CAPÍTULO

3

Introdução a CUDA

Sumário do Capítulo

Introdução

Para um programador CUDA™, o sistema de computação consiste em um *host* (hospedeiro), que é uma unidade central de processamento (CPU) tradicional, como um microprocessador de arquitetura Intel® nos computadores pessoais de hoje, e um ou mais devices (dispositivos), que são processadores maciçamente paralelos, equipados com uma grande quantidade de unidades de execução aritmética. Nas aplicações de software modernas, as seções do programa normalmente exibem uma rica quantidade de paralelismo de dados, uma propriedade que permite que muitas operações aritméticas sejam seguramente realizadas sobre estruturas de dados do programa de uma maneira simultânea. Os devices CUDA aceleram a execução dessas aplicações, reunindo uma grande quantidade de paralelismo de dados. Por desempenhar um papel tão importante em CUDA, vamos primeiro discutir o conceito de paralelismo de dados antes de introduzirmos as características básicas do CUDA.

3.1 Paralelismo de dados

Muitas aplicações de software que processam uma grande quantidade de dados e, por isso, requerem longos tempos de execução nos computadores de hoje, são projetadas para modelar fenômenos físicos do mundo real. Imagens e quadros de vídeo são retratos de um mundo físico, no qual diferentes partes de uma figura capturam eventos físicos simultâneos e independentes. A física de corpos rígidos e a dinâmica de fluídos modelam forças e movimentos naturais que podem ser avaliados independentemente dentro de pequenos espaços de tempo. Essa avaliação independente é a base do paralelismo de dados nessas aplicações.

Como já dissemos, o paralelismo de dados refere-se à propriedade do programa pela qual muitas operações aritméticas podem ser seguramente realizadas sobre as estruturas de dados de uma maneira simultânea. Ilustramos esse conceito com um exemplo de multiplicação matriz-matriz (multiplicação de matrizes, para abreviar) na Figura 3.1. Nesse exemplo, cada elemento da matriz produto **P** é gerado realizando-se um produto escalar entre uma linha da matriz de entrada **M** e uma coluna da matriz de entrada **N**. Na Figura 3.1, o elemento destacado da matriz **P** é gerado apanhando-se o produto escalar da linha destacada da matriz **M** e a coluna destacada da matriz **N**. Observe que as operações de produto escalar para calcular diferentes elementos da matriz **P** podem ser realizadas simultaneamente. Ou seja, nenhum desses produtos escalares afetará os resultados um do

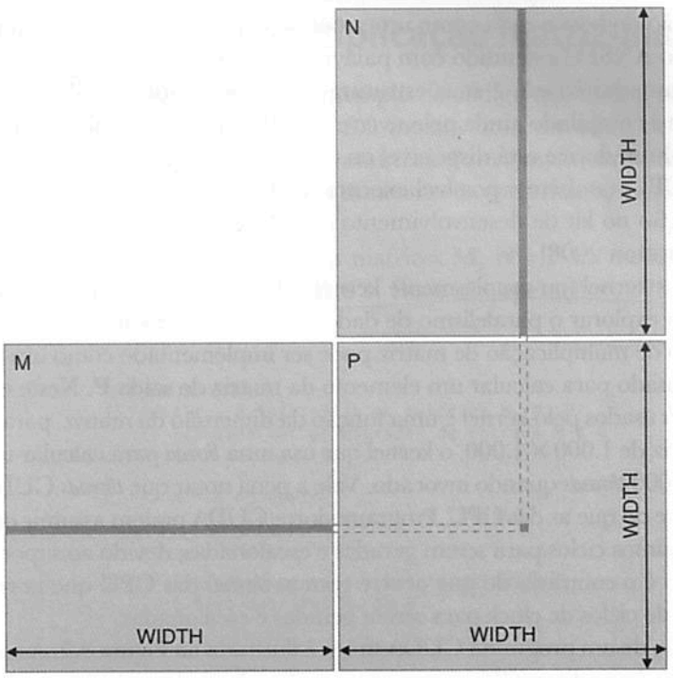

Figura 3.1

Paralelismo de dados na multiplicação de matrizes.

outro. Para matrizes grandes, o número de produtos escalares pode ser também muito grande; por exemplo, uma multiplicação de matrizes de 1.000×1.000 tem 1.000.000 de produtos escalares independentes, cada um envolvendo 1.000 operações aritméticas de multiplicação e 1.000 de acumulação. Dessa maneira, a multiplicação de matrizes de grandes dimensões pode ter uma grande quantidade de paralelismo de dados. Executando muitos produtos escalares em paralelo, um device CUDA pode acelerar significativamente a execução da multiplicação de matrizes em relação a uma CPU hospedeira tradicional. O paralelismo de dados em aplicações reais nem sempre é tão simples quanto no nosso exemplo de multiplicação de matrizes. Em outro capítulo, vamos discutir essas formas mais sofisticadas de paralelismo de dados.

