NOTAS EM MATEMÁTICA APLICADA e-ISSN 2236-5915

Volume 84, 2016

Editores

Jorge Manuel Vieira Capela
Universidade Estadual Paulista - UNESP
Araraquara, SP, Brasil

Edson Luiz Cataldo Ferreira
Universidade Federal Fluminense - UFF
Niterói, RJ, Brasil

Alexandre Loureiro Madureira (Editor Chefe)
Laboratório Nacional de Computação Científica - LNCC
Petrópolis, RJ, Brasil

Amanda Liz Pacífico Manfrim Perticarrari
Universidade Estadual Paulista Júlio de Mesquita Filho - UNESP
Jaboticabal, SP, Brasil

Sandra Augusta Santos
Universidade Estadual de Campinas - UNICAMP
Campinas, SP, Brasil

Eduardo V. O. Teixeira (Editor Executivo)
Universidade Federal do Ceará - UFC
Fortaleza, CE, Brasil

SBMAC **Sociedade Brasileira de Matemática Aplicada e Computacional**

2016

A Sociedade Brasileira de Matemática Aplicada e Computacional - SBMAC publica, desde as primeiras edições do evento, monografias dos cursos que são ministrados nos CNMAC.

Para a comemoração dos 25 anos da SBMAC, que ocorreu durante o XXVI CNMAC em 2003, foi criada a série **Notas em Matemática Aplicada** para publicar as monografias dos minicursos ministrados nos CNMAC, o que permaneceu até o XXXIII CNMAC em 2010.

A partir de 2011, a série passa a publicar, também, livros nas áreas de interesse da SBMAC. Os autores que submeterem textos à série Notas em Matemática Aplicada devem estar cientes de que poderão ser convidados a ministrarem minicursos nos eventos patrocinados pela SBMAC, em especial nos CNMAC, sobre assunto a que se refere o texto.

O livro deve ser preparado em **Latex (compatível com o Miktex versão 2.9), as figuras em eps** e deve ter entre **80 e 150 páginas**. O texto deve ser redigido de forma clara, acompanhado de uma excelente revisão bibliográfica e de **exercícios de verificação de aprendizagem** ao final de cada capítulo.

Veja todos os títulos publicados nesta série na página
http://www.sbmac.org.br/p_notas.php

SBMAC **Sociedade Brasileira de Matemática Aplicada e Computacional**

2016

Introdução à programação paralela em GPU para a implementação de métodos numéricos

Daniel Gregorio Alfaro Vigo
dgalfaro@dcc.ufrj.br
Silvana Rossetto
silvana@dcc.ufrj.br

Departamento de Ciência da Computação
Instituto de Matemática
Universidade Federal do Rio de Janeiro

Sociedade Brasileira de Matemática Aplicada e Computacional

São Carlos - SP, Brasil
2016

Coordenação Editorial: Maria do Socorro Nogueira Rangel

Coordenação Editorial da Série: Fernando Rodrigo Rafaeli

Editora: SBMAC

Capa: Matheus Botossi Trindade

Patrocínio: SBMAC

Copyright ©2016 by Daniel Gregorio Alfaro Vigo e Silvana Rossetto. Direitos reservados, 2016 pela SBMAC. A publicação nesta série não impede o autor de publicar parte ou a totalidade da obra por outra editora, em qualquer meio, desde que faça citação à edição original.

Catalogação elaborada pela Biblioteca do IBILCE/UNESP
Bibliotecária: Maria Luiza Fernandes Jardim Froner

Alfaro Vigo, Daniel G.
Introdução à programação paralela em GPU para a implementação de métodos numéricos - Rio de Janeiro, RJ :
SBMAC, 2016, 112 p., 21.5 cm - (Notas em Matemática Aplicada; v. 84)

e-ISBN 978-85-8215-078-8

1. Métodos numéricos 2. Programação paralela 3. GPU
4. CUDA 5. Equações de Maxwell
I. Alfaro Vigo, Daniel G. II. Rossetto, Silvana IV. Título. V. Série

CDD - 51

Agradecimentos

Ao nosso aluno Leandro Justino Veloso. A implementação paralela da solução das equações de Maxwell foi desenvolvida por ele em seu trabalho de mestrado.

Aos revisores anônimos pela preciosa contribuição.

Aos alunos Bruno Meurer e Thais Luca, primeiros leitores do manuscrito deste livro, pelas sugestões de aprimoramento que esperamos incorporar em versões futuras do texto.

À FAPERJ pelo auxílio financeiro concedido para a aquisição da GPU usada na realização deste trabalho.

Conteúdo

Prefácio **ix**

1 Fundamentos da programação paralela **1**
1.1 Introdução 1
1.2 Exemplos de algoritmos paralelos 2
1.3 Sincronização de algoritmos paralelos 7
1.4 Definição de programação paralela 10
1.5 Métricas de desempenho 12
1.5.1 Tempo de execução 12
1.5.2 Vazão computacional 13
1.5.3 Aceleração 13
1.6 Exercícios 14

2 Programação paralela em GPU **17**
2.1 Introdução 17
2.2 Principais aspectos do hardware das GPUs 18
2.3 Ambiente de programação CUDA 19
2.3.1 Primeiro exemplo de aplicação em CUDA 20
2.3.2 Sincronização de threads em CUDA 30
2.3.3 Modelo de programação de CUDA 36
2.3.4 Modelo de execução de CUDA 40
2.3.5 Hierarquia de memória em CUDA 42
2.3.6 Estratégias para otimizar o desempenho das aplicações 47
2.4 Programação de GPUs com OpenCL 50
2.5 Exercícios 51

3 Solução de equações lineares usando GPUs **53**
3.1 Introdução 53
3.2 Sistemas de equações tridiagonais 54
3.2.1 Método de redução cíclica 55
3.2.2 Método de redução cíclica paralela 64
3.3 Fatoração LU 70
3.4 Avaliação de desempenho 77
3.5 Exercícios 81

4 Solução das equações de Maxwell usando GPUs **83**
4.1 Introdução 83
4.2 Método de diferenças finitas para as equações de Maxwell 83
4.2.1 Esquema de Yee 84

4.2.2 Algoritmo do esquema de Yee 87
4.3 Implementação paralela do esquema de Yee em GPU 88
4.3.1 Aspectos gerais da implementação para GPU 88
4.3.2 Implementação paralela em CUDA 88
4.3.3 Exemplo de aplicação do programa 92
4.4 Avaliação de desempenho 93
4.4.1 Avaliação do algoritmo sequencial 94
4.4.2 Avaliação do algoritmo paralelo 94
4.5 Exercícios 96

Bibliografia **97**

Índice **101**

Prefácio

Nos últimos anos, temos observado grandes avanços em termos de arquiteturas computacionais com hardware paralelo. A grande maioria dos computadores comercializados hoje são fabricados com várias unidades de processamento independentes, ao invés de apenas uma. Além disso, tem se tornado cada vez mais comum a inclusão de *aceleradores* — entre os quais se destacam as GPUs (*Graphics Processing Units*) — nos computadores de uso geral com o objetivo de aumentar a velocidade de execução das aplicações.

