

Introdução à Computação Paralela em Sistemas Heterogêneos

Programação para GPUs com a Plataforma CUDA

Prof. Rogério Aparecido Gonçalves¹
rogerioag@utfpr.edu.br

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

Semana de Informática (SEINFO 2025)
SEINFO25
Minicurso



Agenda i

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
7. Referências

Introdução

Material do Minicurso

Material:

Slides:

<https://github.com/rogerioag/minicurso-cuda-seinfo25>

Código dos Exemplos

Computação Paralela e Sistemas Heterogêneos

- **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

Figura 1: Aceleradores

Classificação de 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.

Modelos de Computação iii

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 and 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
CPU Die Size	551 mm ²	601 mm ²	610 mm ²

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

```
~/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 (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
```

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 (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 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 i

- 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

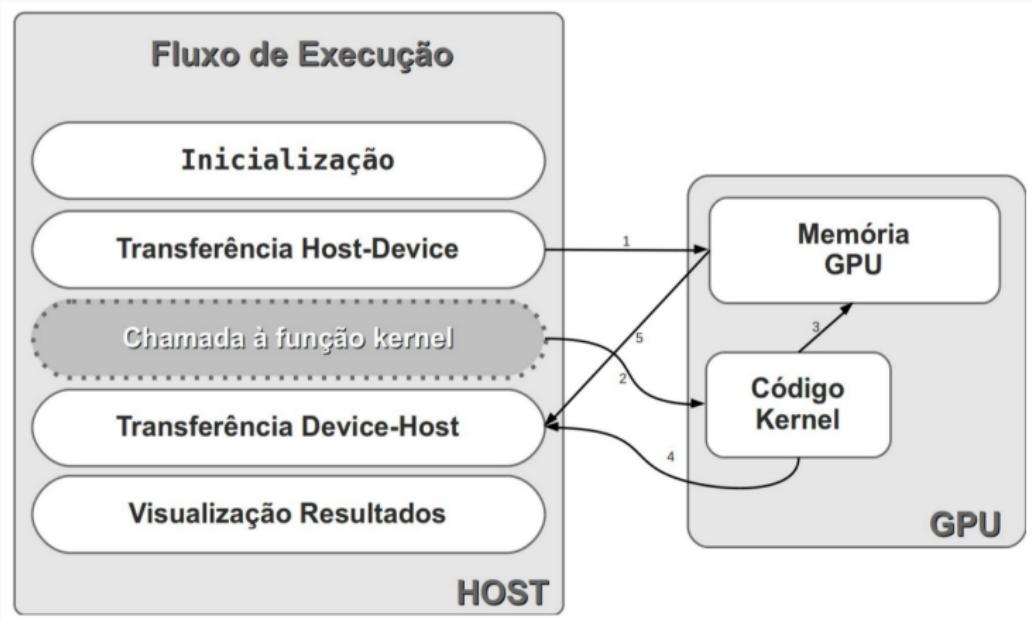


Figura 8: Modelo Clássico de Execução

Fluxo de Execução entre Host e Device

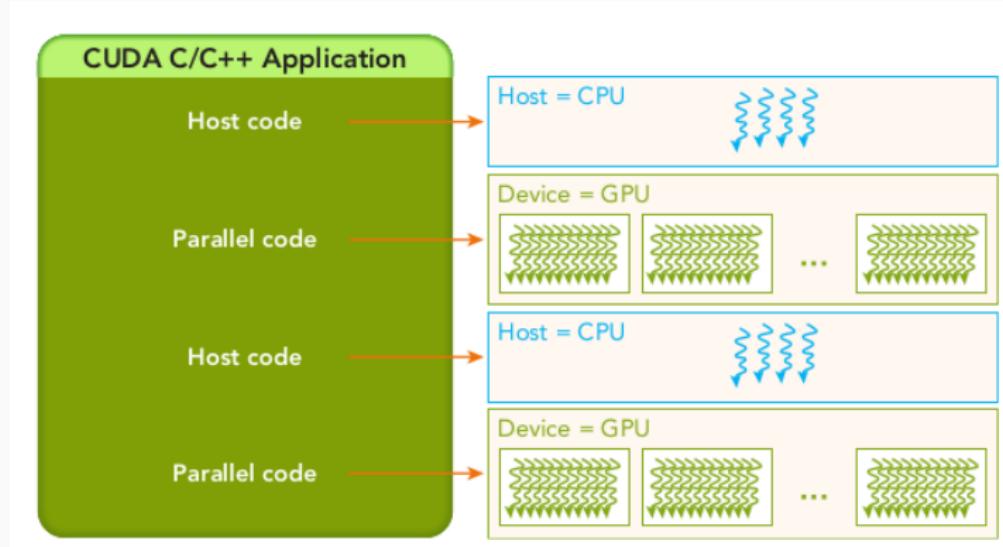


Figura 9: Fluxo de Execução

Plataforma

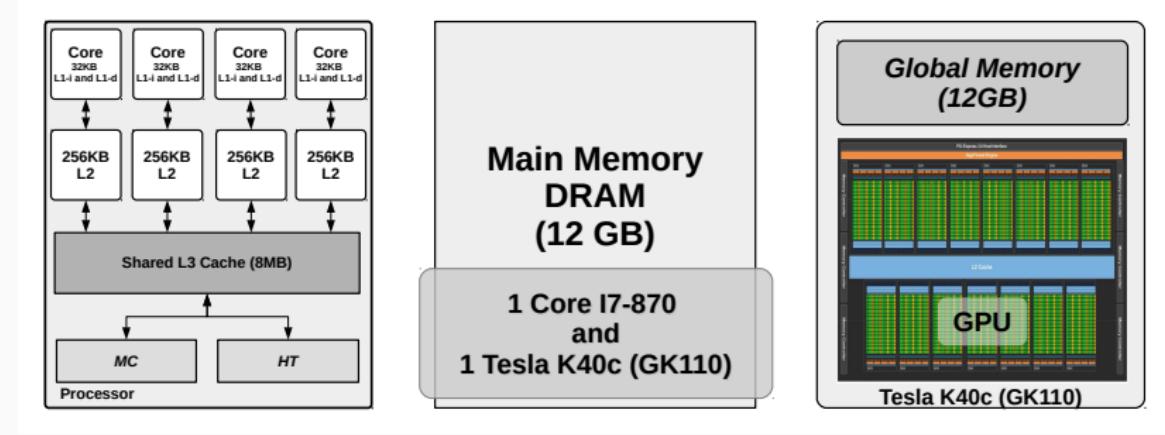


Figura 10: Plataforma i7 com K40c

Hello World!!! i

- Exemplo: hello-world-1

```
#include <stdio.h>

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

Terminal

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

- 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.

Hello World!!! i

- Exemplo: hello-world-2

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