3.2 Estrutura de um programa CUDA

Um programa CUDA consiste em uma ou mais fases que são executadas ou no host (CPU) ou em um device como uma GPU. As fases que exibem pouco ou nenhum paralelismo de dados são implementadas no código do host. Já as que exibem uma rica quantidade de paralelismo de dados são implementadas no código do device. Um programa CUDA é um código fonte unificado compreendendo código do host e código do device. O compilador C da NVIDIA® (nvcc) separa os dois durante o processo de compilação. O código do host é um código ANSI C simples; ele também é compilado com os compiladores C padrão do host e roda como um processo comum da CPU. O código do device é escrito usando ANSI C estendido com palavras-chave para rotular as funções com dados paralelos, chamadas *kernels*, e suas estruturas de dados associadas. O código do device normalmente é compilado ainda pelo nvcc e executado em um device GPU. Em situações nas quais nenhum device está disponível ou o kernel é executado de modo mais apropriado em uma CPU, também é possível executar os kernels em uma CPU usando os recursos de emulação no kit de desenvolvimento de software (SDK) CUDA, ou a ferramenta MCUDA [Stratton 2008].

As funções kernel (ou simplesmente kernels) normalmente geram um grande número de *threads* para explorar o paralelismo de dados. No exemplo de multiplicação de matriz, o cálculo inteiro de multiplicação de matriz pode ser implementado como um kernel no qual cada *thread* é usado para calcular um elemento da matriz de saída **P**. Neste exemplo, o número de *threads* usados pelo kernel é uma função da dimensão da matriz. para uma multiplicação de matriz de 1.000×1.000, o kernel que usa uma *thread* para calcular um elemento **P** geraria 1.000.000 *threads* quando invocado. Vale a pena notar que *threads* CUDA são de peso muito mais leve do que as das CPU. Programadores CUDA podem assumir que essas *threads* usam pouquíssimos ciclos para serem geradas e escalonadas, devido ao suporte eficiente do hardware. Isso é o contrário do que ocorre com as *threads* das CPU que normalmente exigem milhares de ciclos de clock para serem geradas e escalonadas.

A execução de um programa CUDA típico é ilustrada na Figura 3.2. A execução começa com a execução no host (CPU). Quando uma função do kernel é invocada, ou *disparada*, a execução é passada para um device (GPU), no qual um número maior de *threads* é gerado para tirar proveito do abundante paralelismo de dados. Todas as *threads* geradas por um ker-

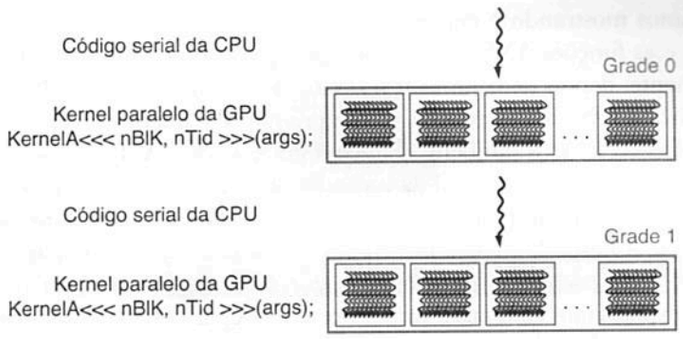

Figura 3.2

Execução de um programa CUDA.

nel durante uma chamada são conhecidas coletivamente como *grid*. A Figura 3.2 mostra a execução de duas grades de *threads*. Em breve, discutiremos como essas grades são organizadas. Quando todas as *threads* de um kernel completam sua execução, a grade correspondente termina e a execução continua no host até que outro kernel seja invocado.

3.3 Um exemplo de multiplicação matriz-matriz

Neste ponto, é útil apresentarmos um exemplo de código que ilustra concretamente a estrutura do programa CUDA. A Figura 3.3 mostra uma estrutura de função main simples para o exemplo de multiplicação de matrizes. Para simplificar, vamos supor que as matrizes sejam quadradas em formato, e a dimensão de cada matriz é especificada pelo parâmetro `Width`.

O programa principal primeiro aloca as matrizes **M**, **N** e **P** na memória do host e depois realiza a E/S para ler **M** e **N** na Parte 1. Essas são operações ANSI C, de modo

```
int main(void) |
1. // Aloca e inicializa as matrizes M, N, P
   // E/S para ler as matrizes de entrada M e N
....

2. // M * N no device
   MatrixMultiplication(M, N, P, Width);


3. // E/S para escrever a matriz de saída P
   // Libera matrizes M, N, P
...
return 0;
}
```

Figura 3.3

Uma função main simples para o exemplo de multiplicação de matrizes.

que não estamos mostrando o código real para poder abreviar. O código detalhado da função main e as funções ANSI C definidas pelo usuário aparecem no Apêndice A. De modo semelhante, depois de completar a multiplicação de matrizes, a Parte 3 da função principal realiza a E/S para escrever a matriz produto **P** e liberar todas as matrizes alocadas. Os detalhes da Parte 3 também aparecem no Apêndice A. E, depois de completar a multiplicação de matrizes, a Parte 3 da função main realiza a E/S para escrever a matriz produto **P** e liberar todas as matrizes alocadas. Os detalhes da Parte 3 também aparecem no Apêndice A. A Parte 2 é o foco principal do nosso exemplo. Ela chama uma função, `MatrixMultiplication()`, para realizar multiplicação de matrizes em um device.

