MC4: Introdução à Programação Paralela em GPU para a Implementação de Métodos Numéricos

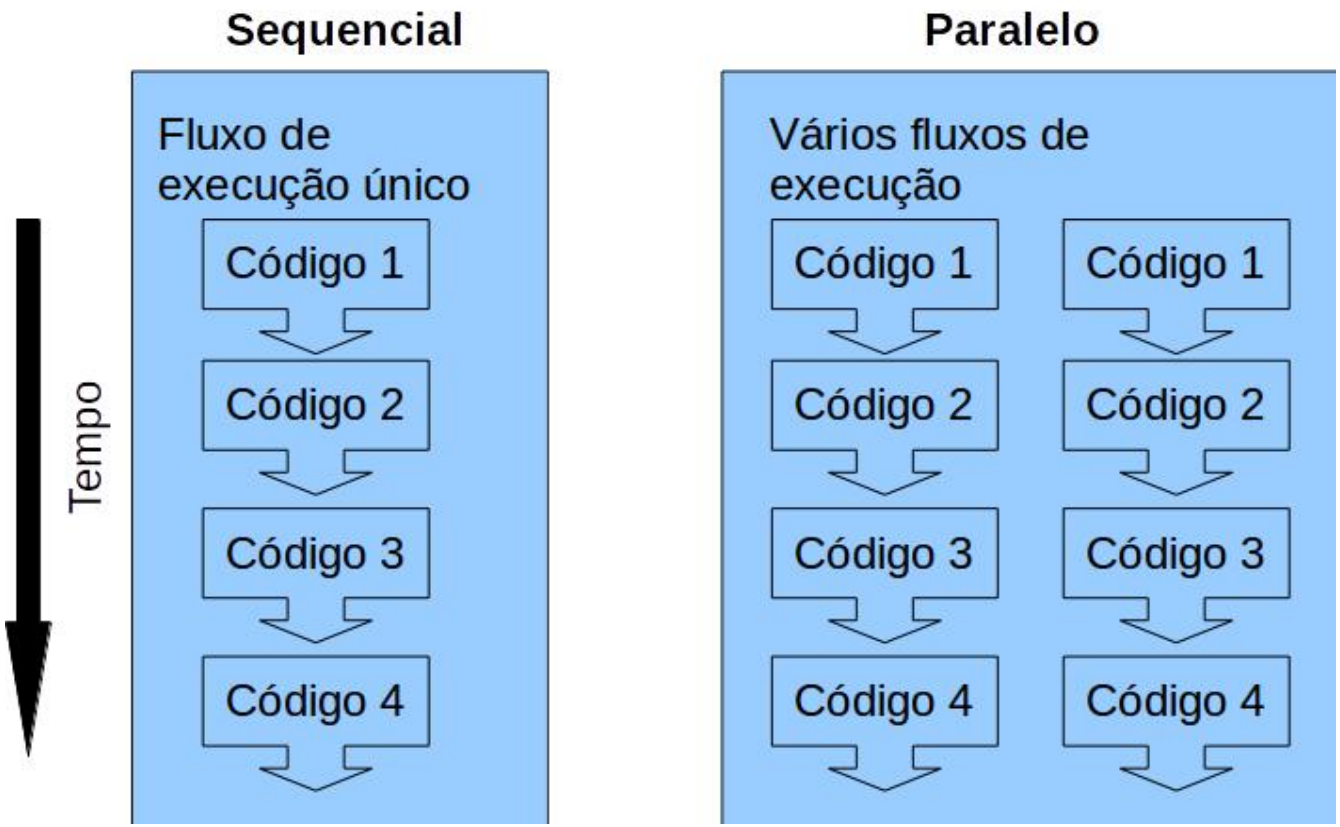
Aula 1: Introdução à programação paralela e ao ambiente de programação CUDA

Profs.: Daniel Alfaro e Silvana Rossetto
(DCC/IM/UFRJ)

20 de setembro de 2017

Definição de computação paralela

Programas de computador que incluem **fluxos de execução distintos** que podem executar simultaneamente na mesma máquina ou em máquinas separadas



Finalidade da computação paralela

O objetivo principal da **computação paralela** é **reduzir o tempo de processamento** requerido para executar uma aplicação

- permite que problemas computacionalmente complexos sejam resolvidos em intervalo de tempo viável

Benefícios da computação paralela

Limite do ganho de velocidade dos processadores

- A possibilidade de ganhos de velocidade significativos dos processadores tem alcançado o limite físico
  - nas novas arquiteturas de hardware paralelo o ganho de desempenho real só é obtido por aplicações paralelas