__global__ void helloKernel(){

}

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

    helloKernel<<<1,1>>>();
    printf("Host: Hello World!!!\n");

    return (0);
}
```

- 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!!!
$
```

Hello World!!! i

- Exemplo: hello-world-3

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

__global__ void helloKernel() {
    int blockIdx = blockIdx.x
        + blockIdx.y * gridDim.x
        + blockIdx.z * gridDim.x * gridDim.y;

    int threadId = threadIdx.x
        + threadIdx.y * blockDim.x
        + threadIdx.z * blockDim.x * blockDim.y
        + blockIdx * blockDim.x * blockDim.y * blockDim.z;
    printf("GPU: gridDim:(%2d, %2d, %2d) blockDim:(%2d, %2d, %2d)
blockIdx:(%2d, %2d, %2d) threadIdx:(%2d, %2d, %2d) ->
Thread[%2d]: %s\n", gridDim.x, gridDim.y, gridDim.z, blockDim.x,
blockDim.y, blockDim.z, blockIdx.x, blockIdx.y, blockIdx.z,
threadIdx.x, threadIdx.y, threadIdx.z, threadId,
"Hello World!!!\n");
}
```

```
int main(int argc, char **argv) {
    // Define the GPU id to work.
    cudaSetDevice(0);

    // Hello from host.
    printf("Host: Hello World!!!\n");

    // Hello from GPU.
    helloKernel<<<1,1>>>();

    // Reset device.
    cudaDeviceReset();

    return (0);
}
```

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: gridDim:( 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()**

Gerenciamento de Memória ii

- 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)`:

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

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

Gerenciamento de Memória iii

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 `<<< #blockos, #threads >>>`
- Serão criadas $(\#blockos \times \#threads)$ *threads*
- Cada uma delas irá executar uma cópia do código do *kernel*.
- A especificação da configuração do arranjo de *threads* pode variar conforme o domínio do problema.

