Notas dos slides

APRESENTAÇÃO

O presente conjunto de slides pertence à coleção produzida para a disciplina *Introdução ao Processamento Paralelo e Distribuído* ofertada aos cursos de bacharelado em Ciência da Computação e em Engenharia da Computação pelo Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas.

Os slides disponibilizados complementam as videoaulas produzidas e tratam de pontos específicos da disciplina. Embora tenham sido produzidos para ser assistidos de forma independente, a sequência informada reflete o encadeamento dos assuntos no desenvolvimento do conteúdo programático previsto para a disciplina.

computação
Universidade Federal de Pelotas

universidade federal de pelotas
CDTec
centro de desenvolvimento tecnológico

Notas da videoaula

DESCRIÇÃO

Nesta videoaula é abordado o modelo de programação CUDA. Através de códigos de aplicações CUDA, são demonstrados alguns dos recursos disponibilizados pela sua interface de programação a fim de possibilitar a descrição de paralelismo em GPUs.

OBJETIVOS

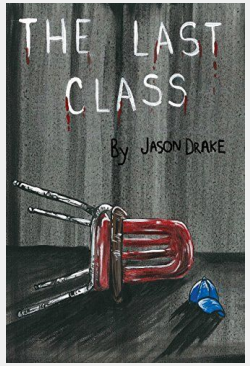
Nesta videoaula o aluno conhecerá, de forma inicial, o modelo de programação CUDA. Através dos exemplos implementados, ele compreenderá como utilizar esta ferramenta para codificar um programa paralelo para GPUs, bem como terá uma base que lhe permite aprofundar seus conhecimentos e implementar aplicações mais complexas.

> **Talent wins games, but teamwork and intelligence win championships.**
>
> Michael Jordan

Programação com CUDA

Conceitos importantes vistos na útima videoaula:

- Ambiente CUDA: compilador **nvcc**;
- Modelo de execução do programa (CPU, GPU, CPU...);
- *Kernel;*
- Grid, blocos e threads;
- ***Host*** e ***Device***?
- Código executado pela GPU acessa apenas a memória da GPU;



Primeiro programa em CUDA

Compilação (utilizar extensão **.cu**):
`nvcc hello_word.cu -o hello_word`

```c
/* hello_word.cu */
#include <iostream>

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

Código somente CPU (*host*):

- `nvcc` utiliza o compilador padrão C do sistema (`gcc`) para compilar o código para CPU;
- Execução somente em CPU;
- Ambiente CUDA é 100% compatível com C;

Primeiro programa em CUDA

Compilação (utilizar extensão **.cu**):
`nvcc hello_word.cu -o hello_word`

```c
/* hello_word.cu */
#include <iostream>

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

Código somente CPU (*host*):

- `nvcc` utiliza o compilador padrão C do sistema (`gcc`) para compilar o código para CPU;
- Execução somente em CPU;
- Ambiente CUDA é 100% compatível com C;

```c
/* hello_word.cu */
#include <iostream>

__global__ void kernel( void ) {
   printf( "Hello, World from GPU!\n" );
}

int main( void ) {
   kernel<<<1,1>>>();
   return 0;
}
```

Código CPU (*host*) e GPU (*device*):

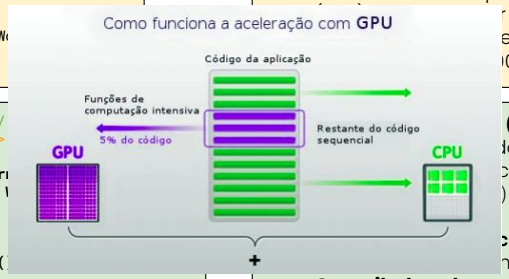
- `__global__` utilizado para definir a função do *kernel* que será executado na GPU;
- `kernel<<<1,1>>>()`: chamada da função com o *kernel*;
- **Compilado pelo `nvcc` e executado na GPU:** *kernel*, chamada em `main` -> linhas em negrito;
- **Compilado pelo `gcc` e executado na CPU:** restante do código;