O mundo real é paralelo

- Escrever programas sequenciais requer impor uma ordem sobre ações que são independentes e poderiam ser executadas concorrentemente

Etapas do desenvolvimento de aplicações paralelas

1. Dividir a aplicação em tarefas que podem ser submetidas à execução concorrente (paralela)
2. Definir a estratégia de interação entre os elementos da aplicação e de controle da evolução da aplicação
3. Minimizar os custos associados à execução paralela

Desafios da computação paralela

Esforço para balancear os diferentes blocos de construção

- Controle cuidadoso da relação entre **velocidade do processador**, da **interconexão de comunicação** e da **hierarquia de memória**
- O tempo de execução da aplicação pode piorar da versão sequencial para a versão paralela

Dependência da arquitetura e modelo de programação

- Dependendo das características e demandas da aplicação, nem toda arquitetura é adequada
- Escolher o ambiente de programação e execução mais adequado é uma das tarefas do desenvolvedor

Objetivos do minicurso

1. Apresentar os conceitos básicos da programação paralela usando como **arquitetura alvo as placas de processamento gráfico** (GPUs) e o ambiente de programação CUDA/C
2. Apresentar exemplos de implementação de **aplicações paralelas para resolver problemas de computação numérica** usando GPUs
3. Desenvolver **habilidades práticas de pensamento computacional voltado para a programação paralela**

Livro texto e bibliografia do minicurso

Livro texto do minicurso

Vol. 84 — Notas em Matemática Aplicada

Bibliografia de referência

- D. Kirk e W. Hwu Wen-mei. Programming massively parallel processors: a hands-on approach. Newnes, 2ª ed., 2012
- CUDA C Programming Guide
- V. Kindratenko, Volodymyr. Numerical Computations with GPUs. Springer, 2014

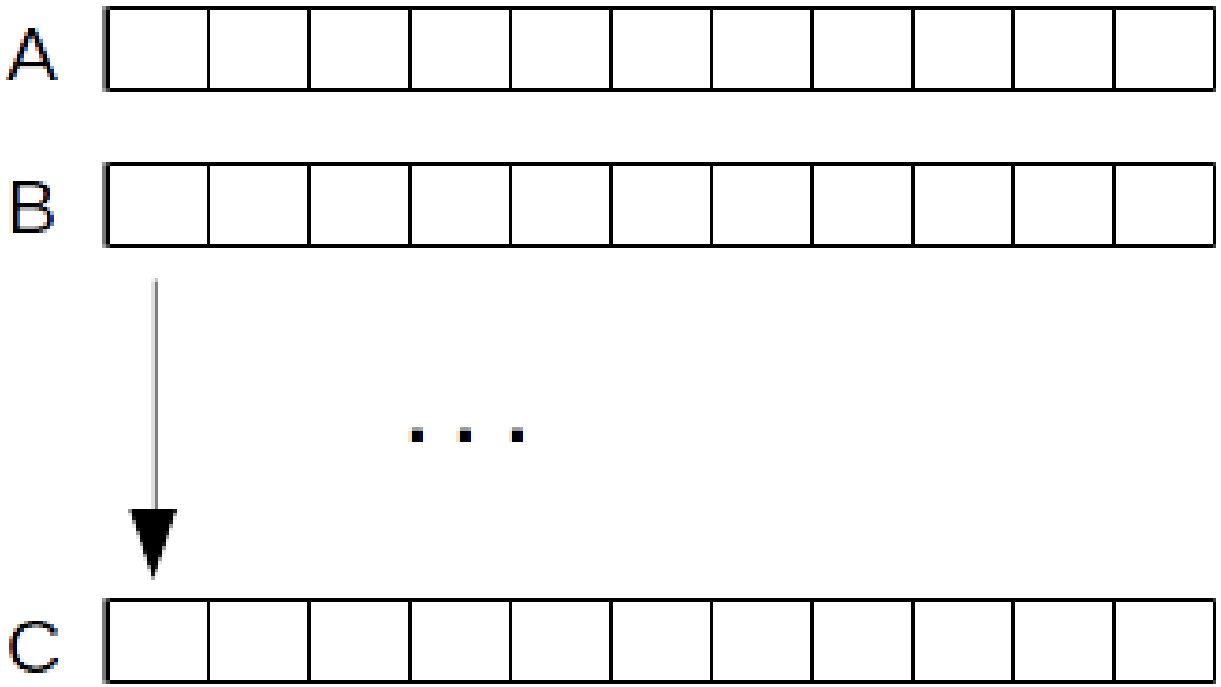
Exemplo inicial (operação SAXPY)

