



Introdução ao CUDA

Jhoan Fernandes

João Augusto





GPUs e GPGPU

- **Contexto Histórico:** As GPUs surgiram no final dos anos 90 para atender à demanda por jogos e animações realistas.
- **GPGPU (General Purpose computing on GPUs):** Programadores começaram a usar o poder computacional das GPUs para problemas gerais (não gráficos), como busca e ordenação.
- **Dificuldades Iniciais:** No início, programar GPUs exigia o uso de APIs gráficas como Direct3D e OpenGL, o que adicionava complexidade, pois os algoritmos precisavam ser reformulados para conceitos gráficos.



Execução SIMD

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with $x[i] \geq 0$	Datapaths with $x[i] < 0$
1	Test $x[i] \geq 0$	Test $x[i] \geq 0$
2	$x[i] += 1$	Idle
3	Idle	$x[i] -= 2$

Table 6.2 Execution of branch on a system with multiple SMs.

Time	Datapaths with $x[i] \geq 0$ (on SM A)	Datapaths with $x[i] < 0$ (on SM B)
1	Test $x[i] \geq 0$	Test $x[i] \geq 0$
2	$x[i] += 1$	$x[i] -= 2$

- Em um processador SIMD, uma única unidade de controle envia a mesma instrução para múltiplas Unidades de processamento.
- Em sistemas SIMD, mesmo com ramificações condicionais, todas as partes do código (o if e o else) são executadas sequencialmente, com datapaths alternadamente ativos e ociosos, para garantir que a mesma instrução seja transmitida a todos.



Arquitetura da GPU

- **Streaming Multiprocessors (SMs):** GPUs Nvidia são compostas por SMs, que podem conter várias unidades de controle e muitos *datapaths* (Streams Processors - SPs). SMs operam assincronamente, o que pode levar a uma execução mais eficiente de branches.
- **SIMT (Single Instruction Multiple Thread):** Nvidia usa o termo SIMT em vez de SIMD. Isso porque as threads em um SM que executam a mesma instrução podem não ser executadas simultaneamente; algumas podem bloquear para esconder a latência de acesso à memória, enquanto outras progridem.



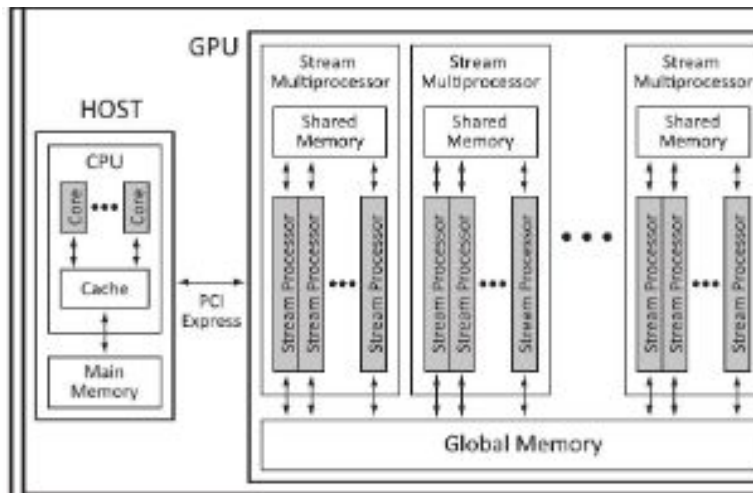
Arquitetura da GPU

- **Hierarquia de Memória na GPU:**

- **Memória Compartilhada:** Pequena, rápida, acessível apenas pelos SPs de um mesmo SM (ou threads de um mesmo bloco).
- **Memória Global:** Maior, mais lenta, acessível por todos os SPs de todos os SMs no chip.
- **Registradores:** Mais rápidos e menores, usados para variáveis locais. Variáveis locais podem ser "derramadas" para a memória global se não houver espaço suficiente nos registradores.

Arquitetura da GPU

- **Host e Device:** A CPU e sua memória são chamados de "host", e a GPU e sua memória são chamados de "device". Em sistemas mais antigos, a transferência explícita de dados entre host e device era necessária.