Antes de explicarmos como usar um device CUDA para executar a função de multiplicação de matriz, é útil primeiro rever como ela funciona apenas para a CPU convencional. Uma versão simples aparece na Figura 3.4. A função `MatrixMultiplication()` implementa um algoritmo simples, que consiste em três níveis de loop. O mais interno percorre a variável **k** e passa por uma linha da matriz **M** e uma coluna da matriz **N**. O loop calcula um produto escalar da linha de **M** e a coluna de **N** e gera um elemento de **P**. Imediatamente após o loop mais interno, o elemento `P` gerado é escrito na matriz de saída **P**.

O índice usado para acessar a matriz **M** no loop mais interno é `i*Width+k`. Isso porque os elementos da matriz **M** são colocados na memória do sistema que por fim é acessada com um endereço linear. Ou seja, cada local na memória do sistema tem um endereço que varia de 0 até o maior local da memória. Para programas em C, o posicionamento de uma matriz bidimensional nessa memória endereçada linearmente é feito de acordo com a convenção de linha primeiro, conforme ilustrado na

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         float sum = 0;
         for (int k = 0; k < Width; ++k) {
            float a = M[i * width + k];
            float b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```

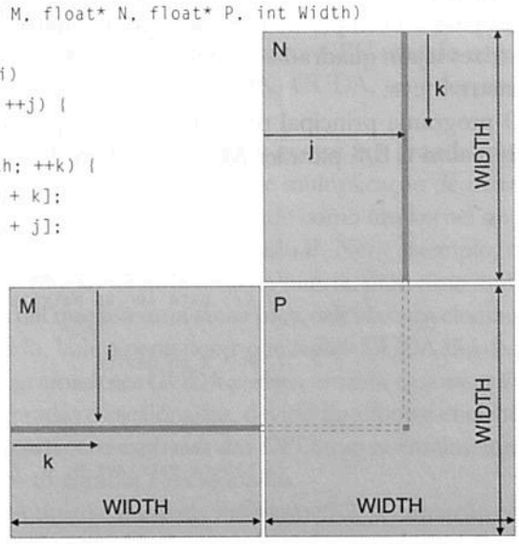

Figura 3.4

Uma função de multiplicação de matrizes simples contendo apenas o código host

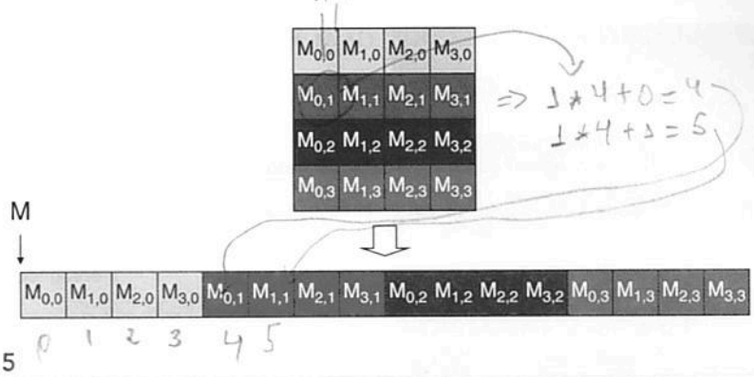

Figura 3.5

Arranjo de elementos de matriz bidimensional na memória do sistema com endereços lineares.

Figura 3.5.[1] Todos os elementos em uma linha são colocados em locais de memória consecutivos. As linhas são, então, colocadas uma após a outra. A Figura 3.5 mostra um exemplo em que uma matriz de 4×4 é colocada em 16 locais consecutivos, com todos os elementos da linha 0 primeiro, seguidos pelos quatro elementos da linha 1 etc. Portanto, o índice para um elemento **M** na linha *i* e coluna *k* é `i*Width+k`. O termo `i*Width` salta por todos os elementos das linhas antes da linha *i*. O termo *k* então seleciona o elemento apropriado dentro da seção para a linha *i*.

Os dois loops mais externos (*i* e *j*) na Figura 3.4 percorrem juntamente todas as linhas de **M** e todas as colunas de **N**; cada iteração conjunta realiza um produto escalar linha-coluna para gerar um elemento **P**. Cada valor de *i* identifica uma linha. Percorrendo sistematicamente todas as linhas de **M** e todas as colunas de **N**, a função gera todos os elementos de **P**. Agora, temos uma função e multiplicação de matriz completa, que executa unicamente na CPU. Observe que todo o código que mostramos até aqui está em C padrão.

Suponha que um programador agora queira portar a função de multiplicação de matriz para CUDA. Um modo simples de fazer isso é modificar a função `MatrixMultiplication()` para mover o núcleo do cálculo para um device CUDA. A estrutura da função revisada aparece na Figura 3.6. A Parte 1 da função aloca a memória do device (GPU) para manter cópias das matrizes **M**, **N** e **P** e cópias dessas matrizes para a memória do device. A Parte 2 invoca um kernel que dispara a execução paralela da multiplicação de matriz real no device. A Parte 3 copia a matriz produto **P** da memória do device para a memória do host.

Observe que a função `MatrixMultiplication()` revisada é basicamente um agente de terceirização que envia dados de entrada para um device, ativa o cálculo no device e coleta os resultado do device. O agente faz isso de modo que o programa principal não precisa sequer saber que a multiplicação de matrizes agora é realmente feita em um device. Os detalhes da função revisada, além do modo de compor a função do kernel, servirão como ilustrações quando introduzirmos as características básicas do modelo de programação CUDA.