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

Exemplo Soma de Vetores usando Blocos i

- 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*

```
--global__ void vecAdd(float *a,           size_t bytes = n * sizeof(float);  
float *b, float *c, int n) {  
    int id = blockIdx.x;  
    if (id < n)  
        c[id] = a[id] + b[id];  
}  
  
// Alocacao de memoria para host e device  
h_a = (float *)malloc(bytes);  
h_b = (float *)malloc(bytes);  
h_c = (float *)malloc(bytes);  
cudaMalloc(&d_a, bytes);  
cudaMalloc(&d_b, bytes);  
cudaMalloc(&d_c, bytes);
```

Exemplo Soma de Vetores usando Blocos ii

```
init_array();

// Copia dos arrays para a GPU.
cudaMemcpy(d_a, h_a, bytes,
cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, bytes,
cudaMemcpyHostToDevice);

vecAdd <<<n, 1>>> (d_a, d_b, d_c, n);

// Copia do resultado.
cudaMemcpy(h_c, d_c, bytes,
cudaMemcpyDeviceToHost);

print_array(); check_result();

// Dealocacao de Memoria.
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
free(h_a);
free(h_b);
free(h_c);
```

Exemplo Soma de Vetores usando Threads i

- 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*

```
--global__ void vecAdd(float *a,           size_t bytes = n * sizeof(float);  
float *b, float *c, int n) {  
    int id = threadIdx.x;  
    if (id < n)  
        c[id] = a[id] + b[id];  
}  
  
// Alocacao de memoria para host e device  
h_a = (float *)malloc(bytes);  
h_b = (float *)malloc(bytes);  
h_c = (float *)malloc(bytes);  
cudaMalloc(&d_a, bytes);  
cudaMalloc(&d_b, bytes);  
cudaMalloc(&d_c, bytes);
```

Exemplo Soma de Vetores usando Threads ii

```
init_array();

// Copia dos arrays para a GPU.
cudaMemcpy(d_a, h_a, bytes,
cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, bytes,
cudaMemcpyHostToDevice);

vecAdd <<<1,n>>> (d_a, d_b, d_c, n);

// Copia do resultado.
cudaMemcpy(h_c, d_c, bytes,
cudaMemcpyDeviceToHost);

print_array(); check_result();

// Dealocacao de Memoria.
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
free(h_a);
free(h_b);
free(h_c);
```

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`

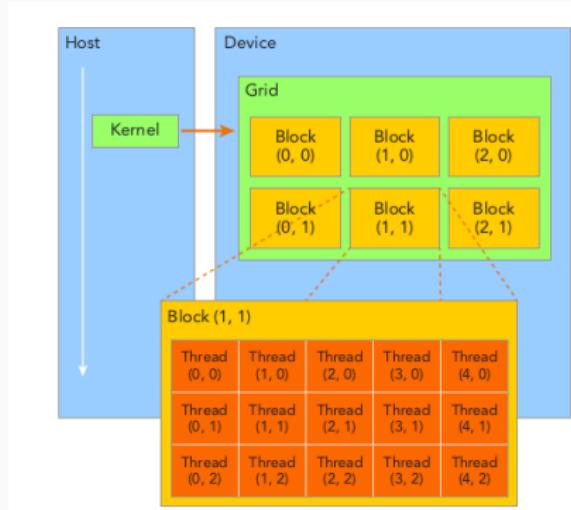


Figura 11: Organização do arranjo de threads

Exemplo de arranjo (4x4)

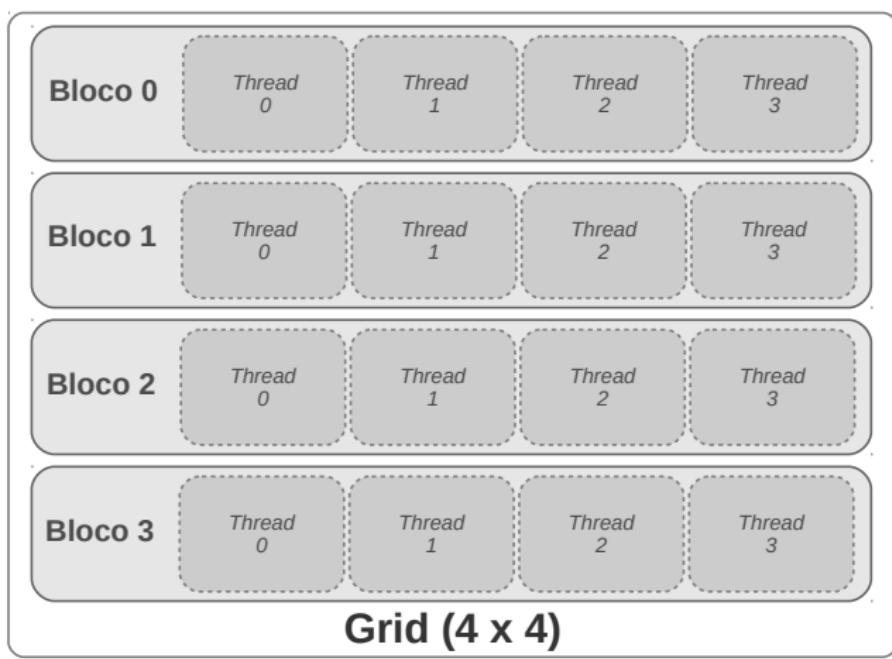


Figura 12: Arranjo (4x4)

