x, d_y, d_xy, nx, ny, nz, funcId);
```

Código 25: Chamada do kernel

Funções para o cálculo do *id* global |

- Função para configurações: $gx > 1 \rightarrow (32, 1, 1)(1, 1, 1)$

```
1 __device__ int getGlobalIdx_grid_1D_x()
2 {
3     return blockIdx.x;
4 }
```

Código 26: Funções para o cálculo do id das threads

- Função para configurações: $bx > 1 \rightarrow (1, 1, 1)(32, 1, 1)$

```
1 __device__ int getGlobalIdx_block_1D_x()
2 {
3     return threadIdx.x;
4 }
```

Código 27: Funções para o cálculo do id das threads

Funções para o cálculo do *id* global II

- Função para configurações 3D-3D, mapeando 6 dimensões.

- gx, gy, gz e $bx, by, bz > 1 \rightarrow (32, 32, 32)(2, 16, 32)$

- Quantidade de Operações: mult: 9 add: 5 (14 operações).

$$\begin{aligned} id = & x + y \times Dx + z \times Dx \times Dy + w \times Dx \times Dy \times Dz + \\ & t \times Dx \times Dy \times Dz \times Dw + u \times Dx \times Dy \times Dz \times Dw \times Dt \end{aligned}$$

```
1 __device__ int getGlobalIdx_grid_3D_xyz_block_3D_xyz()
2 {
3     int blockIdx = blockIdx.x + blockIdx.y * gridDim.x + blockIdx.z *
4         gridDim.x * gridDim.y;
5     int threadId = threadIdx.x + threadIdx.y * blockDim.x +
6         threadIdx.z * blockDim.x * blockDim.y + blockIdx * blockDim.x *
7         blockDim.y * blockDim.z;
8     return threadId;
9 }
```

Código 28: Funções para o cálculo do *id* das threads

Memória Mapeada no Host e Memória Unificada (UVA)

Memória Mapeada no Host I

- Faz o mapeamento da memória do host, a GPU acesse a memória do host
- As alocações devem ser feitas utilizando a função:

```
1 cudaSetDeviceFlags(cudaDeviceMapHost);  
2  
3 cudaHostAlloc(ponteiro, bytes, cudaHostAllocMapped);  
4  
5 /* Vincula a um ponteiro do device */  
6 cudaHostGetDevicePointer(ponteiro_device, ponteiro_host, 0);
```

Código 29: Funções para o cálculo do id das threads

Memória Mapeada no Host II

```
1 size_t bytes = n * sizeof(float);
2
3 // Set the flag in order to allocate pinned host memory that is accessible to the device.
4 cudaSetDeviceFlags(cudaDeviceMapHost);
5
6 // Allocate memory for each vector on host
7 cudaHostAlloc((void **)&h_a, bytes, cudaHostAllocMapped);
8 cudaHostAlloc((void **)&h_b, bytes, cudaHostAllocMapped);
9 cudaHostAlloc((void **)&h_c, bytes, cudaHostAllocMapped);
10
11 init_array();
12
13 cudaHostGetDevicePointer(&d_a, h_a, 0);
14 cudaHostGetDevicePointer(&d_b, h_b, 0);
15 cudaHostGetDevicePointer(&d_c, h_c, 0);
16
17 // Number of threads in each thread block.
18 int threadsPerBlock = 256;
19 // Number of thread blocks in grid.
20 int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
21
22 /* Execute the kernel. */
23 vecAdd <<< blocksPerGrid, threadsPerBlock >>> (d_a, d_b, d_c, n);
24
25 /* Synchronize */
26 cudaDeviceSynchronize();
27
28 print_array();
29 check_result();
30
```

Memória Mapeada no Host III

```
31 // Release Memory,  
32 cudaFreeHost(h_c);  
33 cudaFreeHost(h_b);  
34 cudaFreeHost(h_a);  
35 cudaDeviceReset();
```

Código 30: Utilizando Memória Mapeada

- Cria um espaço de endereçamento único acessível pela CPU e pela GPU
- As alocações devem ser feitas utilizando a função:

```
1 cudaMallocManaged(ponteiro, bytes);
```

Código 31: Funções para o cálculo do id das threads

Memória Unificada (UVA) II

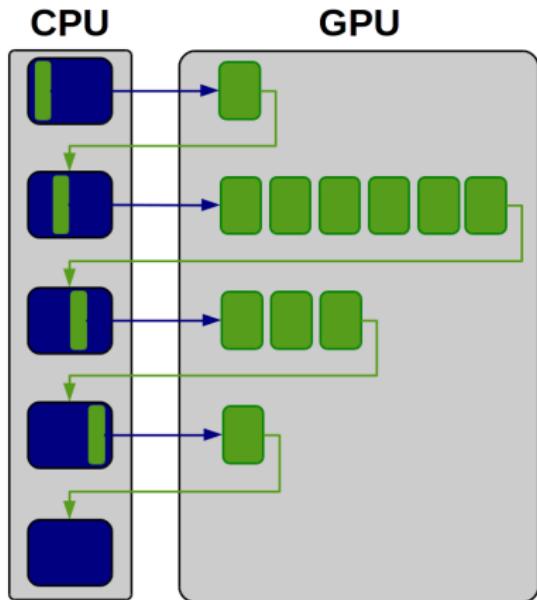
```
1 // Size, in bytes, of each vector
2 size_t bytes = n * sizeof(float);
3 // Allocate memory for each vector on host
4 cudaMallocManaged(&uva_a, bytes);
5 cudaMallocManaged(&uva_b, bytes);
6 cudaMallocManaged(&uva_c, bytes);
7
8 init_array();
9
10 // Number of threads in each thread block.
11 int threadsPerBlock = 256;
12 // Number of thread blocks in grid.
13 int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
14
15 /* Execute the kernel. */
16 vecAdd <<< blocksPerGrid, threadsPerBlock >>> (uva_a, uva_b, uva_c, n);
17
18 /* Synchronize */
19 cudaDeviceSynchronize();
20 print_array();
21 check_result();
22
23 // Release device memory
24 cudaFree(uva_a);
25 cudaFree(uva_b);
26 cudaFree(uva_c);
```

Código 32: Utilizando UVA

Paralelismo Dinâmico

Paralelismo Dinâmico

GPU como um coprocessador



GPU com autonomia

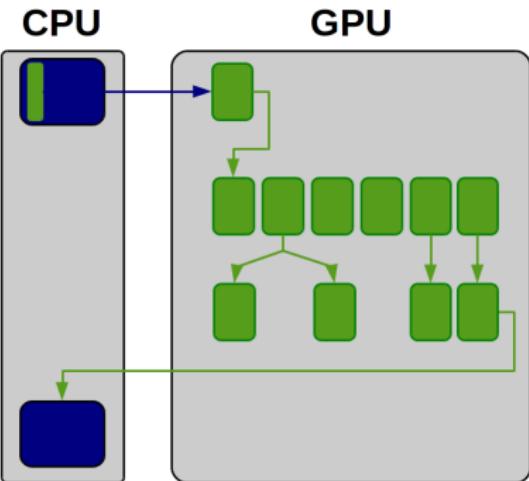


Figura 17: Paralelismo Dinâmico

Paralelismo Dinâmico

```
1 __global__ SubKernel(void* dados)
2 {
3     /* Operacoes */
4 }
5 __global__ MainKernel(void *dados)
6 {
7     if (threadIdx.x == 0)
8     {
9         SubKernel<<<1, 32>>>(dados);
10        cudaThreadSynchronize();
11    }
12    __syncthreads();
13    /* Operacoes do MainKernel */
14 }
15
16 int main()
17 {
18     /* Chamada ao MainKernel no codigo principal no host */
19     MainKernel<<<8, 32>>>(dados);
20     /*...*/
21 }
```

Código 33: Funções para o cálculo do id das threads

Perfilamento e Depuração

NVIDIA Profiler: **nvprof** |

- O **nvprof** é uma ferramenta fornecida com o **CUDA**.
- Pode ser usada para recuperar informações de perfilamento da execução
- Tutorial da NVIDIA

Terminal

```
$ nvprof ./vectoradd.exe 1024
```

NVIDIA Profiler: nvprof II

```
1 ==5622== Profiling application: ./vectoradd.exe 1024
2 ==5622== Profiling result:
3 Time(%) Time Calls Avg Min Max Name
4 36.56% 3.7440us 1 3.7440us 3.7440us 3.7440us [CUDA memcpy DtoH]
5 36.56% 3.7440us 2 1.8720us 1.7280us 2.0160us [CUDA memcpy HtoD]
6 26.88% 2.7520us 1 2.7520us 2.7520us 2.7520us vecAdd(float*, float*, float*, int)
7 ==5622== API calls:
8 Time(%) Time Calls Avg Min Max Name
9 99.57% 139.94ms 3 46.647ms 4.6950us 139.93ms cudaMalloc
10 0.16% 229.49us 83 2.7640us 129ns 99.624us cuDeviceGetAttribute
11 0.08% 112.30us 3 37.432us 4.9860us 97.222us cudaFree
12 0.06% 89.495us 1 89.495us 89.495us 89.495us cudaEventSynchronize
13 0.04% 53.484us 3 17.828us 9.6360us 27.668us cudaMemcpy
14 0.02% 31.793us 1 31.793us 31.793us 31.793us cuDeviceTotalMem
15 0.02% 28.349us 1 28.349us 28.349us 28.349us cuDeviceGetName
16 0.01% 19.696us 1 19.696us 19.696us 19.696us cudaLaunch
17 0.01% 8.8800us 1 8.8800us 8.8800us 8.8800us cudaEventElapsed
18 0.01% 8.1360us 1 8.1360us 8.1360us 8.1360us cudaDeviceSynchronize
19 0.00% 6.1250us 2 3.0620us 2.4640us 3.6610us cudaEventRecord
20 0.00% 3.6420us 4 910ns 163ns 3.0160us cudaSetupArgument
21 0.00% 3.4340us 2 1.7170us 764ns 2.6700us cudaEventCreateWithFlags
22 0.00% 1.4810us 2 740ns 582ns 899ns cuDeviceGetCount
23 0.00% 1.0700us 1 1.0700us 1.0700us 1.0700us cudaConfigureCall
24 0.00% 543ns 2 271ns 209ns 334ns cuDeviceGet
```

Código 34: Saída do nvprof

NVIDIA Profiler: nvprof III

- Consulta a Eventos e Métricas Disponíveis:

Terminal