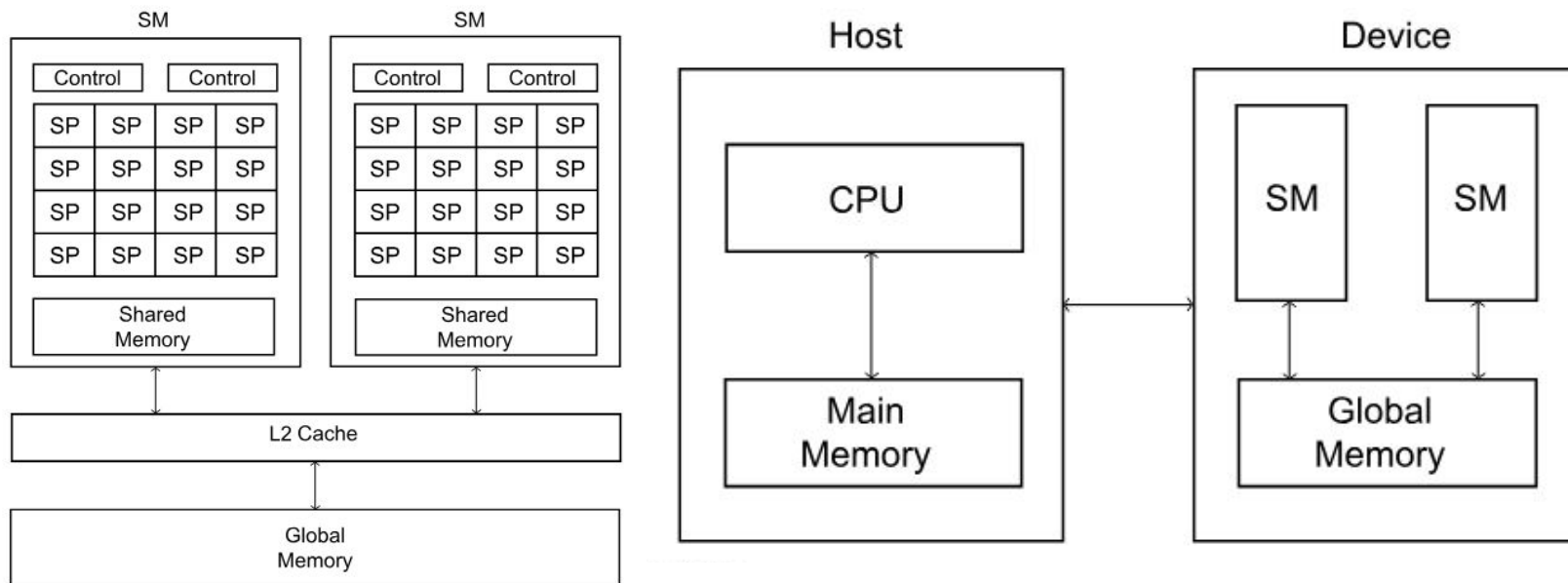





Computação Heterogênea

- A programação em GPUs é um exemplo de computação heterogênea, pois envolve o uso de um processador host (CPU) e um processador device (GPU) com arquiteturas diferentes.
- Embora seja um único programa (SPMD - Single Program Multiple Data), efetivamente são escritos dois programas: um para a CPU e outro para a GPU.

Computação Heterogênea





```
• joao@pop-os:~/Desktop/Competitive_programing/programacao_paralela/cuda/apresentacao$ nvcc -arch=native hello_world.cu -o main
• joao@pop-os:~/Desktop/Competitive_programing/programacao_paralela/cuda/apresentacao$ ./main 5
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
Hello from thread 4!
```

CUDA Hello World

```
1  #include <stdio.h>
2  #include <cuda.h> // Header file for CUDA
3
4  // Device code: runs on GPU
5  __global__ void Hello(void) {
6      printf("Hello from thread %d!\n", threadIdx.x);
7  }
8
9  // Host code: runs on CPU
10 int main(int argc, char* argv[]) {
11     int thread_count; // Number of threads to run on GPU
12
13     thread_count = strtol(argv[1], NULL, 10); // Get thread_count from command line
14
15     Hello<<<1, thread_count>>>(); // Start thread_count threads on GPU
16
17     cudaDeviceSynchronize(); // Wait for GPU to finish
18
19     return 0;
20 }
```



Threads, Blocos e Grids

Threads:

- Unidades de execução mais básicas do código *kernel*.
- Executam a mesma lógica (SPMD), mas em diferentes dados.
- São "leves", permitindo paralelismo massivo (milhares a milhões).
- Possuem um ID único dentro de seu bloco.



Threads, Blocos e Grids

Blocos de Threads:

- Grupos de threads que executam juntas em um único Streaming Multiprocessor (SM).
- Threads dentro do mesmo bloco podem:
 - Compartilhar memória rápida;
 - Sincronizar-se usando barreiras.
- O número de threads por bloco é especificado no lançamento do *kernel*.

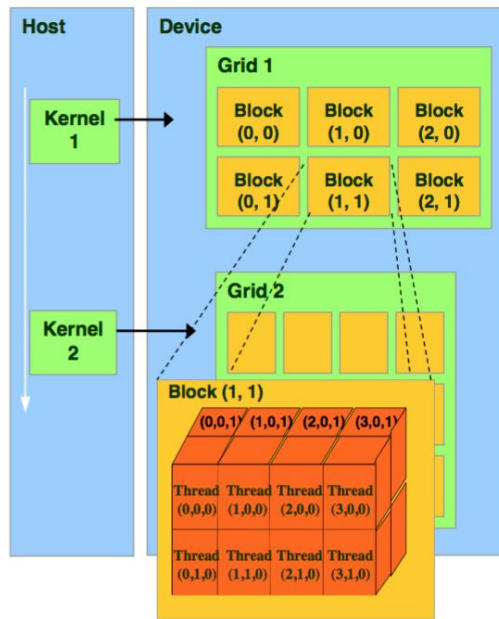


Threads, Blocos e Grids

Grid:

- A coleção de todos os blocos de threads iniciados por um único *kernel*.
- Representa o trabalho total a ser processado pela GPU.
- O número de blocos no grid é especificado no lançamento do *kernel*.