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

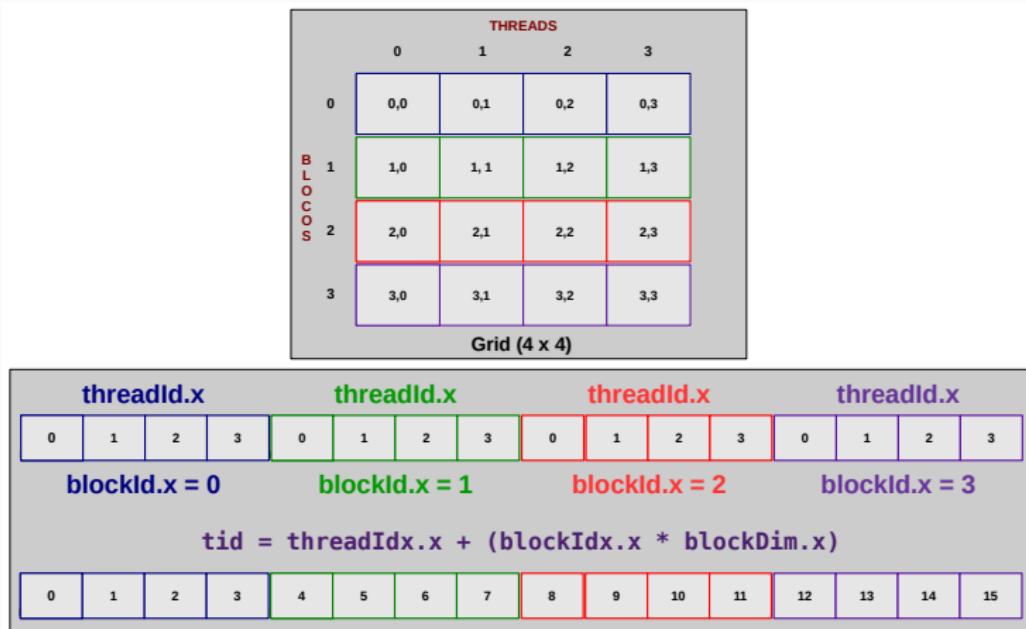


Figura 13: Tradução do id das *threads*

Mapeamento para o Hardware

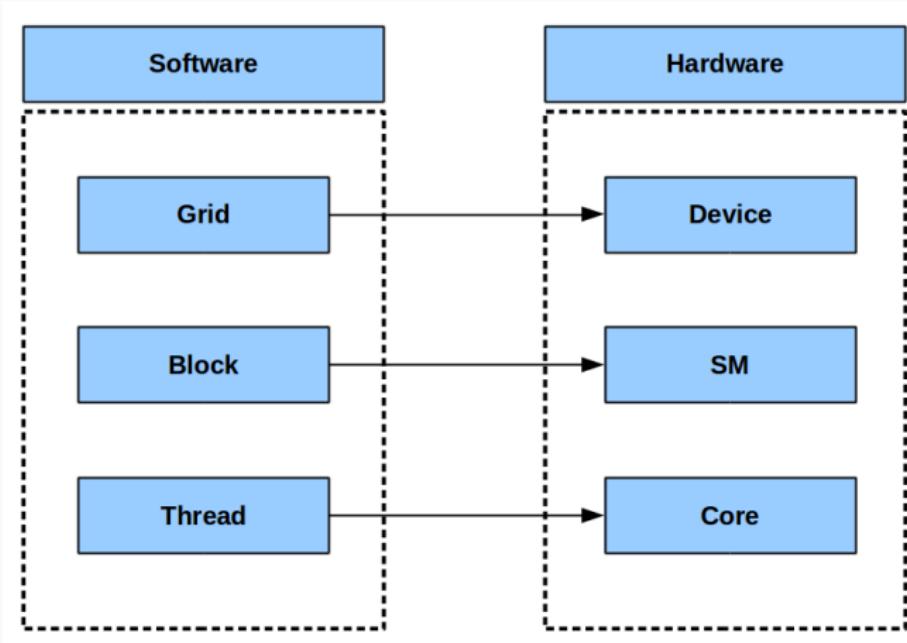


Figura 14: Mapeamento para o Hardware

Visão Lógica e Visão do Hardware

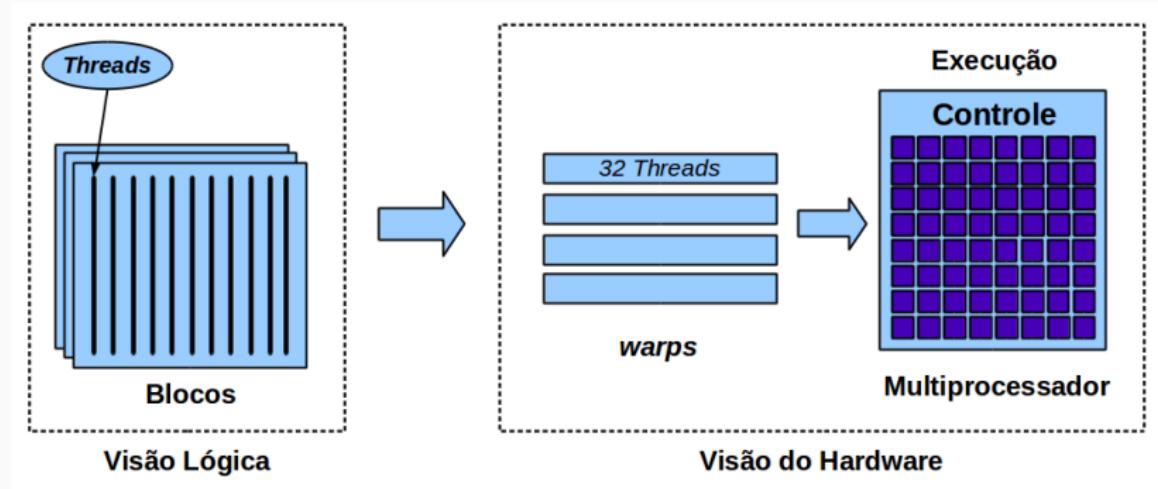


Figura 15: Mapeamento para o Hardware

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.

```
int blockSize, gridSize;

// Numero de threads em cada bloco de threads.
blockSize = 1024;

// Numero de blocos de threads no grid.
gridSize = (int)ceil((float)n/blockSize);

// Chamada a função kernel.
funcKernel<<<gridSize, blockSize>>>(parametros);
```

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

```
int blockSize, gridSize;

__global__ void vecAdd(float *a, float *b, float *c, int n) {
    // Get our global thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < n)
        c[id] = a[id] + b[id];
}

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

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)**