- Considere a operação **C = A * k + B**, sendo A, B e C vetores de tamanho N e k um número
- Para encontrar o vetor de saída C, precisamos calcular cada elemento desse vetor fazendo:
  C[i] = A[i] * k + B[i], $0 <= i < N$

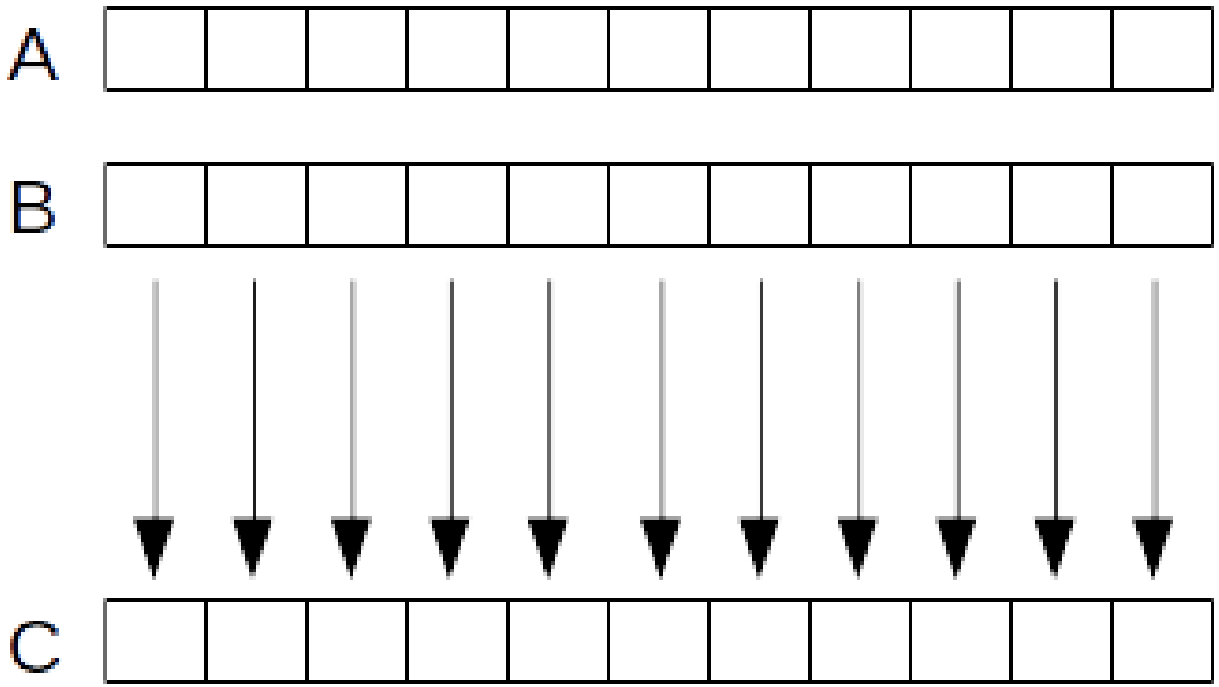
Algoritmo sequencial

```
void somaVetoresSeq(const float a[], const float b[],
                    float c[], float k, int n) {
   for(int i=0; i<n; i++) {
      c[i] = a[i] * k  + b[i];
   }
}
void main() {
   float a[N], b[N], c[N];
   //inicializa os vetores a e b
   ...
   somaVetoresSeq(a, b, c, 2.0, N);
}
```

Algoritmo sequencial



Algoritmo paralelo



Algoritmo paralelo

```
void calculaElementoVetor(const float a[],
     const float b[], float c[], float k, int pos) {
   c[pos] = a[pos] * k + b[pos];
}

void main() {
   float a[N], b[N], c[N];
   //inicializa os vetores a e b
   ...
   //faz C = k * A + B
   for(int i=0; i<N; i++) {
      //dispara um fluxo de execução f para executar:
      //   calculaElementoVetor(a, b, c, 2.0, i)
   }
}
```

Execução sequencial versus execução paralela

O que fizemos foi reduzir a complexidade de processamento de $O(N)$ para $O(1)$, assumindo que N fluxos de execução poderão estar ativos ao mesmo tempo

Ambiente de execução paralela

..mas, se não tivermos processadores em número suficiente para permitir que todos os fluxos de execução estejam ativos ao mesmo tempo?

- em geral, os ambientes de execução paralela se encarregam de "enfileirar" as execuções

- ..ainda assim, vale a pena pensar em outras formas de dividir a tarefa principal?
- quais seriam essas outras formas?