Threads, Blocos e Grids





Capacidades de Computação e Arquiteturas de Dispositivos Nvidia

- **Capacidade de Computação (Compute Capability):** É um número no formato `major.minor` (ex: 8.0, 7.5) que define os recursos e limites de uma GPU. Isso determina, por exemplo:
 - O número máximo de threads por bloco (geralmente 1024 para capacidade > 1.x).
 - A quantidade de memória compartilhada e registradores disponíveis.
- **Arquiteturas de GPU:** A Nvidia nomeia suas microarquiteturas, como **Ampere, Pascal, Volta e Turing**, que correspondem a diferentes capacidades de computação

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.b	2.b	3.b	5.b	6.b	7.0	7.5



Retornando Resultados de Kernels CUDA

- **O Problema Fundamental:** Kernels em CUDA têm retorno `void` e não podem usar passagem por referência padrão do C para retornar um valor, devido aos espaços de memória separados (host vs. dispositivo).

```
__global__ void Add(int x, int y, int* sum_p) {  
    *sum_p = x + y;  
} /* Add */
```

```
int main(void) {  
    int sum = -5;  
    Add <<<1, 1>>> (2, 3, &sum);  
    cudaDeviceSynchronize();  
    printf("The sum is %d\n", sum);  
  
    return 0;  
}
```



Retornando Resultados de Kernels CUDA

- **Solução 1: Usar Ponteiros para Memória Alocada**
 - **Com Memória Unificada:** Usa-se `cudaMallocManaged` para alocar memória acessível por ambos. O kernel escreve o resultado no endereço de memória, que fica disponível para o host.
 - **Com Transferência Explícita:** Aloca-se memória separada no host (`malloc`) e no dispositivo (`cudaMalloc`). O resultado é copiado de volta para o host com `cudaMemcpy`.



Retornando Resultados de Kernels CUDA

```
int main(void) {  
    int* sum_p;  
    cudaMallocManaged(&sum_p, sizeof(int));  
    *sum_p = -5;  
    Add <<<1, 1>>> (2, 3, sum_p);  
    cudaDeviceSynchronize();  
    printf("The sum is %d\n", *sum_p);  
    cudaFree(sum_p);  
  
    return 0;  
}
```



Retornando Resultados de Kernels CUDA

```
int main(void) {  
    int *hsum_p, *dsum_p;  
    hsum_p = (int*) malloc(sizeof(int));  
    cudaMalloc(&dsum_p, sizeof(int));  
    *hsum_p = -5;  
    Add <<<1, 1>>> (2, 3, dsum_p);  
    cudaMemcpy(hsum_p, dsum_p, sizeof(int),  
               cudaMemcpyDeviceToHost);  
    printf("The sum is %d\n", *hsum_p);  
    free(hsum_p);  
    cudaFree(dsum_p);  
  
    return 0;  
}
```



Retornando Resultados de Kernels CUDA

- **Solução 2: Variável Global Gerenciada (`__managed__`)**
 - Declara-se uma variável global com o qualificador `__managed__`, tornando-a acessível tanto pelo host quanto pelo dispositivo.

```
__managed__ int sum;

__global__ void Add(int x, int y) {
    sum = x + y;
} /* Add */

int main(void) {
    sum = -5;
    Add <<<1, 1>>> (2, 3);

    cudaDeviceSynchronize();
    printf("After kernel: The sum is %d\n", sum);

    return 0;
}
```

Soma de Vetores

```
global __void Vec_add(  
    const float x[], /* in */  
    const float y[], /* in */  
    float z[],       /* out */  
    const int n      /* in */  
)  
{  
    int my_elt = blockDim.x * blockIdx.x + threadIdx.x;  
  
    /* total threads = blk_ct * th_per_blk pode ser > n */  
    if (my_elt < n)  
    {  
        z[my_elt] = x[my_elt] + y[my_elt];  
    }  
} /* Vec_add */
```



Regra dos trapézios

```
float Serial_trap(  
    const float a, /* in */  
    const float b, /* in */  
    const int n    /* in */  
)  
{  
    float x, h = (b - a) / n;  
    float trap = 0.5 * (f(a) + f(b));  
  
    for (int i = 1; i <= n - 1; i++)  
    {  
        x = a + i * h;  
        trap += f(x);  
    }  
  
    trap = trap * h;  
  
    return trap;  
} /* Serial_trap */
```