```
$ nvprof --query-events
```

Terminal

```
$ nvprof --query-metrics
```

NVIDIA Profiler: nvprof IV

- Número de **warps** lançados em um SM.

Terminal

```
$ nvprof --events warps_launched ./vectoradd.exe 32
```

```
1 /* ... Imprime a saída do programa */
2 ==5756== NVPROF is profiling process 5756, command: ./vectoradd.exe 32
3 /* ... Imprime a saída do programa */
4 ==5756== Profiling application: ./vectoradd.exe 32
5 ==5756== Profiling result:
6 ==5756== Event result:
7 Invocations Event Name Min Max Avg
8 Device "Tesla K40c (0)"
9     Kernel: vecAdd(float*, float*, float*, int)
10        1 warps_launched 8 8 8
```

Código 35: Saída do nvprof

- Número de instruções executadas, eficiência dos SMs e instruções executadas por ciclo.

Terminal

```
$ nvprof --metrics inst_executedsm_efficiencyipc ./vectoradd.exe 32
```

NVIDIA Profiler: nvprof VI

```
1 /* ... Imprime a saída do programa */
2 ==5817== NVPROF is profiling process 5817, command: ./vectoradd.exe 32
3 /* ... Imprime a saída do programa */
4 ==5817== Profiling application: ./vectoradd.exe 32
5 ==5817== Profiling result:
6 ==5817== Metric result:
7 Invocations Metric Name Metric Description Min Max Avg
8 Device "Tesla K40c (0)"
9     Kernel: vecAdd(float*, float*, float*, int)
10        1 inst_executed Instructions Executed 75 75 75
11        1 sm_efficiency Multiprocessor Activity 2.02% 2.02% 2.02%
12        1 ipc Executed IPC 0.041028 0.041028 0.041028
```

Código 36: Saída do nvprof

Tratamento de Erros

- As chamadas para as funções da **API CUDA** retornam um código de erro (`cudaError_t`)
- Erro na chamada corrente ou de uma operação assíncrona anterior.
- O código do erro pode ser recuperado com a função:

```
1 cudaError_t cudaGetLastError(void);
```

Código 37: Saída do nvprof

- Uma *string* que descreve o erro pode ser recuperada com a função:

```
1 char * cudaGetLastErrorString(cudaError_t);
```

Código 38: Saída do nvprof

Contato

- Prof. Rogério Aparecido Gonçalves
- **E-mail:** rogerioag@utfpr.edu.br

Informações

O material desse minicurso foi preparado no âmbito do projeto de extensão “***Escola de Computação Paralela***”.

Material: <https://github.com/rogerioag/minicurso-cuda-seinfo17>

Agradecimentos

Os autores agradecem à NVIDIA pela doação de uma **GPU Titan X Pascal** através do GPU Grant Program para o projeto **Estudo Exploratório sobre Técnicas e Mecanismos para Paralelização Automática e Offloading de Código em Sistemas Heterogêneos** (UTFPR 916/2017).

Referências I

- Bridges, T. 1990. «The GPA machine: a generally partitionable MSIMD architecture». Em *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, 196–203.
- Darema, Frederica. 2001. «The SPMD Model: Past, Present and Future». Em *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK: Springer-Verlag, 1.
- Denise Stringhini, Alfredo Goldman, Rogério Aparecido Gonçalves. 2012. «Introdução à Computação Heterogênea». Em *XXXI Jornadas de Atualização em Informática (JAI)*, 1.º ser., ed. Renata Galante Luiz Carlos Albini Alberto Ferreira de Souza. SBC, 262–309.
http://www.imago.ufpr.br/csbc2012/anais_csbc/eventos/jai/index.html.
- Flynn, M. 1972. «Some Computer Organizations and Their Effectiveness». *IEEE Trans. Comput.* C-21: 948+. http://en.wikipedia.org/wiki/Flynn's_taxonomy.
- Jurkiewicz, Tomasz, e Piotr Danilewski. 2011. «Efficient Quicksort and 2D Convex Hull for CUDA, and MSIMD as a Realistic Model of Massively Parallel Computations.»
- NVIDIA. 2016. *Whitepaper: NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built - Featuring Pascal GP100, the World's Fastest GPU*. NVIDIA Corporation.
<https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>.