Essas mudanças arquiteturais tiveram início nos anos 2000, quando os projetistas de computadores começaram a esbarrar nos limites físicos dos condutores fazendo com que o aumento da capacidade de processamento de um único *chip* não mais compensasse os custos elevados com a dissipação do calor gerado e outras consequências. A evolução das GPUs vem seguindo seu curso de forma independente e com avanços bastante relevantes, deixando de ser um hardware voltado apenas para aplicações de finalidades específicas para se tornar um tipo de acelerador capaz de permitir ganhos significativos em termos de capacidade de processamento nas máquinas atuais.

Essa alteração no projeto do hardware trouxe impactos consideráveis no desenvolvimento de software. Anteriormente, nas arquiteturas com um único núcleo de processamento, a cada nova geração de processador, o usuário era capaz de perceber os ganhos de desempenho das aplicações sem a necessidade de alterá-las. Com o aumento da velocidade de processamento do novo *chip*, as mesmas aplicações passavam a executar de forma mais rápida, ou novas aplicações tornavam-se viáveis de serem executadas dentro de um intervalo de tempo definido. Nas arquiteturas multinúcleos isso não necessariamente ocorre, e para explorar de fato toda a capacidade de processamento desse novo hardware é imprescindível o uso das técnicas de programação paralela.

Os avanços do hardware dos computadores — a um custo financeiro acessível para usuários convencionais — permite hoje que se possa dispor em uma única máquina da capacidade de processamento que até anos atrás só era possível em um *cluster* (coleção) de computadores. Para a computação numérica, esse novo cenário pode propiciar grandes avanços. Isso porque boa parte dos métodos numéricos são computacionalmente intensivos, isto é, requerem muito mais operações de processamento (com uso intensivo de aceleradores) do que operações de entrada e saída. Dessa forma, a possibilidade de dispor de uma maior capacidade de processamento em um único nó computacional se adequa bastante bem com as demandas desses algoritmos.

O principal desafio que se coloca para os programadores é como (re)implementar os algoritmos numéricos para que eles sejam executados de forma eficiente nessas novas arquiteturas. O uso das técnicas de programação paralela ganha notável destaque como condição fundamental para viabilizar os ganhos de desempenho espera-

dos, assim como o conhecimento básico dos principais aspectos e modo de operação das arquiteturas de hardware paralelo.

O primeiro passo no estudo da programação paralela é como projetar algoritmos com mais de um fluxo de execução, garantindo a execução correta desses algoritmos. Esse primeiro passo é desafiador, pois nem sempre é trivial dividir uma tarefa em subtarefas que podem ser executadas em paralelo e, em boa parte dos casos, é necessário ainda acrescentar ao algoritmo etapas de sincronização entre os fluxos paralelos para garantir a execução correta da aplicação. Uma vez superado esse passo inicial, a etapa seguinte é aprofundar o conhecimento sobre as técnicas de programação paralela e sobre a arquitetura de execução alvo para explorar ao máximo o potencial de paralelismo das aplicações frente às implementações sequenciais mais otimizadas.

Neste livro, daremos ênfase maior ao primeiro passo do estudo da programação paralela voltada para arquiteturas com GPUs, apresentando seus conceitos e desafios fundamentais e como aplicá-la na implementação de métodos numéricos. Esperamos que o seu conteúdo dê ao leitor a formação básica necessária para avançar para a etapa seguinte de explorar com mais profundidade as técnicas de paralelização visando obter ganhos de desempenho significativos para as suas aplicações.

Selecionamos como casos de estudo a solução de sistemas de equações lineares e das equações de Maxwell. Embora já existam diferentes implementações paralelas para esses problemas disponíveis na literatura, nosso objetivo é mostrar de forma sistemática para o leitor como construir algoritmos paralelos para execução em GPUs, selecionando e avaliando diferentes técnicas de particionamento do problema. Assim, esperamos apresentar os fundamentos básicos para que o leitor possa construir seus próprios algoritmos paralelos para atender requerimentos específicos de suas aplicações.

O livro se destina a estudantes de graduação e pós-graduação das áreas de ciências exatas. Espera-se que o leitor tenha cursado disciplinas básicas de programação de computadores e de cálculo numérico, e que tenha domínio da linguagem de programação C. Não é necessário que o estudante tenha conhecimento prévio de programação paralela.

O conteúdo do livro está organizado em quatro capítulos. No Capítulo 1 apresentamos os fundamentos da programação paralela, incluindo técnicas de particionamento de problemas e métricas para avaliação de desempenho. No Capítulo 2, apresentamos uma visão geral da arquitetura das GPUs e do ambiente de programação CUDA. Nos Capítulos 3 e 4 abordamos a solução de sistemas de equações lineares e das equações de Maxwell, respectivamente. Neles discutimos estratégias básicas de implementação paralela desses problemas para GPUs. Para cada uma das implementações, apresentamos uma avaliação preliminar dos ganhos de desempenho obtidos.

Todos os códigos apresentados no texto estão disponíveis para *download* no endereço http://www.dcc.ufrj.br/~silvana/.

Esperamos que este material sirva de apoio e incentivo para que os estudantes explorem de forma eficiente as vantagens das novas arquiteturas de hardware paralelo, em particular as GPUs.

Rio de Janeiro, 22 de julho de 2016.

Daniel Alfaro e Silvana Rossetto

Capítulo 1

Fundamentos da programação paralela

1.1 Introdução

Tradicionalmente aprendemos a programar desenvolvendo algoritmos sequenciais, ou seja, algoritmos que modelam uma única sequência de passos consecutivos. A **programação paralela** estende esse modelo básico permitindo dividir a sequência de passos consecutivos do algoritmo em duas ou mais subsequências de passos que podem ser executadas ao mesmo tempo em processadores separados. Um dos objetivos da programação paralela é diminuir o tempo total requerido para executar a tarefa desejada, explorando toda a capacidade do hardware disponível.