Algoritmo paralelo alternativo

```
void calculaElementosVetor(const float a[],const float b[],
      float c[], float k, int inicio, int salto, int n) {
   int i;
   for(i=inicio; i<n; i=i+salto) {
      c[i] = a[i] * k + b[i];
   }
}
void main() {
   float a[N], b[N], c[N];
   //inicializa os vetores a e b
   ...
   //faz C = k * A + B
   for(int i=0; i<P; i++) {
      //dispara um fluxo de execução f para executar:
       //calculaElementosInterVetor(a, b, c, 2.0, i, P, N);
   }
}
```

Definição de programação paralela

- *Navarro et al.* definem a programação paralela como a tarefa de resolver um problema de tamanho n dividindo o seu domínio em $k \geq 2$ ($k \in \mathbb{N}$) partes que deverão ser executadas em p processadores físicos simultaneamente
- Seguindo essa definição, um problema P_D com domínio D é dito paralelizável se for possível decompor P_D em k subproblemas:

$$D = d_1 \oplus d_2 \oplus \cdots \oplus d_k = \sum_{i=1}^{k} d_i$$

- Dependendo das características do problema, há diferentes formas de realizar essa decomposição

Paralelismo de dados e paralelismo de tarefas

- Dizemos que o problema P_D é um problema com paralelismo de dados se D é composto de elementos de dados e é possível aplicar uma função $f(\cdots)$ para todo o domínio:

$$f(D) = f(d_1) \oplus f(d_2) \oplus \cdots \oplus f(d_k) = \sum_{i=1}^{k} f(d_i)$$

- Por outro lado, se D é composto por funções e a solução do problema consiste em aplicar cada função sobre um fluxo de dados comum, dizemos que o problema P_D é um problema com paralelismo de tarefas:

$$D(S) = d_1(S) \oplus d_2(S) \oplus \cdots \oplus d_k(S) = \sum_{i=1}^{k} d_i(S)$$

Metodologia para projetar algoritmos paralelos

Projetar e implementar algoritmos paralelos não é uma tarefa simples e não há uma regra geral para desenvolver algoritmos paralelos perfeitos

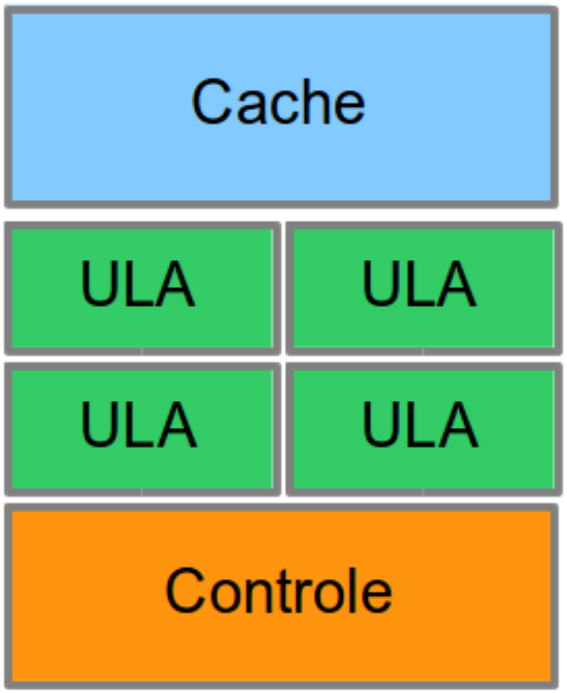
Etapas propostas por Ian Foster

- **Particionamento**: a tarefa que deve ser executada (e o conjunto de dados associado a ela) é decomposta em subtarefas menores
  - o objetivo é identificar oportunidades de execução paralela
- **Comunicação**: determina-se a comunicação requerida para coordenar a execução das subtarefas e define-se as estruturas e algoritmos de comunicação mais apropriados