[1] Observe que o FORTRAN adota a técnica de posicionamento de coluna primeiro: Todos os elementos de uma coluna são primeiro posicionados em locais consecutivos, e todas as colunas são então colocadas em sua ordem numérica.

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Aloca memória do device para M, N e P
   // Copia M e N para todos os locais de memória do device alocado

2. // Código de chamada do kernel - para que o device realize
   // a multiplicação de matriz real

3. // Copia P da memória do device
   // Libera matrizes do device

}
```

Figura 3.6

Esboço de um código de host `MatrixMultiplication()` revisado, que move a multiplicação de matrizes para um device.

3.4 Memórias de device e transferência de dados

Em CUDA, o host e os devices possuem espaços de memória separados. Isso reflete a realidade de que os devices normalmente são placas de hardware que vêm com sua própria memória de acesso aleatório dinâmica (DRAM). Por exemplo, o processador NVIDIA T10 vem com até 5 GB (bilhões de bytes, ou gigabytes) de DRAM. Para executar um kernel em um device, o programador precisa alocar memória no device e transferir dados pertinentes da memória do host para a memória do device alocado. Isso corresponde à Parte 1 da Figura 3.6. De modo semelhante, após a execução no device, o programador precisa transferir os dados de resultado da memória do device de volta à memória do host e liberar a memória do device que não é mais necessária. Isso corresponde à Parte 3 da Figura 3.6. O sistema de *runtime* CUDA oferece funções da interface de programação de aplicação (API) para realizar essas atividades em favor do programador. Deste ponto em diante, vamos simplesmente dizer que uma parte dos dados é transferida do host para o device, como uma abreviação para dizer que a parte dos dados é transferida da memória do host para a memória do device. O mesmo pode ser dito para a direção oposta na transferência de dados.

A Figura 3.7 mostra uma visão geral do modelo de memória de device CUDA para os programadores raciocinarem sobre a alocação, movimentação e uso dos diversos tipos de memória de um device. Na parte de baixo da figura, vemos a memória global e a memória constante. Essas são as memórias que o código do host pode transferir *de* e *para* o device, conforme ilustrado pelas setas bidirecionais entre essas memórias e o host. A memória constante permite o acesso apenas de leitura pelo código do device e é descrita no Capítulo 5. Por enquanto, vamos focalizar o uso da memória global. Observe que a memória do host não é mostrada explicitamente na Figura 3.7, mas é considerada como estando contida no host.[2] O modelo de memória CUDA é aceito pelas funções

[2] Observe que omitimos a memória de textura da Figura 3.7 para simplificar. Vamos apresentar a memória de textura mais adiante.

- O código do device pode:
  - L/E registradores por thread
  - L/E memória local por thread
  - L/E memória compartilhada por bloco
  - L/E memória global por grade
  - Apenas ler memória constante por grade
- O código do host pode:
  - Transferir dados de/para memórias global e constante por grade

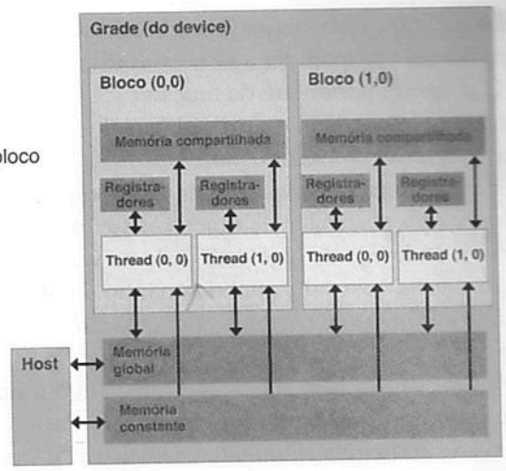

Figura 3.7

Visão geral do modelo de memória do device CUDA.

da API que ajudam os programadores CUDA a gerenciar dados nessas memórias. A Figura 3.8 mostra as funções da API para alocar e desalocar a memória global do device. A função `cudaMalloc()` pode ser chamada pelo código do host para alocar uma parte da memória global para um objeto. O leitor deverá ser capaz de observar a grande semelhança entre `cudaMalloc()` e a função `malloc()` da biblioteca de runtime C padrão. Isso é intencional; CUDA é C com extensões mínimas. CUDA utiliza a função `malloc()` da biblioteca de *runtime* C padrão para gerenciar a memória do host e acrescenta `cudaMalloc()` como uma extensão à biblioteca de *runtime* C. Mantendo a interface o mais próximo possí-

- cudaMalloc()
  - Aloca objeto na memória global do device
  - Dois parâmetros
    - **Endereço de um ponto** no objeto alocado
    - **Tamanho** do objeto alocado em termos de bytes
- cudaFree()
  - Libera objeto da memória global do device
    - Ponteiro para objeto liberado

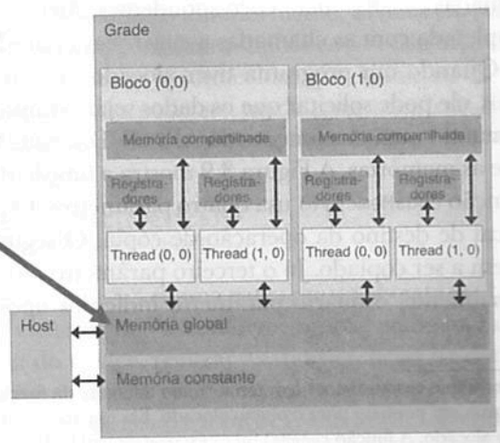

Figura 3.8

Funções de API CUDA para gerenciamento da memória global do device.

vel das bibliotecas de *runtime* C originais, CUDA minimiza o tempo que um programador C precisa para reaprender a usar essas extensões.

O primeiro parâmetro da função `cudaMalloc()` é o endereço de uma variável de ponteiro que precisa apontar para o objeto alocado após a tarefa de alocação. O endereço da variável deve ser convertido para `(void **)`, porque a função espera um valor de ponteiro genérico; a função de alocação de memória é uma função genérica que não é restrita a qualquer tipo particular de objetos. Esse endereço permite que a função `cudaMalloc()` escreva o endereço do objeto alocado na variável de ponteiro.[3] O segundo parâmetro da função `cudaMalloc()` diz qual o tamanho do objeto a ser alocado, em bytes. O uso desse segundo parâmetro é coerente com o parâmetro de tamanho da função `malloc()` da linguagem C.

Agora, vamos usar o exemplo de código ilustrando o uso de `cudaMalloc()`. Essa é a continuação do exemplo na Figura 3.6. Para tornar mais claro, vamos nomear uma variável de ponteiro terminando com a letra "d", para indicar que a variável é usada para apontar para um objeto no espaço de memória do device. O programador passa o endereço de **Md** (ou seja, `&Md`) como primeiro parâmetro após convertê-lo para um ponteiro void; ou seja, **Md** é o ponteiro que aponta para a região de memória global do device alocada para a matriz **M**. O tamanho da matriz alocada será `Width*Width*4` (o tamanho de um número de ponto flutuante com precisão simples). Depois do cálculo, `cudaFree()` é chamada com o ponteiro **Md** como entrada para liberar o espaço de armazenamento para a matriz **M** da memória global do device:

```
float *Md
int size = Width * Width * sizeof(float);
cudaMalloc((void**)&Md, size);
...
cudaFree(Md);
```

O leitor deverá completar a Parte 1 do exemplo de `MatrixMultiplication()` na Figura 3.6 com declarações semelhantes de variáveis de ponteiro **Nd** e **Pd**, bem como suas chamadas `cudaMalloc()` correspondentes. Além do mais, a Parte 3 na Figura 3.6 pode ser completada com as chamadas a `cudaFree()` para **Nd** e **Pd**.

Quando um programa tiver alocado memória global de device para os objetos de dados, ele pode solicitar que os dados sejam transferidos do host para o device. Isso é feito chamando uma das funções da API CUDA, `cudaMemcpy()`, para a transferência de dados entre as memórias. A Figura 3.9 mostra a função da API para essa transferência de dados. A função `cudaMemcpy()` usa quatro parâmetros. O primeiro parâmetro é um ponteiro para o local de destino da operação de cópia. O segundo aponta para o objeto de dados de origem a ser copiado. Já o terceiro parâmetro especifica o número de bytes a ser copiado e, finalmente, o quarto parâmetro indica os tipos de memória envolvidos na cópia: da

[3] Observe que `cudaMalloc()` tem um formato diferente da função `malloc()` da C. A função `malloc()` da C retorna um ponteiro para o objeto alocado. Ela usa apenas um parâmetro que especifica o tamanho do objeto alocado. A função `cudaMalloc()` escreve na variável de ponteiro cujo endereço é dado como o primeiro parâmetro. Como resultado, a função `cudaMalloc()` utiliza dois parâmetros. O formato de dois parâmetros de `cuda-Malloc()` permite que ela utilize o valor de retorno para informar quaisquer erros da mesma forma que as outras funções da API CUDA.

- cudaMemcpy()
  - Transferência de dados da **memória**
  - Requer quatro parâmetros
    - Ponteiro para destino
    - Ponteiro para origem
    - Número de byts copiados
    - Tipo de transferência
      - Host para host
      - Host para device
      - Device para host
      - Device para device
  - Transferência é assíncrona

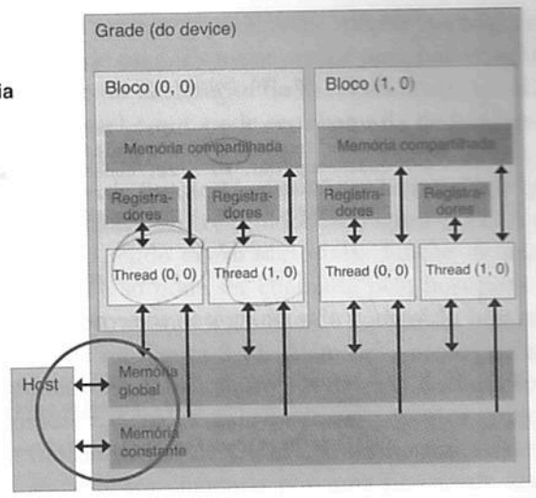

Figura 3.9

Funções da API CUDA para transferência de dado entre as memórias.

memória do host para a memória do host, da memória do host para a memória do device, da memória do device para a memória do host e da memória do device para a memória do device. Por exemplo, a função de cópia da memória pode ser usada para copiar dados de um local da memória do device para outro local da memória do device. Por favor, observe que `cudaMemcpy()` não pode ser usada para copiar dados entre diferentes GPUs em sistemas com GPUs múltiplas.

Para o exemplo de multiplicação de matriz, o código do host chama a função `cudaMemcpy()` para copiar as matrizes **M** e **N** da memória do host para a memória do device antes da multiplicação e depois para copiar a matriz **P** da memória do device para a memória do host, depois que a multiplicação terminar. Suponha que **M**, **P**, **Md**, **Pd** e o tamanho já tenham sido definidos conforme discutimos anteriormente; as duas chamadas de função aparecem a seguir. observe que as duas constantes simbólicas, `cudaMemcpyHostToDevice` e `cudaMemcpyDeviceToHost`, são constantes reconhecidas, predefinidas, do ambiente de programação CUDA. A mesma função pode ser usada para transferir dados nas duas direções, ordenando corretamente os ponteiros de origem e destino e usando a constante apropriada ao tipo de transferência:

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

Resumindo, o programa principal na Figura 3.3 chama `MatrixMultiplication()`, que também é executado no host. `MatrixMultiplication()`, conforme representado na Figura 3.6, é responsável por alocar a memória do device, realizar as transferências de dados e ativar o kernel que realiza a multiplicação de matriz real. Normalmente, nos referimos a esse trecho de código do host como uma *sub function* para invocar um kernel. Depois da multiplicação de matriz, `MatrixMultiplication()` também copia os dados de resultado do device para o host. Mostramos uma versão mais encorpada da função `MatrixMultiplication()` na Figura 3.10.

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;

1.  // Transfere M e N para a memória do device
    cudaMalloc((void**) &Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevide);
    cudaMalloc((void**) &Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevide);

    // Aloca P no device
    cudaMalloc((void**) &Pd, size);

2.  // Código de chamada do kernel - mostrado mais adiante
    ...

3.  // Transfere P do device para o host
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
    // Libera matrizes no device
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
    }
```