Para ilustrar essa ideia, considere que temos a tarefa de pintar quatro cômodos de uma casa e que para isso dispomos inicialmente dos serviços de um único pintor. O algoritmo mais simples para realizar essa tarefa é determinar que o pintor comece pintando o primeiro cômodo, depois o segundo, em seguida o terceiro e por fim o quarto. Esse é um exemplo de uma sequência consecutiva de passos (algoritmo sequencial) para executar uma tarefa determinada. Entretanto, se dispusermos dos serviços de dois pintores, podemos dividir a tarefa entre eles, determinando que o primeiro deverá pintar o primeiro e o segundo cômodos e o segundo pintor deverá pintar o terceiro e quarto cômodos. Isso é possível porque não existe dependência em relação à ordem em que os cômodos são pintados e não há restrição em pintar dois cômodos distintos ao mesmo tempo. Se todos os cômodos tiverem o mesmo tamanho, o tempo total para realizar a tarefa com dois pintores será aproximadamente a metade do tempo requerido com um único pintor.

Hoje, os computadores de uso geral possuem duas ou mais unidades de processamento [34]. Podemos dizer que é como se tivéssemos sempre, no mínimo, dois pintores disponíveis para realizar a tarefa de pintar uma casa inteira. Precisamos, então, aprimorar nossa capacidade de pensar e projetar algoritmos paralelos, dividindo as atividades de uma tarefa completa em subsequências de passos independentes. Caso contrário, não conseguiremos explorar toda a capacidade disponível do hardware; ou perderemos a oportunidade de executar nossas aplicações em um tempo menor.

Outra evolução significativa dos computadores nos últimos anos foi a generalização do hardware das GPUs (*Graphics Processing Units*) tornando possível usá-las para computação de propósito geral, e não apenas para processamento gráfico. As

GPUs possuem milhares de unidades de processamento e podem ser adicionadas ao hardware do computador a um custo bastante razoável. Com o uso das GPUs é possível reduzir significantemente o tempo total de execução de aplicações que processam grandes volumes de dados com certa independência entre suas instâncias. Também nesse caso — para que todo o potencial do hardware disponível seja de fato explorado — é necessário compreender o modelo de operação dessas novas arquiteturas e desenvolver algoritmos paralelos para dividir o processamento total entre as várias unidades de execução disponíveis.

A programação paralela engloba o uso de técnicas para dividir uma tarefa em subtarefas menores, e de mecanismos para coordenar a execução dessas subtarefas por processadores distintos. Dependendo das características do problema, e do ambiente de execução paralela alvo (CPU multinúcleo ou GPU), o programador deve selecionar as alternativas mais adequadas.

Neste capítulo, apresentamos os conceitos fundamentais da programação paralela. Começamos descrevendo em que consiste escrever um programa paralelo e quais são os mecanismos de controle que precisam ser usados para escrever programas corretos. Em seguida, apresentamos métricas de desempenho das aplicações, as quais são usadas para avaliar os ganhos obtidos em relação ao código sequencial e o grau de escalabilidade dos algoritmos paralelos, ou seja, como o algoritmo se comporta com o aumento da carga de trabalho e das unidades de processamento disponíveis.

1.2 Exemplos de algoritmos paralelos

Para introduzir os conceitos básicos da programação paralela, vamos começar tomando como referência dois exemplos de problemas que podem ser implementados de forma paralela: **soma de vetores** e **multiplicação de matrizes**.

Soma de vetores Considere a operação SAXPY (*Single-Precision* $A{\cdot}X$ *Plus* Y) [11] que consiste em calcular `C = A * k + B`, sendo A, B e C vetores de tamanho N e k um número. Essa é uma operação que aparece com frequência em problemas de álgebra linear computacional. Em razão disso, há diversas implementações otimizadas para ela em bibliotecas especializadas (por exemplo, essa é uma das operações básicas oferecidas na biblioteca BLAS [5]), ou mesmo como uma instrução de máquina em algumas arquiteturas de computadores (como é o caso da biblioteca de operações matemáticas MKL da Intel [19]). Nosso objetivo aqui não é propor uma implementação mais eficiente dessa operação, mas apenas usá-la como exemplo inicial de um problema bastante conhecido que pode ser implementado de forma paralela.

Para encontrar o vetor de saída C, precisamos calcular cada elemento desse vetor fazendo: `C[i] = A[i] * k  + B[i], 0<=i<N`. Na programação sequencial, executamos a sentença acima uma vez para cada posição do vetor C, como ilustrado no programa 1.1 (escrito na linguagem C).

```
1  #define N 1000000 //N igual ao número de elementos do vetor
2  //calcula C = k * A + B
3  void somaVetoresSeq(const float a[], const float b[], float c
       [], float k, int n) {
4     int i;
5     for(i=0; i<n; i++) {
6        c[i] = a[i] * k  + b[i];
```

```
7       }
8   }
9   void main() {
10      float a[N], b[N], c[N];
11      //inicializa os vetores a e b
12      ...
13      somaVetoresSeq(a, b, c, 2.0, N);
14  }
```

Código 1.1: Programa sequencial para somar dois vetores.

Esse programa será executado por um único processador que calculará os elementos do vetor de saída C na ordem crescente das suas posições. A Figura 1.1 ilustra essa ideia (a seta representa o fluxo de execução do programa).

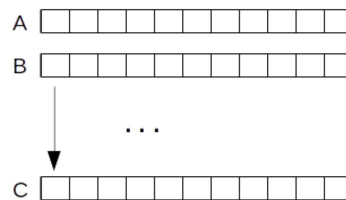


Figura 1.1: Soma sequencial dos elementos de dois vetores.

Para escrever um algoritmo paralelo que calcule o vetor `C = A * k + B`, vamos pensar também em termos da menor tarefa do problema, que é o cálculo de uma posição do vetor de saída. Como cada posição do vetor é independente das demais, podemos computar todos os valores ao mesmo tempo, desde que tenhamos um número de unidades de processamento igual ao número de elementos do vetor. Nesse caso, ao invés de usarmos o comando de repetição (*for*), definimos uma função que calcula o valor de um único elemento e disparamos N fluxos de execução distintos que executarão essa função uma única vez, informando a posição que deverá ser calculada. A Figura 1.2 mostra essa estratégia (cada seta representa um fluxo de execução independente).