Primeiro programa em CUDA

Compilação (utilizar extensão **.cu**):
`nvcc hello_word.cu -o hello_word`

```
/* hello_word.cu */
#include <iostream>

int main( void ) {
   printf("Hello, W
   return 0;
}
```



Código somente CPU (*host*):
- nvcc utiliza o compilador padrão C do sistema
- o código para CPU;
- em CPU;
- 0% compatível com C;

```
/* hello_word.cu */
#include <iostream>

__global__ void keru
   printf( "Hello, W
}

int main( void ) {
   kernel<<<1,1>>>(
   return 0;
}
```

(*device*):
- do para definir a função do
- cutado na GPU;
-) : chamada da função com

cc e executado na GPU:
- main -> linhas em negrito;

- **Compilado pelo gcc e executado na CPU:** restante do código;

CUDA

Questão 1

Por que preciso de um outro compilador (nvcc) para construir meu programa CUDA?

Questão 2

Por que tenho de utilizar o marcador `__global__` para definir o *kernel*?

Questão 3

Por que é necessário utilizar o `<<<1,1>>>` para executar o *kernel*?

Questão 4

Qual o ciclo de execução do programa `hello_word` ao rodá-lo na sua versão GPU?

Passando parâmetros

Compilação (utilizar extensão **.cu**):
`nvcc soma.cu -o soma`

```
/* soma.cu */
#include <iostream>

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

int main( void ) {
   int c, *dev_c;

   cudaMalloc(&dev_c, sizeof(int));

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

   cudaMemcpy(&c, dev_c, sizeof(int),
              cudaMemcpyDeviceToHost);

   printf( "2 + 7 = %d\n", c );

   cudaFree(dev_c);
   return 0;
}
```

Definição do *kernel*:
- Realiza a soma e armazena na variável `c`;
- `c` é um ponteiro, sendo utilizado para transferir memória (*device->host*);

Alocação de memória na GPU:
- `cudaMalloc`: indica ao compilador para alocar `dev_c` na memória do *device*:
  - Após alocado, `dev_c` pode ser utilizado para ler/escrever na memória da GPU a partir do código que executa no *device*;
  - **NÃO** pode ser utilizado para ler/escrever na memória da GPU a partir do código que executa no *host*;

Passando parâmetros

Compilação (utilizar extensão **.cu**):
`nvcc soma.cu -o soma`

```
/* soma.cu */
#include <iostream>

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

int main( void ) {
   int c, *dev_c;

   cudaMalloc(&dev_c, sizeof(int));

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

   cudaMemcpy(&c, dev_c, sizeof(int),
              cudaMemcpyDeviceToHost);

   printf( "2 + 7 = %d\n", c );

   cudaFree(dev_c);
   return 0;
}
```

Chamada do *kernel*:
- Chamada do *kernel* para execução no *device*;
- Passagem dos parâmetros:
  - `<<<1, 1>>>`: kernel executado por 1 bloco e 1 thread (sequêncial);
  - 2 e 7: constantes;
  - `dev_c`: ponteiro com memória previamente alocada no *device*;
- Valor da soma gravado em `*c` (`*dev_c`), porém apenas na memória da GPU:
  - Após a execução do *kernel*, o valor da soma **NÃO** pode ser acessado diretamente no *host* (função `main()` neste caso);

Passando parâmetros

Compilação (utilizar extensão **.cu**):
`nvcc soma.cu -o soma`

```
/* soma.cu */
#include <iostream>

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

int main( void ) {
   int c, *dev_c;

   cudaMalloc(&dev_c, sizeof(int));

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

   cudaMemcpy(&c, dev_c, sizeof(int),
              cudaMemcpyDeviceToHost);

   printf( "2 + 7 = %d\n", c );

   cudaFree(dev_c);
   return 0;
}
```

Cópia de memória:

- `cudaMencpy`: efetua a cópia da memória entre endereços do *host* e *device*;
- Parâmetros:
  - 1- ponteiro de destino;
  - 2- ponteiro de origem;
  - 3- tamanho a ser copiado;
  - 4- direção da cópia:
    - `cudaMemcpyDeviceToHost`: GPU->CPU;
    - `cudaMemcpyHostToDevice`: CPU->GPU;