```
/* Definicao do arranjo de threads em blocos do grid. */
dim3 grid(32, 64, 16);
dim3 block(32, 32, 1);

/* Chamada do kernel */
myKernel<<<grid, block>>>(parametros);
```

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 i

- Imprime o valor das variáveis embutidas:

```
--global__ void checkIndex(int funcId) {
    printf("gridDim:(%2d, %2d, %2d) blockDim:(%2d, %2d, %2d)
           blockIdx:(%2d, %2d, %2d) threadIdx:(%2d, %2d, %2d) -> id: %2d\n",
           gridDim.x, gridDim.y, gridDim.z, blockDim.x, blockDim.y, blockDim.z,
           blockIdx.x, blockIdx.y, blockIdx.z, threadIdx.x, threadIdx.y,
           threadIdx.z, getGlobalIdFunc[funcId]());
}

dim3 grid(gx, gy, gz);
dim3 block(bx, by, bz);

int funcId = calculateFunctionId(grid, block);
cudaSetDevice(gpuid);

checkIndex<<<grid, block>>>(funcId);
```

Cálculo das Dimensões de *Kernels*

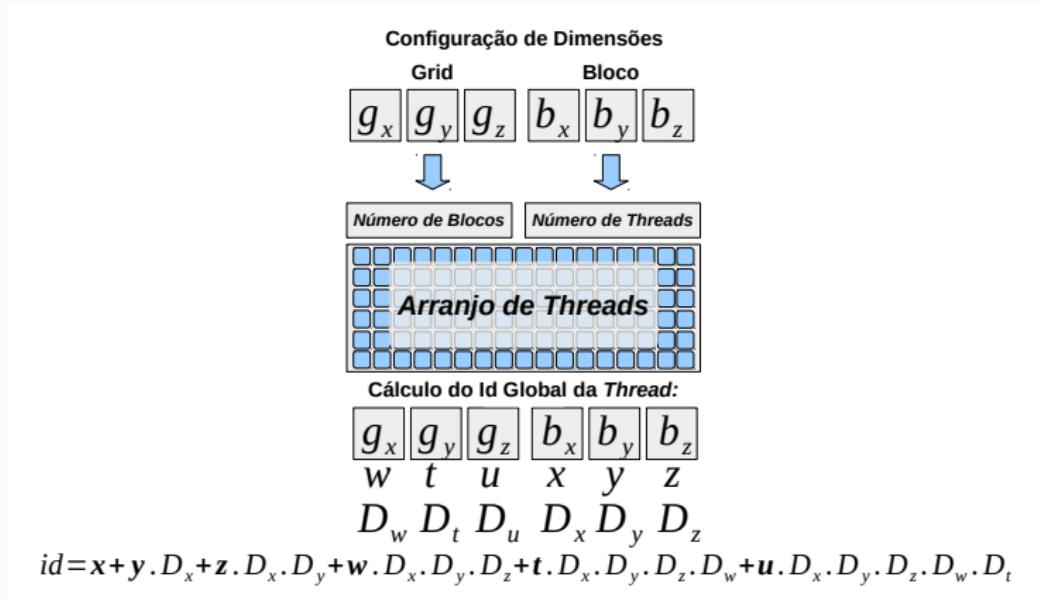


Figura 16: Cálculo das Dimensões

Exemplo: sincos

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