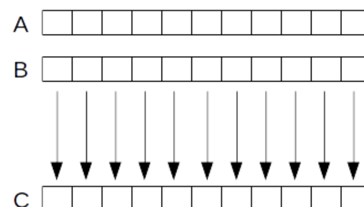


Figura 1.2: Soma paralela dos elementos de dois vetores.

No código 1.2, mostramos como implementar esse algoritmo. Vamos assumir por ora que na linha 15 usaremos funções especiais providas pela linguagem C para criar e disparar fluxos independentes de execução. 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. Entretanto, nem sempre dispomos de tantas unidades de processamento simultâneas (como ocorre com as CPUs). Nesse caso, podemos variar o tamanho da tarefa processada por cada fluxo de

execução, por exemplo, atribuindo para cada um deles o cálculo de um subconjunto de elementos do vetor de saída.

```
1  #define N 1000000 //N igual ao número de elementos do vetor

3  //calcula C[i] = k * A[i] + B[i]
4  void calculaElementoVetor(const float a[], const float b[],
       float c[], float k, int pos) {
5     c[pos] = k * a[pos] + b[pos];
6  }

8  void main() {
9     float a[N], b[N], c[N];
10    int i;
11    //inicializa os vetores a e b
12    ...
13    //faz C = k * A + B
14    for(i=0; i<N; i++) {
15       //dispara um fluxo de execução f para executar:
              calculaElementoVetor(a, b, c, 2.0, i)
16    }
17 }
```

Código 1.2: Pseudo-código paralelo para somar dois vetores. Cada fluxo de execução paralela calcula um elemento do vetor de saída.

Assumindo que vamos criar um fluxo de execução paralela para cada unidade de processamento da máquina, uma alternativa é dividir o número total de elementos do vetor (N) pelo número de unidades de processamento (P), e deixar para a última unidade o cálculo dos elementos adicionais (no caso da divisão anterior não ser exata), como ilustrado no programa 1.3.

```
1  #define N 1000000 //N igual ao número de elementos do vetor
2  #define P 4 //P igual ao número de unidades de processamento

4  //calcula uma parte de C = k * A + B
5  void calculaElementosConsVetor(const float a[], const float b
       [], float c[], float k, int inicio, int fim) {
6     int i;
7     for(i=inicio; i<fim; i++) {
8        c[i] = k * a[i] + b[i];
9     }
10 }

12 void main() {
13    float a[N], b[N], c[N];
14    int i, inicio, fim;
15    //inicializa os vetores a e b
16    ...
17    //faz C = k * A + B
18    for(i=0; i<P; i++) {
19       inicio = i * (N/P);
20       //o último fluxo trata os elementos restantes
21       if (i<P-1) {
22          fim = inicio + (N/P);
23       } else {
```

```
24            fim = N;
25         }
26         //dispara um fluxo de execução f para executar:
               calculaElementosConsVetor(a, b, c, 2.0, inicio, fim);
27     }
28 }
```

Código 1.3: Pseudo-código paralelo para somar dois vetores. Cada fluxo de execução paralela calculará um subconjunto de elementos consecutivos do vetor de saída.

Outra alternativa para dividir a tarefa completa (N elementos) entre as unidades de processamento (P) é alternar o elemento que deve ser calculado por cada unidade. Nesse caso, não é necessário lidar explicitamente com o caso do número de elementos não ser divisível pelo número de unidades de processamento pois os elementos restantes serão automaticamente divididos entre as unidades de processamento.

O programa 1.4 mostra como implementar essa alternativa. Cada fluxo de execução deverá usar como índice inicial o seu identificador único. A cada passo do algoritmo salta-se P elementos. Em todos os fluxos, a repetição termina quando o valor do índice alcança o valor N.

```
1  #define N 1000000 //N igual ao número de elementos do vetor
2  #define P 4 //P igual ao número de unidades de processamento
3  //calcula uma parte de C = k * A + B
4  void calculaElementosInterVetor(const float a[], const float b
       [], float c[], float k, int inicio, int salto, int n) {
5     int i;
6     for(i=inicio; i<n; i=i+salto) {
7        c[i] = k * a[i]  + b[i];
8     }
9  }
10 void main() {
11    float a[N], b[N], c[N];
12    int i;
13    //inicializa os vetores a e b
14    ...
15    //faz C = k * A + B
16    for(i=0; i<P; i++) {
17       //dispara um fluxo de execução f para executar:
              calculaElementosInterVetor(a, b, c, 2.0, i, P, N);
18    }
19 }
```

Código 1.4: Pseudo-código paralelo para somar dois vetores. Cada fluxo de execução paralela calculará um subconjunto de elementos intercalados do vetor de saída.

Uma preocupação importante quando projetamos um algoritmo paralelo é garantir que todas as unidades de processamento recebam a mesma carga de trabalho para aproveitar ao máximo todos os recursos de processamento disponíveis. Se uma unidade termina sua execução muito antes das demais, esse recurso ficará ocioso e não será aproveitado pela aplicação. Todas as alternativas de paralelização mostradas acima garantem o balanceamento de carga entre as unidades de processamento. Entretanto, o padrão de acesso à memória varia de uma alternativa para a outra, o que poderá fazer com que o desempenho de cada uma delas — em termos de tempo de acesso à memória — varie de forma significativa. Discutiremos com mais profundidade o impacto que o padrão de acesso à memória pode ocasionar no desempenho

das aplicações paralelas no capítulo 2, tratando especificamente da organização da memória nas GPUs.

Multiplicação de matrizes Consideremos agora o problema de multiplicação de matrizes. Dadas duas matrizes de entrada, A e B, queremos calcular a matriz `C = A * B`. A parte central do algoritmo sequencial para resolver esse problema está implementada no código 1.5. Por simplicidade, assume-se que as matrizes são quadradas de dimensão $N \times N$.

```
1  ...
2  for(i=0; i<N; i++) {
3     for(j=0; j<N; j++) {
4        soma = 0.0;
5        for(k=0; k<N; k++) {
6           soma = soma + a[i][k] * b[k][j];
7        }
8        c[i][j] = soma;
9     }
10 }
11 ...
```

Código 1.5: Trecho de código de um programa sequencial para multiplicar duas matrizes.