Passando parâmetros

Compilação (utilizar extensão **.cu**):
`nvcc soma.cu -o soma`

```
/* soma.cu */
#include <iostream>

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

int main( void ) {
   int c, *dev_c;

   cudaMalloc(&dev_c, sizeof(int));

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

   cudaMemcpy(&c, dev_c, sizeof(int),
              cudaMemcpyDeviceToHost);

   printf( "2 + 7 = %d\n", c );

   cudaFree(dev_c);
   return 0;
}
```

Impressão do resultado:

- c já possui o valor da soma calculado pelo *kernel* executado no *device*:
  - Transferência de memória já executada;

Liberação da memória:

- `cudaFree`: indica ao compilador para liberar a memória alocada no *device* para `dev_c`:
  - Sempre que executar um `cudaMalloc`, deve-se executar um `cudaFree`;

Mas e o paralelismo?

Exemplos até o momento somente sequenciais:

- Um bloco e uma thread para executar o *kernel*;
- **Estratégia simples para ativar o paralelismo:**
  - Definir na chamada do *kernel* mais blocos para executá-lo;
  - *Lembrando*: blocos são executados em paralelo.

Exemplo paralelo

Compilação (utilizar extensão **.cu**):
`nvcc soma_vetor.cu -o soma_vetor`

```
/* soma_vetor.cu - Parte 1 */
#include <iostream>

#define N 100

//Kernel [add(int *a, int *b, int *c)] na Parte 2
int main( void ) {
   int a[N], b[N], c[N], *dev_a, *dev_b, *dev_c;

   cudaMalloc(&dev_a, N * sizeof(int));
   cudaMalloc(&dev_b, N * sizeof(int));
   cudaMalloc(&dev_c, N * sizeof(int));

   for(int i=0; i<N; i++) {
       a[i] = -i; b[i] = i * i;
   }

   cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);

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

   cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);

   for(int i=0; i<N; i++)
       printf("%d + %d = %d\n", a[i], b[i], c[i]);

   cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
   return 0;
}
```

- Alocação de memória na GPU.
- Inicialização dos vetores de entrada.
- Cópia de memória da CPU (*host*) para a GPU (*device*) dos vetores de entrada.
- Chamada do *kernel* criando N blocos, um para cada posição do vetor. Cada bloco possuirá uma única thread.
- Cópia de memória da GPU (*device*) para a CPU (*host*) do vetor com o resultado da soma.
- Imprime o resultado da soma dos vetores, sendo que o valor de `c` foi computado na GPU.
- Liberação da memória na GPU.

Exemplo paralelo

Compilação (utilizar extensão **.cu**):
`nvcc soma_vetor.cu -o soma_vetor`

```
/* soma_vetor.cu - Parte 1 */
#include <iostream>

#define N 100

//Kernel [add(int *a, int *b, int *c)] na Parte 2
int main( void ) {
   int a[N], b[N], c[N], *dev_a, *dev_b, *dev_c;

   cudaMalloc(&dev_a, N * sizeof(int));
   cudaMalloc(&dev_b, N * sizeof(int));
   cudaMalloc(&dev_c, N * sizeof(int));

   for(int i=0; i<N; i++) {
        a[i] = -i; b[i] = i * i;
   }

   cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice);

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

   cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);

   for(int i=0; i<N; i++)
        printf("%d + %d = %d\n", a[i], b[i], c[i]);

   cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
   return 0;
}
```

Chamada do *kernel* criando N blocos, um para cada posição do vetor. Cada bloco possuirá uma única thread.

Exemplo paralelo

Compilação (utilizar extensão **.cu**):
`nvcc soma_vetor.cu -o soma_vetor`

```
/* soma_vetor.cu - Parte 2 */

__global__ void add(int *a, int *b, int *c) {

   int tid = blockIdx.x;

   if(tid < N)
      c[tid] = a[tid] + b[tid];
}
```

`blockIdx.x`: variável definida pelo CUDA que contém o índice (número) do bloco (0…N-1) que está executando o *kernel* no *device*.

`tid`: através do índice do bloco, pode-se acessar as posições dos vetores.