Figura 3.10

A função MatrixMultiplication() revisada.

Em comparação com a Figura 3.6, a função `MatrixMultiplication()` revisada está completa na Parte 1 e na Parte 3. A Parte 1 aloca a memória de device para **Md**, **Nd** e **Pd**, os correspondentes no device de **M**, **N** e **P**, e transfere **M** para **Md** e **N** para **Nd**. Isso é feito com chamadas às funções `cudaMalloc()` e `cudaMemcpy()`. Encorajamos o leitor a escrever suas próprias chamadas de função com os valores de parâmetro apropriados e comparar seu código com o que aparece na Figura 3.10. A Parte 2 chama o kernel e será descrita no texto a seguir. A Parte 3 lê os dados do produto da memória do device para a memória do host, de modo que o valor esteja disponível para `main()`. Isso é feito com uma chamada para a função `cudaMemcpy()`. Depois, ela libera **Md**, **Nd** e **Pd** da memória do device, o que é feito com chamadas para as funções `cudaFree()`.

3.5 Funções do kernel e threading

Agora, estamos prontos para discutir mais sobre as funções do kernel CUDA e o resultado de chamar essas funções do kernel. Em CUDA, uma função do kernel especifica o código a ser executado por todas as *threads* durante uma etapa paralela. Como todas essas *threads* executam o mesmo código, a programação CUDA é uma caso do conhecido estilo de programação de único programa e múltiplos dados (SPMD — Single-Program, Multiple-Data) [Atallah 1998], um estilo de programação popular para sistemas de computador maciçamente paralelos.[4]

[4] Observe que SPMD não é a mesma coisa que única instrução, múltiplos dados (SIMD). Em um sistema SPMD, as unidades de processamento paralelo executam o mesmo programa sobre várias partes dos dados; essas unidades de processamento não precisam estar executando a mesma instrução ao mesmo tempo. Em um sistema SIMD, todas as unidades de processamento estão executando a mesma instrução em determinado instante.

A Figura 3.11 mostra a função do kernel para multiplicação de matriz. A sintaxe está em ANSI C com algumas extensões dignas de nota. Primeiro, existe uma palavra-chave específica do CUDA "__global__" na frente da declaração da `MatrixMulKernel()`. Essa palavra-chave indica que a função é um kernel e que pode ser chamada de funções do host para gerar uma grade de *threads* em um device.

Em geral, CUDA estende as declarações de função C com três palavras-chave qualificadoras. Os significados dessas palavras-chave são resumidos na Figura 3.12. A palavra-chave `__global__` indica que a função sendo declarada é uma função de kernel CUDA. A função será executada no device e só pode ser chamada pelo host para gerar uma grade de *threads* em um device. Mostraremos a sintaxe do código do host para chamar uma função de kernel mais adiante na Figura 3.14. Além de `__global__`, existem duas outras palavras-chave que podem ser usadas na frente de uma declaração de função. A Figura 3.12 resume o significado delas. A palavra-chave `__device__` indica que a função sendo declarada é uma função de device CUDA. Uma função de device é executada em um device CUDA e só pode ser chamada por uma função de kernel ou outra função de device. As funções de device não podem conter nem chamadas de função recursivas nem chamadas de função indiretas através de ponteiros. A palavra-

```
// Kernel para multiplicação de matriz - especificação de thread
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // ID do Thread 2D
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue armazena o elemento Pd que é calculado pela thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
      float Mdelement = Md[ty * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
    }

    // Escreve a matriz na memória do device, cada thread escreve um elemento
    Pd[ty * Width + tx] = Pvalue;
}
```

Figura 3.11

A função de kernel para multiplicação de matriz.