De que forma podemos paralelizar esse algoritmo? Uma estratégia é isolarmos o cálculo de cada elemento da matriz de saída (linhas 4 a 7 do código 1.5) e tratar esse subproblema como um tarefa básica que poderá ser executada por fluxos de execução paralelos. Na Figura 1.3 ilustramos essa ideia. O cálculo de cada elemento da matriz de saída consiste em uma sequência de operações de multiplicação e soma, completamente independente do cálculo dos demais elementos. Para calcular o elemento (i, j) da matriz de saída C, precisamos apenas dos elementos da linha i da matriz A e dos elementos da coluna j da matriz B. O resultado da computação é armazenado em uma posição única da matriz de saída.

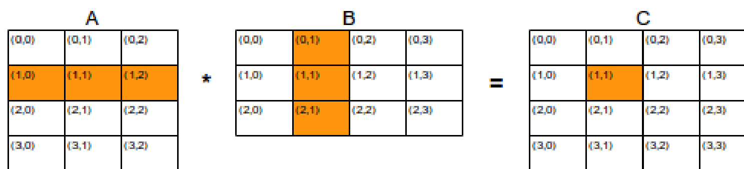


Figura 1.3: Cálculo de um elemento da matriz de saída de uma multiplicação de matrizes: $C_{(1,1)} = A_{(1,0)} * B_{(0,1)} + A_{(1,1)} * B_{(1,1)} + A_{(1,2)} * B_{(2,1)}$

A linha i da matriz A também é usada para calcular outros elementos da matriz C (por exemplo, os elementos $C_{(i,j-1)}$ e $C_{(i,j+1)}$), assim como a coluna j da matriz B é usada mais de uma vez para calcular elementos diferentes da matriz de saída. Essa reutilização de valores de entrada (que não ocorria no exemplo da soma de vetores) não afeta a paralelização do problema, pois dois fluxos de execução podem ler a mesma variável de entrada ao mesmo tempo.

O programa 1.6 mostra o pseudo-código de um algoritmo paralelo para multiplicar duas matrizes quadradas. Por simplicidade, deixamos as matrizes como variáveis globais. Nos exercícios no final dessa seção, discutiremos outras alternativas de paralelização desse problema.

```
#define N 1000 //N igual à dimensão da matriz
float a[N][N], b[N][N], c[N][N];
//calcula o elemento (i,j) da matriz C (C = A * B)
//C[i][j]=A[i][0]*B[0][j]+...+A[i][n-1]*B[n-1][j]
void calculaElementoMatriz(int dim, int i, int j) {
   int k, soma = 0.0;
   for(k=0; k<dim; k++) {
      soma = soma + a[i][k] * b[k][j];
   }
   c[i][j] = soma;
}
void main() {
   int i, j;
   //inicializa as matrizes a e b
   ...
   //faz C = A * B
   for(i=0; i<N; i++) {
      for(j=0; j<N; j++) {
         //dispara um fluxo de execução f para executar:
             calculaElementoMatriz(N, i, j);
      }
   }
}
```

Código 1.6: Pseudo-código paralelo para multiplicar duas matrizes. Cada fluxo de execução paralela calculará um elemento da matriz de saída.

1.3 Sincronização de algoritmos paralelos

Os exemplos que vimos na seção anterior são relativamente simples de paralelizar pois é possível dividí-los em subproblemas menores que podem ser completamente resolvidos de forma independente, isto é, sem a necessidade dos fluxos de execução paralela interagirem. A maioria dos problemas, entretanto, requerem um esforço maior para serem paralelizados pois a computação dos subproblemas exige que os fluxos de execução troquem informações entre si e sincronizem suas ações.

Vamos tomar como exemplo o problema de **somar os elementos de um vetor**. Considere que temos um vetor A com N elementos e precisamos somar todos os seus valores. O programa 1.7 mostra a implementação de um algoritmo sequencial para resolver esse problema. A função `main` cria uma variável local para armazenar o resultado da soma de todos os elementos do vetor (`soma_total`), chama a função que executa a soma e imprime o resultado.

```
#define N 1000000 //N igual ao número de elementos do vetor
//Soma os n elementos do vetor a
float somaElemVetorSeq(const float a[], int n) {
   int i;
   float soma = 0;
   for(i=0; i<n; i++) {
      soma = soma + a[i];
   }
   return soma;
}
void main() {
```

```
12    float a[N], soma_total;
13    int i;
14    //inicializa o vetor a
15    ...
16    soma_total = somaElemVetorSeq(a, N);
17    //imprime o valor da variável 'soma_total'
18 }
```

Código 1.7: Programa sequencial para somar os elementos de um vetor.

Para paralelizar esse algoritmo vamos usar a mesma estratégia adotada no programa 1.3 de dividir o número total de elementos do vetor (N) em grupos consecutivos de P elementos e criar um fluxo de execução paralela para somar os elementos de cada grupo. O programa 1.8 mostra essa solução. (Na seção 2.3.2 discutiremos outra estratégia de paralelização desse problema).

Há vários pontos a destacar nesse programa. O primeiro deles é que, diferente dos exemplos de **soma de vetores** e de **multiplicação de matrizes**, que possuíam como saída uma estrutura de dados com a mesma dimensão da estrutura de dados de entrada, no problema de somar os elementos de um vetor temos como entrada um vetor e como saída um único valor que deverá ser preenchido com a contribuição do processamento realizado por diferentes fluxos de execução. No programa 1.8, declaramos a variável `soma_total` como variável global (linha 3) para permitir que a função executada pelos fluxos de execução (`somaGrupoConsElemVetor`) possa alterá-la diretamente. (Outra alternativa seria passar o endereço dessa variável como argumento para a função executada pelos fluxos de execução.)