Boa prática verificar se o índice (`tid`) está dentro do conjunto do vetor, mesmo que se saiba que neste exemplo sempre estará.

Assim:

Não é necessário ter um laço para percorrer o vetor.
Ao executar a chamada do *kernel*...

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

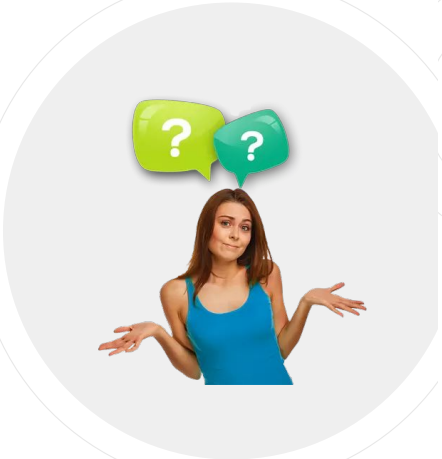
serão **criados N blocos** para executá-lo na GPU (blocos executados em paralelo).

Cada bloco possuirá um índice que pode ser acessado pela variável `blockIdx.x`.

E as threads?

É possível lançar *kernels* para serem executados combinando blocos e threads. **Exemplos:**

- O programa precisa de muitos índices, mais do que os blocos permitidos pela GPU:
  - Varia de acordo com a arquitetura;
  - Normalmente é 65.535;
- Deseja-se utilizar a memória compartilhada do bloco para a computação - cooperação;
- Processamento com mais dimensões:
  - **Blocos:** suportam 2 dimensões;
  - **Threads:** suportam 3 dimensões;



Exemplos com blocos e threads

Usando threads:

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

int main( void ) {
   //Código omitido
   add<<<1,N>>>(dev_a, dev_b, dev_c);
   //Código omitido
}
```

`threadIdx.x`: variável definida pelo CUDA que contém o índice (número) da thread dentro do bloco que está executando o *kernel* no *device*.

Como só há 1 bloco, `tid` pode receber o índice da thread diretamente.

Chamada do *kernel* criando 1 bloco com N threads.

Exemplos com blocos e threads

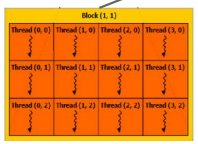
Usando blocos e threads:

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

int main( void ) {
    //Código omitido
    add<<<32, 32>>>(dev_a, dev_b, dev_c);
    //Código omitido
}
```

Fórmula para calcular o `tid` baseada no índice do bloco e da thread.

`blockDim.x`: variável definida pelo CUDA que contém o índice da dimensão atual do bloco, **pois as threads são dispostas dentro do bloco em várias dimensões**;



Chamada do *kernel* criando 32 blocos com 32 threads cada.

Exemplos blocos e threads

Dimensões:

```
__global__ void kernel_exemplo() {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    __shared__ int var_test = x + y;
    //...
}

int main( void ) {
    //...
    dim3 blocks(32,32);
    dim3 threads(16,16);
    kernel_exemplo<<<blocks,threads>>>();
    //...
}
```

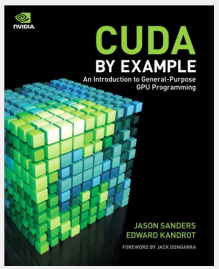
Acesso à segunda dimensão de `threadIdx`, `blockIdx` e `blockDim`, através de `.y`

`__shared__`: indica ao compilador para alocar `var_test` na memória compartilhada do bloco.

`dim3`: estrutura do CUDA que permite executar o *kernel* com mais de uma dimensão de blocos e threads.

Dúvidas?

- Programação para GPUs com CUDA é difícil:
  - **Já podemos implementar um programa simples para GPU;**
- Recomendação:
  - Livro: CUDA by Example
- Mais recursos de CUDA:
  - **CUDA Unified Memory**: facilita o gerenciamento da memória:
    - https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
  - **CUDA Thrust**: STL do C++ implementada em CUDA:
    - https://docs.nvidia.com/cuda/thrust/index.html



SANDERS, Jason; KANDROT, Edward. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional: 2010.

* Exemplos desta videoaula retirados deste livro.