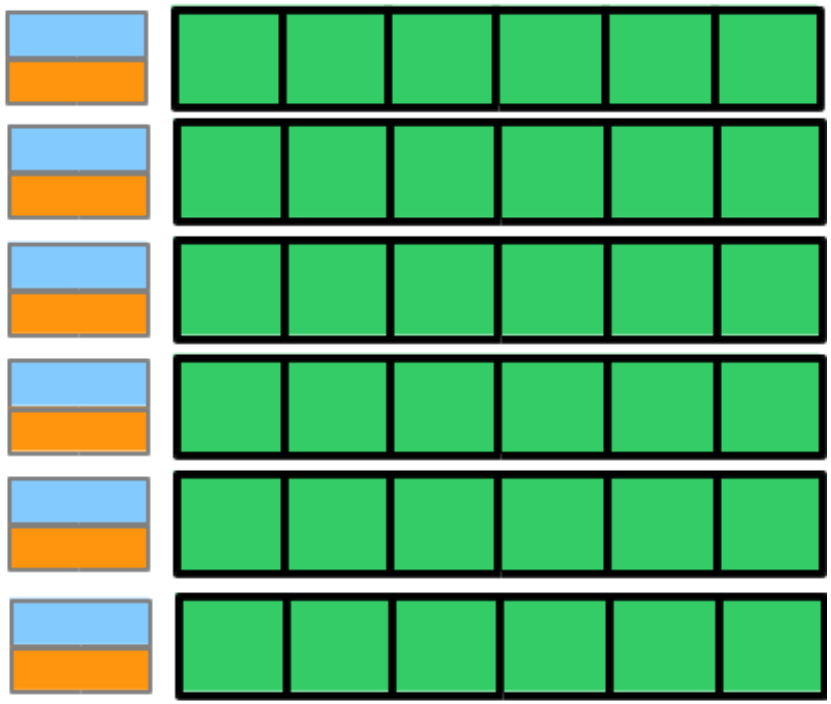
Etapas propostas por Ian Foster

- **Aglomeração**: as subtarefas e estruturas de comunicação são avaliadas com respeito aos requisitos de desempenho e custos de implementação e, se necessário, são combinadas em tarefas maiores
- **Mapeamento**: cada tarefa é designada para uma unidade de processamento com a meta de maximizar o uso da capacidade de processamento paralela disponível e minimizar os custos de comunicação e gerência
  - para essa etapa é fundamental conhecer as características principais da arquitetura alvo

Ambientes de execução paralela



CPU



GPU

CUDA e linguagens

- CUDA foi introduzida em 2006 como uma **plataforma de computação paralela de propósito geral** e um **modelo de programação para GPUs NVIDIA**
- Os códigos que executam na GPU podem ser escritos usando diretamente o conjunto de instruções de máquina de CUDA, chamado **PTX**
- Entretanto, o mais comum é usar linguagens de programação e bibliotecas de nível mais alto, como C, C++ e Fortran
- Também é possível usar CUDA integrado com outros ambientes de programação, como MATLAB e Mathematica

CUDA, bibliotecas e cursos

Há versões CUDA para bibliotecas populares:

- **cuBLAS** (*CUDA Basic Linear Algebra Subroutines*)
- **CUSP** (funções C++ com implementações paralelas para algoritmos de manipulação de matrizes esparsas e de resolução de sistemas lineares esparsos)

- Uma lista de bibliotecas complementares de CUDA pode ser encontrada em https://developer.nvidia.com/gpu-accelerated-libraries
- Diversos cursos sobre programação paralela com CUDA são oferecidos por diferentes universidades https://developer.nvidia.com/educators/existing-courses

Modelo de programação CUDA

Um programa CUDA consiste de uma ou mais fases que são executadas na CPU (chamada *host*) ou na GPU (chamada *device*)

As fases com **pouco ou nenhum paralelismo de dados** são tipicamente executadas na CPU, enquanto as fases com **grande paralelismo de dados** são executadas na GPU

Exemplo inicial: operação SAXPY

```
//função kernel para execução paralela na GPU
__global__ void somaVetoresPar(const float a[],
                const float b[], float c[], float k) {
   int i = threadIdx.x;
   c[i] = a[i] * k + b[i];
}
int main() {
   float a[N], b[N], c[N];
   //inicializa os vetores a e b
   ...
   // invoca o kernel com 1 bloco de N threads
   somaVetoresPar<<<1, N>>>(A, B, C, k);
   ...
}
```

Função kernel

A função **kernel** (precedida por `__global__`) é sempre projetada para ser executada por um **fluxo de execução independente** na GPU (chamado *thread*)