	Executada no:	Só é chamada pelo:
`__device__float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Figura 3.12

Extensões CUDA às declarações das funções do C.

-chave `__host__` indica que a função sendo declarada é uma função de host CUDA. Uma função de host é simplesmente uma função C tradicional que é executada no host e só pode ser chamada por outra função de host. Como padrão, todas as funções em um programa CUDA são funções de host se não tiverem uma das palavras-chave CUDA em sua declaração. Isso faz sentido, pois muitas aplicações CUDA são originárias de ambientes de execução de CPU. O programador acrescentaria funções de kernel e funções de device durante o processo de adaptação. As funções originais permanecem como funções de host. Com todas as funções sendo consideradas como funções de host como padrão, o programador não precisa ter o trabalho de alterar todas as declarações de função originais.

Observe que é possível usar ao mesmo tempo `__host__` e `__device__` em uma declaração de função. Essa combinação dispara o sistema de compilação para gerar duas versões da mesma função. Uma é executada no host e só pode ser chamada por uma função de host. A outra é executada no device e só pode ser chamada por uma função de device ou de kernel. Isso dá suporte a um uso comum quando o mesmo código fonte da função pode ser simplesmente recompilado para gerar uma versão de device. Muitas funções da biblioteca do usuário provavelmente estarão nessa categoria.

Outras extensões dignas de nota da ANSI C, na Figura 3.11, são as palavras-chave `threadIdx.x` e `threadIdx.y`, que referem-se aos índices de uma *thread*. Observe que todas as *threads* executam o mesmo código do kernel. É preciso haver um mecanismo para permitir que elas sejam distinguidas e se dirijam para as partes específicas da estrutura de dados sobre as quais foram preparadas para atuar. Essas palavras-chave identificam variáveis pré-definidas que permitem que uma *thread* acesse os registradores do hardware em *runtime* que oferecem as coordenadas de identificação para uma *thread* específica. Diferentes *threads* verão diferentes valores nas variáveis `threadIdx.x` e `threadIdx.y`. Para simplificar, vamos nos referir a uma *thread* como $Thread_{\text{threadIdx.x, threadIdx.y}}$. Observe que as coordenadas refletem uma organização multidimensional para as *threads*. Retornaremos a esse assunto em breve.

Uma comparação rápida entre as Figura 3.4 e 3.11 revela um detalhe importante para as funções de kernel CUDA e a chamada do kernel CUDA. A função de kernel na Figura 3.11 tem apenas um loop, que corresponde ao loop mais interno na Figura 3.4. Os leitores deverão estar se perguntando onde se encontram os outros dois níveis de loop mais externos. A resposta é que os dois níveis de loop mais externos agora são substituídos pela grid de *threads*. A grid inteira forma o equivalente do loop de dois níveis. Cada *thread* na grade corresponde a uma das iterações do loop original de dois níveis. As variáveis do loop original, *i* e *j*, agora são substituídas por *threadIdx.x* e *threadIdx.y*. Ao invés de fazer com que o loop incremente os valores de *i* e *j* para uso em cada iteração de loop, o hardware de *threading* CUDA gera todos os valores de *threadIdx.x* e *threadIdx.y* para cada *thread*.

Na Figura 3.11, cada *thread* usa seu *threadIdx.x* e *threadIdx.y* para identificar a linha de **Md** e a coluna de **Nd** para realizar a operação de produto escalar. Deve ficar claro que esses índices simplesmente assumem o papel das variáveis *i* e *j* na Figura 3.8. Observe que atribuímos *threadIdx.x* à variável *tx* do C e *threadIdx.y* à variável *ty* para abreviar, na Figura 3.8. Cada *thread* também usa seus valores de *threadIdx.x* e *threadIdx.y* para selecionar o elemento de **Pd** pelo qual é responsável; por exemplo, $Thread_{2,3}$ realizará um produto escalar entre a

coluna 2 de **Nd** e a linha 3 de **Md**, escrevendo o resultado no elemento (2,3) de **Pd**. Desse modo, as *threads* geram coletivamente todos os elementos da matriz **Pd**.

Quando um kernel é chamado, ou *disparado*, ele é executado como uma *grade* de *threads* paralelas. Na Figura 3.13, o disparo do Kernel 1 cria a Grade 1. Cada grade de *threads* CUDA normalmente é composta de milhares a milhões de *threads* GPU por chamada do kernel. A criação de *threads* que sejam capazes de utilizar o hardware completamente costuma exigir uma grande quantidade de paralelismo de dados; por exemplo, cada elemento de uma matriz grande poderia ser calculado em uma *thread* separada.