```
void sincos_function_(DATA_TYPE* x, DATA_TYPE* y,
DATA_TYPE* xy, int nx, int ny, int nz) {
    int i, j, k, indice;
    for (i = 0; i < nx; ++i) {
        for (j = 0; j < ny; ++j) {
            for (k = 0; k < nz; ++k) {
                indice = (i * ny * nz) + (j * nz) + k;
                xy[indice] = sin(x[indice]) + cos(y[indice]);
            }
        }
    }
}
```

Exemplo: sincos i

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

```
--global__ void sincos_kernel_3(DATA_TYPE* x, DATA_TYPE* y,  
DATA_TYPE* xy, int nx, int ny, int nz) {  
    int i, j, k, indice;  
    for (i = 0; i < nx; ++i) {  
        for (j = 0; j < ny; ++j) {  
            for (k = 0; k < nz; ++k) {  
                indice = (i * ny * nz) + (j * nz) + k;  
                xy[indice] = sin(x[indice]) + cos(y[indice]);  
            }  
        }  
    }  
}
```

Exemplo: sincos ii

- Apenas uma *thread* deve executá-lo.

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

Exemplo: sincos i

- 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*

```
--global__ void sincos_kernel_2(DATA_TYPE* x, DATA_TYPE* y,
DATA_TYPE* xy, int nx, int ny, int nz, int funcId) {
    int i, j, k, indice;
    i = getGlobalIdFunc[funcId]();
    for (j = 0; j < ny; ++j) {
        for (k = 0; k < nz; ++k) {
            indice = (i * ny * nz) + (j * nz) + k;
            xy[indice] = sin(x[indice]) + cos(y[indice]);
        }
    }
}
```

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.

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

Exemplo: sincos i

- 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 $indice = (i * nz) + k$

```
--global__ void sincos_kernel_1(DATA_TYPE* x, DATA_TYPE* y,
DATA_TYPE* xy, int nx, int ny, int nz, int funcId) {
    int i, k, indice;
    i = getGlobalIdFunc[funcId]();
    for (k = 0; k < nz; k++) {
        indice = (i * nz) + k;
        xy[indice] = sin(x[indice]) + cos(y[indice]);
    }
}
```

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.

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

Exemplo: sincos i

- 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*.

```
__global__ void sincos_kernel_0(DATA_TYPE* x, DATA_TYPE* y,
DATA_TYPE* xy, int nx, int ny, int nz, int funcId) {
    int indice;
    indice = getGlobalIdFunc[funcId]();
    xy[indice] = sin(x[indice]) + cos(y[indice]);
}
```

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.

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

Funções para o cálculo do *id* global i

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

```
__device__ int getGlobalIdx_grid_1D_x() {
    return blockIdx.x;
}
```

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

```
__device__ int getGlobalIdx_block_1D_x() {
    return threadIdx.x;
}
```

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).

$$id = x + y \times Dx + z \times Dx \times Dy + w \times Dx \times Dy \times Dz + t \times Dx \times Dy \times Dz \times Dw + u \times Dx \times Dy \times Dz \times Dw \times Dt$$

Funções para o cálculo do *id* global iii

```
--device__ int getGlobalIdx_grid_3D_xyz_block_3D_xyz() {  
    int blockIdx = blockIdx.x + blockIdx.y * gridDim.x +  
        blockIdx.z * gridDim.x * gridDim.y;  
    int threadId = threadIdx.x + threadIdx.y * blockDim.x +  
        threadIdx.z * blockDim.x * blockDim.y +  
        blockIdx * blockDim.x * blockDim.y * blockDim.z;  
    return threadId;  
}
```

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:

```
cudaSetDeviceFlags(cudaDeviceMapHost);
```

```
cudaHostAlloc(ponteiro, bytes, cudaHostAllocMapped);
```

```
/* Vincula a um ponteiro do device */
```

```
cudaHostGetDevicePointer(ponteiro_device, ponteiro_host, 0);
```

Memória Mapeada no Host ii