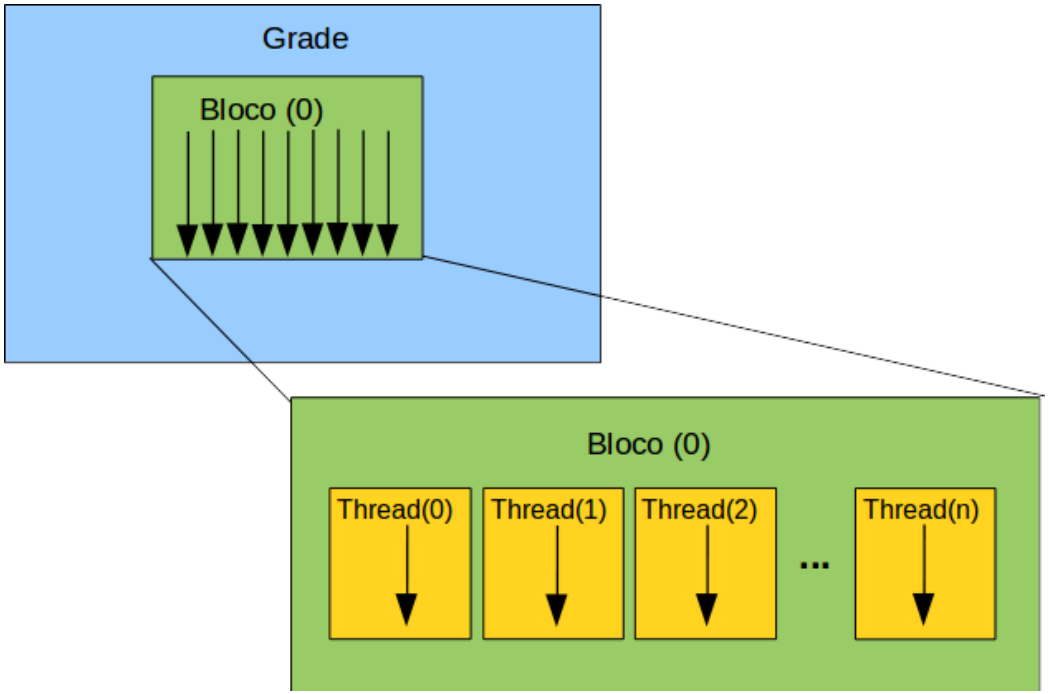
Toda chamada para uma **função kernel** é assíncrona (isto é, retorna para a CPU antes da sua execução ser concluída)

A chamada para a função kernel especifica sua configuração de execução dentro de uma expressão na forma $<<< D_g, D_b, N_s, S >>>$ que precede a sua lista de argumentos

- juntos, D_g e D_b indicam a quantidade de threads que serão criadas ($n_T = D_g * D_b$)
- N_s e S podem ser omitidos