Dentro da função `somaGrupoConsElemVetor`, cada fluxo de execução calcula a soma dos elementos do vetor no intervalo definido e armazena essa soma em uma variável local chamada `soma_local` (linhas 7 a 9). Essa abordagem permite que os fluxos de execução trabalhem de forma completamente independente enquanto somam os valores dos elementos dentro do intervalo dado.

```
1  #define N 1000000 //N igual ao número de elementos do vetor
2  #define P 4 //P igual ao número de unidades de processamento
3  float soma_total = 0;
4  //Soma uma parte dos n elementos do vetor a
5  void somaGrupoConsElemVetor(const float a[], int inicio, int
       fim) {
6     int i;
7     float soma_local = 0;
8     for(i=inicio; i<fim; i++) {
9        soma_local = soma_local + a[i];
10    }
11    // !!! solicita exclusão mútua !!!
12    soma_total = soma_total + soma_local;
13    // !!! libera a exclusão mútua !!!
14 }
15 void main() {
16    float a[N];
17    int i, inicio, fim;
18    //inicializa o vetor a
19    ...
20    //soma os n elementos de a
21    for(i=0; i<P; i++) {
22       inicio = i * (N/P);
23       //o último fluxo trata os elementos restantes
```

```
24         if (i<P-1) {
25            fim = inicio + (N/P);
26         } else {
27            fim = N;
28         }
29         //dispara um fluxo de execução f para executar:
               somaGrupoConsElemVetor(a, inicio, fim);
30      }
31      // !!! espera todos os fluxos de execução terminarem !!!
32      //imprime o valor da variável 'soma_total'
33      printf("soma total = %f", soma_total);
34   }
```

Código 1.8: Pseudo-código paralelo para somar os elementos de um vetor. Cada fluxo de execução paralela somará um subconjunto de elementos consecutivos do vetor de saída. Nesse código é necessário usar mecanismos de sincronização.

Quando terminam de realizar essa soma, os fluxos de execução precisam acrescentar na soma total o valor da sua soma local. Isso é feito na linha 12. Porém, se deixarmos dois ou mais fluxos de execução atualizarem a variável global `soma_total` ao mesmo tempo, o resultado das várias atualizações poderá ficar incorreto. Vejamos porque isso poderia acontecer. Quando um fluxo de execução processa a linha 12, ele primeiro carrega os valores atuais das variáveis `soma_total` e `soma_local` e guarda esses valores em áreas de memória especiais — e exclusivas de cada fluxo de execução — chamadas **registradores**. Em seguida, a unidade de processamento aritmético do computador soma o conteúdo desses dois registradores e armazena o resultado em um terceiro registrador. Após isso, o conteúdo desse terceiro registrador é carregado de volta para atualizar a variável `soma_total`. Então, se dois ou mais fluxos de execução processarem essa sequência de instruções ao mesmo tempo, eles armazenarão nos seus registradores o mesmo valor inicial de `soma_total` (vamos considerar, por exemplo, que esse valor seja 0). Em seguida, acrescentarão a esse valor a sua soma local (fazendo 0 + `soma_local`), e finalmente atualizarão a variável `soma_total` com o resultado dessa última soma. O que acontecerá ao final é que o novo valor da variável `soma_total` será o valor atribuído pelo último fluxo de execução, que conterá apenas o incremento da sua soma local, as demais atualizações serão perdidas.

Sincronização por exclusão mútua Para evitar esse problema, é preciso garantir que os fluxos de execução paralela executarão a linha 12 com **exclusão mútua**, isto é, apenas um fluxo de execução a cada vez. Podemos dizer de outra forma que a sentença da linha 12 deve ser **atômica**, isto é, ela deve ser completamente executada antes que outro fluxo de execução comece a processá-la. Assim, quando um fluxo de execução terminar de executar a linha 12, teremos a garantia de que o valor da variável global `soma_total` está atualizado. O próximo fluxo de execução irá ler o valor atual de `soma_total` e acrescentar a sua soma local.

A **exclusão mútua** é um tipo de **sincronização** que pode ser requerida dentro de um algoritmo paralelo. Os fluxos de execução precisam sincronizar suas ações para evitar a ocorrência de erros de execução próprios da programação paralela. No programa 1.8, enfatizamos isso nas linhas 11 e 13, assumindo por ora que algum mecanismo de exclusão mútua (provido pela linguagem C) será invocado nessas linhas para sinalizar, respectivamente, a entrada e saída de um trecho de código atômico.

Sincronização por condição Na linha 33 do programa 1.8, aparece a demanda por outro tipo de sincronização. O fluxo de execução principal (aquele que começa com a execução do programa na função `main`) precisa aguardar o término dos demais fluxos de execução para então imprimir o valor final da variável `soma_total`. Esse tipo de sincronização é chamado de **sincronização por condição**, uma vez que há uma condição lógica do problema que requer que o resultado final somente seja exibido após todos os elementos do vetor serem somados. Nesse caso, o fluxo de execução principal deverá ser explicitamente bloqueado até que todos os demais fluxos de execução completem a atualização da variável global `soma_total`.

Sincronização por barreira Um tipo particular de sincronização por condição aparece em problemas que requerem que todos os fluxos de execução (ou um grupo deles) sincronizem suas ações a cada passo do algoritmo paralelo. Essa sincronização é chamada de **sincronização por barreira** porque funciona exatamente como uma "barreira" (ou bloqueio coletivo) que faz com que todos os fluxos de execução aguardem pelos demais até que todos tenham chegado a esse ponto (ou passo) do algoritmo. A Figura 1.4 ilustra essa situação.

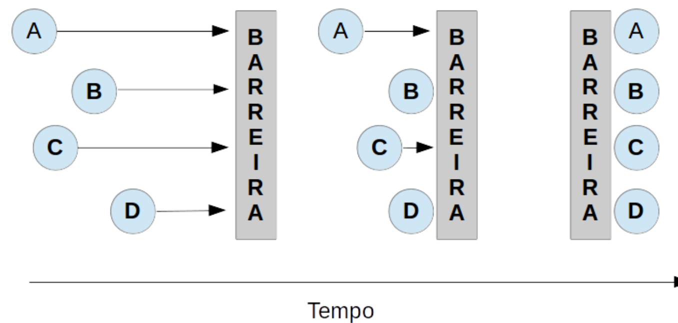


Figura 1.4: Ilustração do mecanismo de sincronização por barreira. Todos os fluxos de execução (A, B, C e D) devem chegar ao ponto da execução onde está a barreira para então poderem continuar suas execuções.

Em resumo, para coordenar a execução de um algoritmo paralelo é necessário que os fluxos de execução troquem informações entre si sobre seus estados de execução e possam suspender ou retomar suas execuções dependendo do estado de execução dos demais fluxos ou do estado global da aplicação. Novamente, dependendo do ambiente de programação paralela usado, há diferentes mecanismos ou alternativas que permitem a comunicação entre os fluxos de execução e a coordenação das suas execuções. Veremos nos Capítulos 2, 3 e 4 como essas demandas aparecem em diferentes exemplos de problemas e como tratá-las no ambiente específico das GPUs.