```
size_t bytes = n * sizeof(float);

// Set the flag in order to allocate pinned host memory that is
cudaSetDeviceFlags(cudaDeviceMapHost);

// Allocate memory for each vector on host
cudaHostAlloc((void **)&h_a, bytes, cudaHostAllocMapped);
cudaHostAlloc((void **)&h_b, bytes, cudaHostAllocMapped);
cudaHostAlloc((void **)&h_c, bytes, cudaHostAllocMapped);

init_array();
```

Memória Mapeada no Host iii

```
size_t bytes = n * sizeof(float);
cudaHostGetDevicePointer(&d_a, h_a, 0);
cudaHostGetDevicePointer(&d_b, h_b, 0);
cudaHostGetDevicePointer(&d_c, h_c, 0);

// Number of threads in each thread block.
int threadsPerBlock = 256;
// Number of thread blocks in grid.
int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

/* Execute the kernel. */
vecAdd <<< blocksPerGrid, threadsPerBlock >>> (d_a, d_b, d_c, n)
```

Memória Mapeada no Host iv

```
size_t bytes = n * sizeof(float);  
  
/* Synchronize */  
cudaDeviceSynchronize();  
  
print_array();  
check_result();  
  
// Release Memory,  
cudaFreeHost(h_c);  
cudaFreeHost(h_b);  
cudaFreeHost(h_a);  
cudaDeviceReset();
```

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

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

Memória Unificada (UVA) ii

```
// Size, in bytes, of each vector
size_t bytes = n * sizeof(float);
// Allocate memory for each vector on host
cudaMallocManaged(&uva_a, bytes);
cudaMallocManaged(&uva_b, bytes);
cudaMallocManaged(&uva_c, bytes);

init_array();

// Number of threads in each thread block.
int threadsPerBlock = 256;
// Number of thread blocks in grid.
int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;

/* Execute the kernel. */
```

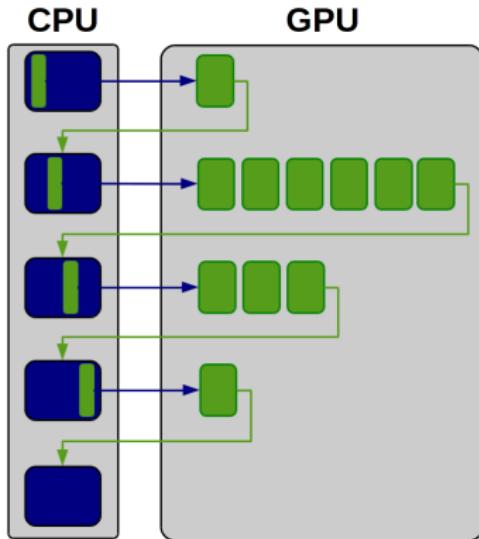
```
vecAdd <<< blocksPerGrid, threadsPerBlock >>> (uva_a, uva_b,
uva_c, n);

/* Synchronize */
cudaDeviceSynchronize();
print_array();
check_result();

// Release device memory
cudaFree(uva_a);
cudaFree(uva_b);
cudaFree(uva_c);
```

Paralelismo Dinâmico

GPU como um coprocessador



GPU com autonomia

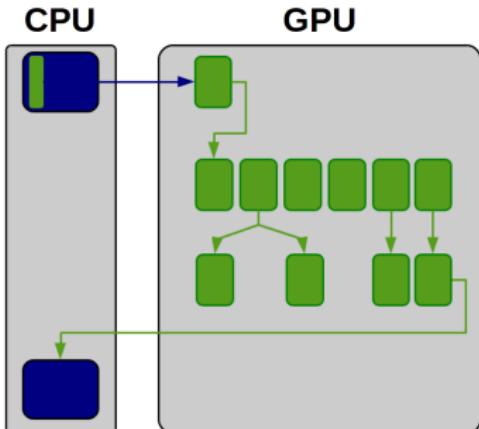


Figura 17: Paralelismo Dinâmico

Paralelismo Dinâmico ii

Paralelismo Dinâmico

```
--global__ SubKernel(void* dados){
    /* Operacoes */
}

--global__ MainKernel(void *dados){
    if (threadIdx.x == 0) {
        SubKernel<<<1, 32>>>(dados);
        cudaThreadSynchronize();
    }
    __syncthreads();
    /* Operacoes do MainKernel */
}

int main(){
    /* Chamada ao MainKernel no codigo principal no host */
    MainKernel<<<8, 32>>>(dados);
    /*...*/
}
```

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