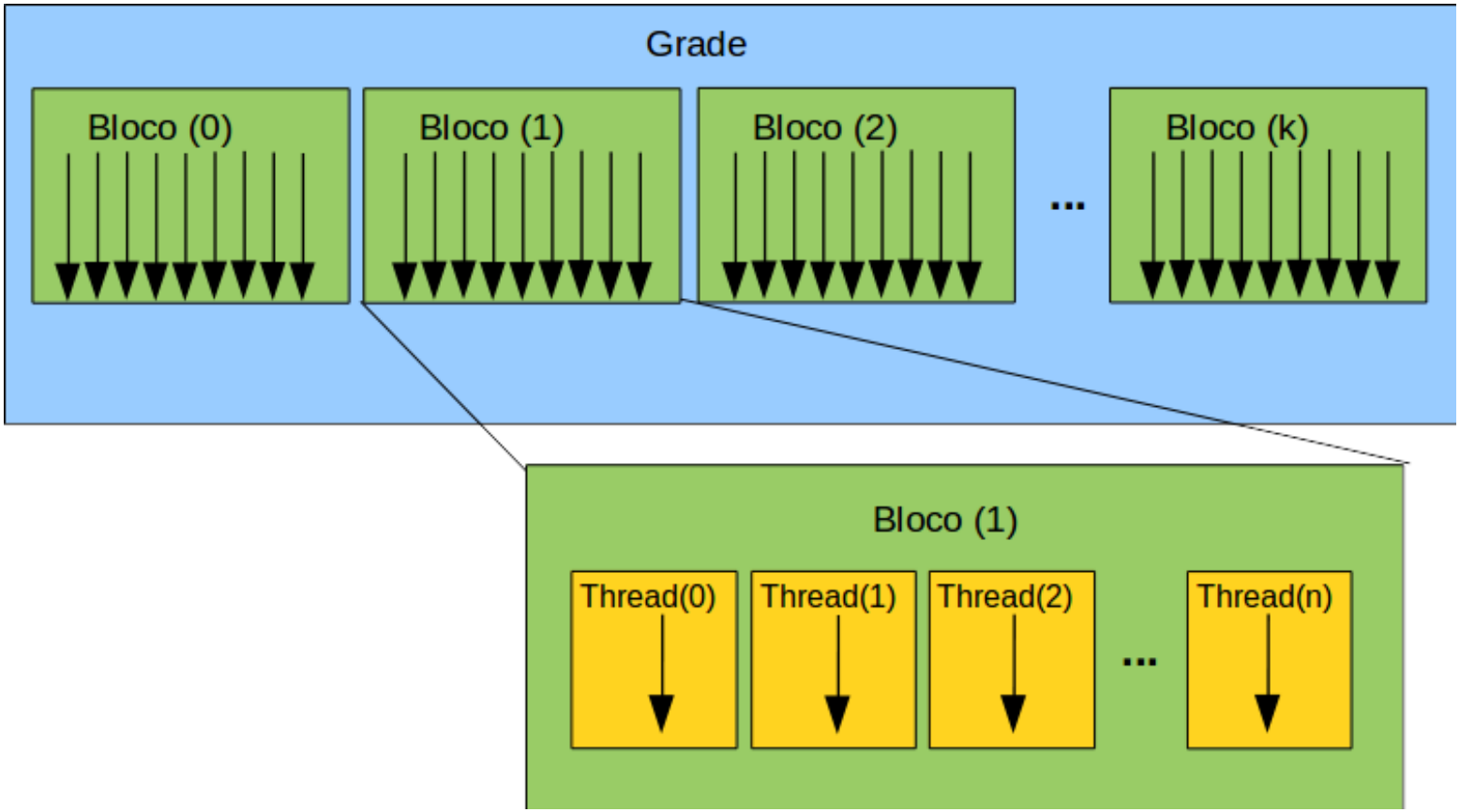
Organização das threads na GPU

As threads na GPU são organizadas em **blocos** e os blocos são organizados em uma **grade**



A chamada para a função kernel retornará com **falha** se D_g ou D_b **forem maiores que o tamanho máximo permitido** (ex., máximo 1024 threads por bloco)

Trabalhando com vários blocos de threads



Usamos o **identificador da thread** (`threadIdx.x`) mais o **identificador do bloco** (`blockIdx.x`) para definir qual elemento do vetor a thread deverá processar. O **tamanho de cada bloco** é dado por `blockDim.x`

Exemplo inicial com vários blocos de threads

```
//kernel para execução paralela na GPU
__global__ void somaVetoresPar(const float a[],
      const float b[], float c[], float k, int n) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if(i < n) {
      c[i] = a[i] * k + b[i];
   }
}
int main() {
   float a[N], b[N], c[N];
   //inicializa os vetores a e b ...
   int n_threads = 1024; //número de threads por bloco
   int n_blocos = (N + n_threads-1)/n_threads;
   somaVetoresPar<<<n_blocos,n_threads>>>(A,B,C,k,N);
   ...
}
```

Transferência de dados para a memória da GPU

Todas as variáveis acessadas pelas threads dentro da função kernel precisam estar armazenadas no espaço de memória da GPU

- Variáveis passadas por referência (ex., vetores A, B e C) precisam apontar para um **endereço de memória válido na memória da GPU** (previamente alocado e carregado)
- Variáveis passadas por valor (ex., k e n) são **automaticamente copiadas para a área de memória local das threads**

Funções para alocação e transferência de dados CPU-GPU

CUDA oferece funções específicas para reservar e liberar espaço de memória na GPU e para transferir dados da CPU para a GPU e vice-versa

- A função **cudaMalloc** aloca uma sequência contínua de bytes na memória da GPU:
  `cudaError_t cudaMalloc (void **devPtr, size_t size)`
- Para transferir/copiar dados entre as memórias da CPU e GPU, CUDA oferece uma única função chamada **cudaMemcpy**:

```
cudaError_t cudaMemcpy(void *dst, const void *src,
                       size_t count, enum cudaMemcpyKind kind)
```

- A função **cudaFree** libera a memória alocada:
  `cudaError_t cudaFree(void *devPtr)`