1.4 Definição de programação paralela

Em [20], os autores 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 de-

compor P_D em k subproblemas:

$$D = d_1 \oplus d_2 \oplus \cdots \oplus d_k = \sum_{i=1}^{k} d_i \tag{1.4.1}$$

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

De acordo com a definição de Navarro et al. [20], 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:

$$f(D) = f(d_1) \oplus f(d_2) \oplus \cdots \oplus f(d_k) = \sum_{i=1}^{k} f(d_i) \tag{1.4.2}$$

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**:

$$D(S) = d_1(S) \oplus d_2(S) \oplus \cdots \oplus d_k(S) = \sum_{i=1}^{k} d_i(S) \tag{1.4.3}$$

Projetar e implementar algoritmos paralelos não é uma tarefa simples e não há uma regra geral para desenvolver algoritmos paralelos perfeitos. Foster [8] sugere um método para projetar algoritmos paralelos que se divide em quatro etapas: particionamento, comunicação, aglomeração e mapeamento.

1. **Particionamento**: a tarefa que deve ser executada e o conjunto de dados associado a ela são decompostos em tarefas menores. Nessa etapa, o objetivo é identificar oportunidades de execução paralela. Para isso, caracterizar o domínio do problema (com "paralelismo de dados" ou "paralelismo de tarefas") é fundamental para escolher uma boa estratégia de particionamento.

2. **Comunicação**: nessa etapa, determina-se a comunicação requerida para coordenar a execução da tarefa e define-se as estruturas e algoritmos de comunicação mais apropriados.

3. **Aglomeração**: as subtarefas e estruturas de comunicação definidas nas etapas anteriores são avaliadas com respeito aos requisitos de desempenho e custos de implementação e, se necessário, as subtarefas são combinadas em tarefas maiores para melhorar o desempenho ou reduzir os custos de desenvolvimento.

4. **Mapeamento**: nessa última etapa, cada tarefa é designada para uma unidade de processamento de uma forma que satisfaça a meta de maximizar o uso da capacidade de processamento paralela disponível e minimizar os custos de comunicação e gerência. O mapeamento pode ser feito de forma estática ou determinado em tempo de execução. Para essa etapa é fundamental conhecer as características principais da arquitetura alvo.

Olhando para o hardware específico das GPUs, o desempenho das aplicações é melhor quando os fluxos de execução paralela executam as mesmas instruções sobre partes distintas do domínio dos dados. Por isso, problemas com **paralelismo de dados** são os mais adequados para execução em GPUs. Trataremos neste texto apenas desse tipo de problema.

1.5 Métricas de desempenho

As métricas de desempenho são usadas para quantificar a qualidade de um algoritmo. Os algoritmos sequenciais são normalmente avaliados e comparados por meio de duas métricas: **tempo de execução** e **espaço de memória requerido**. Para os algoritmos paralelos, métricas adicionais são normalmente usadas, entre elas **vazão computacional** e **aceleração**. Nesta seção, apresentaremos essas métricas.

1.5.1 Tempo de execução

Dado um problema de tamanho n, vamos denotar por $T_p(n,p)$ o tempo de execução de um algoritmo paralelo usando p processadores e por $T_s(n)$ o tempo de execução de um algoritmo sequencial. Medimos o **tempo de execução** (T) de um programa, ou de parte dele, tomando o instante de tempo inicial (imediatamente antes do início do trecho do programa que queremos avaliar) e o instante de tempo final (imediatamente após a execução do trecho de código avaliado). Em seguida, calculamos o intervalo de tempo decorrido.

Para que essa tomada de tempo seja realizada de forma correta, precisamos usar funções que meçam o tempo de forma contínua, isto é, sem a interferência de alterações ou ajustes no relógio do sistema. Em ambientes Linux, por exemplo, podemos calcular o tempo de execução de um trecho de um programa escrito em C usando a função `clock_gettime`, definida na biblioteca `time.h`. Essa função retorna uma estrutura de dados que contém dois campos: um contador de segundos e um contador de nanosegundos cujos valores são preenchidos com o tempo transcorrido desde uma data predefinida.

O código 1.9 mostra um exemplo de uso da função `clock_gettime` para calcular o tempo gasto para executar um determinado trecho do programa. A função auxiliar `tempoAtual` chama a função `clock_gettime` e em seguida processa a estrutura de dados retornada para gerar uma medida de tempo em segundos.

```
1  #include <time.h>
2  //retorna o instante de tempo atual em segundos
3  double tempoAtual() {
4     struct timespec tempo;
5     clock_gettime(CLOCK_MONOTONIC_RAW, &tempo);
6     return tempo.tv_sec + tempo.tv_nsec/1000000000.0;
7  }

9  void main() {
10    double inicio, fim, delta;
11    inicio = tempoAtual();
12    //...trecho do programa que queremos medir o tempo
13    fim = tempoAtual();
14    //tempo consumido para executar o trecho do programa
15    delta = fim - inicio;
16 }
```

Código 1.9: Exemplo de código em C com tomada do tempo de execução.

Normalmente, quando já dispomos de uma implementação sequencial do problema e queremos paralelizá-la, o primeiro passo consiste em identificar quais partes do programa demandam maior tempo de execução e então avaliar as possibilidades de paralelização dessas partes. Problemas que demandam acesso intensivo a dados que estão armazenados na memória podem consumir parte significativa do tempo

de execução no acesso a esses dados. Nesse caso, o desempenho do programa é limitado principalmente pela largura de banda de acesso à memória. Por outro lado, quando o problema gasta a maior parte do tempo com operações aritméticas e de ponto flutuante, o seu desempenho é limitado pelo processamento. De forma geral, problemas limitados por processamento têm maior potencial de paralelismo em GPUs.

1.5.2 Vazão computacional

A **vazão computacional** (V) avalia a taxa de execução de operações aritméticas por unidade de tempo. Uma medida comumente usada para essa finalidade é `GFLOP/s` que mede a quantidade de operações de ponto flutuante (FLOP) realizadas por segundo. O prefixo `G` estende para `giga` e representa o valor 10^9.