```
--=5622== Profiling application: ./vectoradd.exe 1024
```

```
--=5622== Profiling result:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
36.56%	3.7440us	1	3.7440us	3.7440us	3.7440us	[CUDA memcpy DtoH]
36.56%	3.7440us	2	1.8720us	1.7280us	2.0160us	[CUDA memcpy HtoD]
26.88%	2.7520us	1	2.7520us	2.7520us	2.7520us	vecAdd(float*, float)

```
--=5622== API calls:
```

Time(%)	Time	Calls	Avg	Min	Max	Name
99.57%	139.94ms	3	46.647ms	4.6950us	139.93ms	cudaMalloc
0.16%	229.49us	83	2.7640us	129ns	99.624us	cuDeviceGetAttribute
0.08%	112.30us	3	37.432us	4.9860us	97.222us	cudaFree
0.06%	89.495us	1	89.495us	89.495us	89.495us	cudaEventSynchronize
0.04%	53.484us	3	17.828us	9.6360us	27.668us	cudaMemcpy
0.02%	31.793us	1	31.793us	31.793us	31.793us	cuDeviceTotalMem
0.02%	28.349us	1	28.349us	28.349us	28.349us	cuDeviceGetName
0.01%	19.696us	1	19.696us	19.696us	19.696us	cudaLaunch
0.01%	8.8800us	1	8.8800us	8.8800us	8.8800us	cudaEventElapsedTime
0.01%	8.1360us	1	8.1360us	8.1360us	8.1360us	cudaDeviceSynchronize
0.00%	6.1250us	2	3.0620us	2.4640us	3.6610us	cudaEventRecord

NVIDIA Profiler: nvprof iii

0.00%	3.6420us	4	910ns	163ns	3.0160us	cudaSetupArgument
0.00%	3.4340us	2	1.7170us	764ns	2.6700us	cudaEventCreateWithF
0.00%	1.4810us	2	740ns	582ns	899ns	cuDeviceGetCount
0.00%	1.0700us	1	1.0700us	1.0700us	1.0700us	cudaConfigureCall
0.00%	543ns	2	271ns	209ns	334ns	cuDeviceGet

- 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
```

```
/* ... Imprime a saida do programa */
==5756== NVPROF is profiling process 5756, command: ./vectoradd.exe 32
/* ... Imprime a saida do programa */
==5756== Profiling application: ./vectoradd.exe 32
==5756== Profiling result:
==5756== Event result:
Invocations                      Event Name      Min          Max
Device "Tesla K40c (0)"
Kernel: vecAdd(float*, float*, float*, int)
           1                  warps_launched    8          8
```

NVIDIA Profiler: nvprof v

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

Terminal

```
$ nvprof --metrics inst_executed,sm_efficiency,ipc ./vectoradd.exe 32
```

NVIDIA Profiler: nvprof vi

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

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:

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

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

```
char * cudaGetLastErrorMessage(cudaError_t);
```

Projetos i

Projetos de Pesquisa

Título: *Estudo Exploratório De Técnicas E Mecanismos Para Paralelização Automática E Offloading De Código Em Sistemas Heterogêneos (Pa-Code-Offload).*

Pesquisa em Computação Paralela.

Título: *Investigação sobre Infraestrutura e Suporte ao Desenvolvimento de Aplicações utilizando Tecnologias Blockchain (PRO-BLOCKCHAIN).*

Pesquisa em Tecnologias Blockchain.

Projetos de Extensão

Título: Escola de Computação Paralela (ECP 2024).

Preparar e ministrar minicursos em temas de Computação Paralela.

Título: Academia Blockchain.

Preparar e ministrar minicursos relacionados às Tecnologias Blockchain.

Grupo Estudos

GEDVAC: Grupo de Estudos sobre Desempenho, máquinas Virtuais, Arquitetura de computadores e Compiladores da UTFPR-CM.

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 (ECP 2024)*”.

Material: <https://github.com/rogerioag/minicurso-cuda-seinfo25>

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).

Word Cloud



Referências

Referências i

- Bridges, T. 1990. “[The GPA Machine: A Generally Partitionable MSIMD Architecture](#).” In *Frontiers of Massively Parallel Computation*, 1990. *Proceedings., 3rd Symposium on the*, 196–203.
- Darema, Frederica. 2001. “The SPMD Model: Past, Present and Future.” In *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.

Referências ii

- Denise Stringhini, Alfredo Goldman, Rogério Aparecido Gonçalves. 2012.
“Introdução à Computação Heterogênea.” In *XXXI Jornadas de Atualização Em Informática (JAI)*, 1st series, 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, and Piotr Danilewski. 2011. “Efficient Quicksort and 2D Convex Hull for CUDA, and MSIMD as a Realistic Model of Massively Parallel Computations.”

Referências iii

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>.