Tipos de transferências de dados

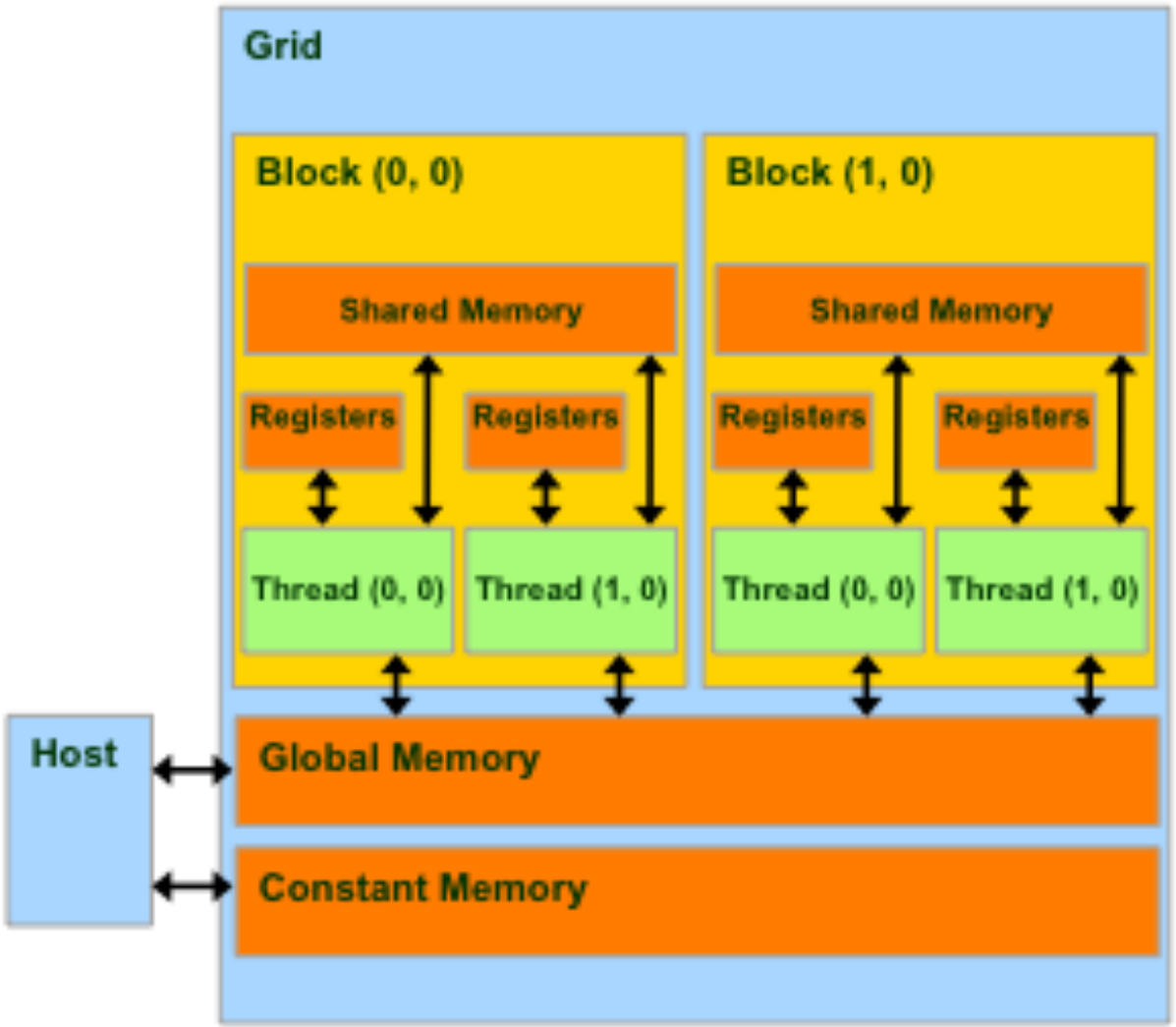
- da CPU (*host*) para a GPU (*device*): `kind` deve receber o valor **cudaMemcpyHostToDevice**
- da GPU para a CPU: `kind` deve receber o valor **cudaMemcpyDeviceToHost**

Hierarquia de memória em CUDA

As threads CUDA acessam diferentes espaços de memória:

- **memória privada**: armazena as variáveis automáticas (aquelas criadas dentro da função kernel)
- **memória compartilhada**: exclusivo de cada bloco e visível para todas as threads do mesmo bloco
- **memória global**: acessada por todas as threads de uma grade e também pela CPU

Organização da memória em CUDA



- R/W registradores (por thread)
- R/W memória local (por thread)
- R/W memória compartilhada (por bloco)
- R/W memória global (por grade)
- R memória constante (por grade)

Código completo em C

Ver código completo em C da operação SAXPY

Referências bibliográficas

1. *Programming massively parallel processors: a hands-on approach*, D. Kirk and W. Hwu Wen-mei, Newnes, 2 ed., 2012
2. *CUDA C Programming Guide* (http://docs.nvidia.com/cuda/cuda-c-programming-guide).
3. *A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures*, C. A. Navarro, N. Hitschfeld-Kahler e L. Mateu, Communications in Computational Physics, vol 15, 2014
4. *Designing and building parallel programs : concepts and tools for parallel software engineering*, I. Foster, Addison-Wesley, 1995