As *threads* em uma grade são organizados em uma hierarquia de dois níveis, conforme ilustra a Figura 3.13. Para simplificar, um pequeno número de *threads* aparece na Figura 3.13. Na realidade, uma grade normalmente terá muito mais *threads*. No nível mais alto, cada grade consiste em um ou mais blocos de *threads*. Todos os blocos em uma grade têm o mesmo número de *threads*. Na Figura 3.13, a Grade 1 é organizada como uma matriz 2×2 com 4 blocos. Cada bloco tem uma coordenada bidimensional exclusiva, dada pelas palavras-chave CUDA específicas `blockIdx.x` e `blockIdx.y`. Todos os blocos de *threads* precisam ter o mesmo número de *threads* organizadas da mesma maneira.

Cada bloco de *threads*, por sua vez, é organizado como uma matriz tridimensional de *threads*, com um tamanho total de até 512 *threads*. As coordenadas das *threads* em um bloco são definidas de modo exclusivo por três índices de *thread*: `threadIdx.x`, `threadIdx.y` e `threadIdx.z`. Nem todas as aplicações usarão todas as três dimensões de um bloco de *threads*. Na Figura 3.13, cada bloco de *thread* é organizado em uma matriz tridimensional 4×2×2 de *threads*. Isso dá à Grade 1 um total de 4×16 = 64 *threads*. Este, obviamente, é um exemplo bastante simplificado.

No exemplo de multiplicação de matrizes, uma grade é chamada para calcular a matriz produto. O código da Figura 3.11 não usa qualquer índice de bloco no acesso aos dados de

- Um bloco de threads é um lote de threads que podem cooperar entre si:
  - Sincronizando sua execução
    - Para acessos à memória compartilhada sem restrições
  - Compartilhando dados de modo eficiente por uma memória compartilhada com pouco latência
- Duas threads de dois blocos diferentes não podem cooperar

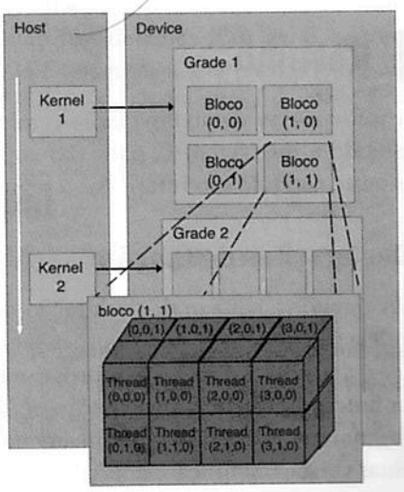

Figura 3.13

Organização de threads em CUDA.

```
// Monta a configuração para execução
dim3 dimBlock(Width, Width);
dim3 dimGrid(1,1);

// Dispara as threads de cálculo no device!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Figura 3.14

Exemplo de código de host que dispara um kernel.

entrada e saída. As *threads* com os mesmos valores de `threadIdx` de diferentes blocos acabariam acessando os mesmos elementos de dados de entrada e saída. Como resultado, o kernel só pode usar um bloco de *threads*. Os valores de `threadIdx.x` e `threadIdx.y` são usados para organizar o bloco em uma matriz de *threads* bidimensional. Como o bloco de *threads* só pode ter até 512 *threads*, e cada *thread* calcula um elemento da matriz produto na Figura 3.11, o código só pode calcular uma matriz produto de até 512 elementos; obviamente, isso não é aceitável. Conforme já explicamos, a matriz produto poderá ter milhões de elementos a fim de ter uma quantidade suficiente de paralelismo de dados para se beneficiar da execução em um device. Voltaremos a essa questão no Capítulo 4, usando múltiplos blocos.

Quando o código do host chama um kernel, ele define a grade e as dimensões do bloco de *threads* por meio de parâmetros de *configuração de execução*. Isso é ilustrado na Figura 3.14. Duas variáveis `struct` de tipo `dim3` são declaradas. A primeira é para descrever a configuração dos blocos, que são definidos como grupos de 16×16 *threads*. A segunda variável, `dimGrid`, descreve a configuração da grade. Neste exemplo, temos apenas um bloco (1×1) em cada grade. A linha de código final chama o kernel. A sintaxe especial entre o nome da função de kernel e os parâmetros tradicionais da função no C é uma extensão CUDA à ANSI C. Ela fornece as dimensões da grade em termos do número de blocos e a dimensões dos blocos em termos do número de *threads*.

3.6 Resumo

Este capítulo serve como uma visão geral rápida do modelo de programação CUDA. CUDA estende a linguagem C para dar suporte à computação paralela. As extensões discutidas neste capítulo são resumidas a seguir.

3.6.1 Declarações de função

CUDA estende a sintaxe de declaração de função em C para dar suporte à computação paralela heterogênea. As extensões são resumidas na Figura 3.12. Usando `__global__`, `__device__` ou `__host__`, um programador CUDA pode instruir o compilador a gerar uma função de kernel, uma função de device ou uma função de host. Se `__host__` e `__device__` forem utilizados simultaneamente em uma declaração de função, o compilador gera duas versões da função, uma para o device e uma para o host. Se uma declaração de função não tiver qualquer palavra-chave de extensão CUDA, a função é considerada uma função de host como padrão.