Para calcular a vazão computacional, precisamos contabilizar o número de operações aritméticas (i) realizadas para cada elemento do domínio de dados do problema. De forma geral, cada operação aritmética (soma, subtração, multiplicação e divisão) está associada a uma operação de ponto flutuante. Assim, a vazão computacional (V) pode ser computada de acordo com a seguinte equação:

$$V(n,p) = i \cdot n/(T_p(n,p) \cdot 10^9) \tag{1.5.4}$$

em que n é o tamanho da entrada, p é o número de processadores usados, i é a quantidade de operações de ponto flutuante executadas para cada elemento da entrada e T_p é o tempo de processamento em segundos do algoritmo paralelo.

1.5.3 Aceleração

Dado um problema de tamanho n, a **aceleração** (A) (do termo em inglês *speedup*) é a razão entre o tempo de execução da melhor versão sequencial do algoritmo ($T_s(n)$) e o tempo de execução da versão paralela ($T_p(n,p)$) usando p processadores:

$$A(n,p) = T_s(n)/T_p(n,p) \tag{1.5.5}$$

Caso a implementação sequencial do algoritmo não esteja disponível, calculamos a aceleração substituindo o tempo do algoritmo sequencial ($T_s(n)$) pelo tempo do algoritmo paralelo usando um único processador ($T_p(n,1)$). Nesse caso temos:

$$A(n,p) = T_p(n,1)/T_p(n,p) \tag{1.5.6}$$

Quando paralelizamos um problema, o ideal seria conseguir dividir a tarefa igualmente entre os fluxos de execução paralela sem adicionar carga de trabalho extra. Nesse caso, teríamos $T_p(n,p) = T_s(n)/p$ e a aceleração seria perfeitamente linear, ou seja, com p processadores o algoritmo paralelo executaria p vezes mais rápido que o algoritmo sequencial. Teoricamente esse é o valor máximo de aceleração de um algoritmo paralelo.

Entretanto, há vários fatores que impossibilitam uma curva de aceleração perfeitamente linear: o algoritmo paralelo requer uma carga de processamento adicional, necessária para gerenciar a execução dos fluxos de execução; nem sempre é possível paralelizar o problema inteiro ou dividir a tarefa igualmente entre os fluxos de execução; a complexidade de gerência dos fluxos de execução e a competição pelo acesso à memória pode crescer com o aumento do número de processadores. Nesses casos, a curva da aceleração é dita *sublinear* e o principal desafio para os

programadores é tentar minimizar a degradação de desempenho causada por esses fatores.

Há ainda situações (pouco comuns) em que a curva da aceleração ultrapassa o seu limite teórico, sendo chamada de *superlinear*. Diferentes fatores relacionados com o hardware da máquina podem levar a essa situação, entre eles o seu modelo de organização e acesso à memória.

1.6 Exercícios

1. No programa 1.6 mostramos uma forma de paralelizar o algoritmo de multiplicação de matrizes criando um fluxo de execução independente para calcular cada um dos elementos da matriz de saída. Proponha outra solução onde a tarefa de cada fluxo de execução seja calcular uma linha inteira da matriz de saída.

2. Para arquiteturas de hardware com poucas unidades de processamento (como é o caso das CPUs *multicores*) geralmente é melhor criar uma quantidade de fluxos de execução igual ao número de unidades de processamento. Altere a solução do exercício anterior fixando o número de fluxos de execução e dividindo o cálculo das linhas da matriz de saída entre eles.

3. A série mostrada abaixo pode ser usada para estimar o valor da constante π. A função `piSequencial()` implementa o cálculo dessa série de forma sequencial. Proponha um algoritmo paralelo para resolver esse problema dividindo a tarefa de estimar o valor de π entre M fluxos de execução independentes.

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots)$$

```
1 double piSequencial (long long n) {
2    double soma = 0.0, fator = 1.0;
3    long long i;
4    for (i = 0; i < n; i++) {
5       soma = soma + fator/(2*i+1);
6       fator = -fator;
7    }
8    return 4.0 * soma;
9 }
```

4. A série infinita mostrada abaixo estima o valor de `log(1+x)` $(-1 < x <= 1)$:

$$\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

Dois programas foram implementados para calcular o valor dessa série (um programa sequencial e outro paralelo) usando N termos. Após a implementação, foram realizadas execuções dos dois programas, obtendo as medidas de tempo apresentadas na Tabela 1.1. A coluna N informa o número de elementos da série, a coluna p informa o número de processadores, e as colunas T_s e T_p informam os tempos de execução do programa sequencial e do programa paralelo, respectivamente.

   (a) Complete a coluna A com os valores de aceleração.

N	p	T_s (s)	T_p (s)	A
1×10^6	1	0,88	0,89	
1×10^6	2	0,88	0,50	
1×10^7	1	8,11	8,34	
1×10^7	2	8,11	4,44	
2×10^7	1	16,21	16,41	
2×10^7	2	16,21	8,84	

Tabela 1.1: Medidas de tempo para calcular log(1+x).

(b) Avalie os resultados obtidos para essa métrica. Considere os casos em que a carga de dados aumenta junto com o número de processadores e os casos isolados onde apenas a carga de trabalho ou o número de processadores aumenta.

5. Assumindo que o tempo para um processador executar um dado programa seja de 1 unidade de tempo e que p seja a fração do programa que pode ser executada em paralelo, a *Lei de Amdahl* [1] estima a aceleração teórica (A_t) que poderá ser obtida por uma versão paralela desse programa usando n processadores:

$$A_t = \frac{1}{(1-p) + (p/n)}$$

Considere uma aplicação composta por 10 atividades (todas consomem o mesmo tempo de execução), das quais 8 podem ser executadas em paralelo.

(a) Se dispusermos de uma máquina com 4 processadores, qual será a aceleração teórica que poderá ser alcançada paralelizando essa aplicação?

(b) Quantos processadores precisariam ser usados para se obter uma aceleração teórica de 4, ou seja, reduzir a 1/4 o tempo de execução da aplicação?

(c) Se apenas 5 atividades pudessem ser executadas em paralelo, qual seria a aceleração teórica considerando novamente uma máquina com 4 processadores?

6. A *Lei de Amdahl* assume que a fração do processamento sequencial é fixa, independente do tamanho do problema. Gustafson [12] argumenta que em muitos problemas práticos essa hipótese não é verdadeira e que a fração do processamento que pode ser feita de forma paralela cresce com o aumento do tamanho do problema. Considerando um programa paralelo que executa com n processadores, e sendo s e p a quantidade de tempo necessária para executar a parte sequencial e paralela do programa respectivamente, Gustafson propõe que o cálculo da aceleração teórica (A_t) seja feito da seguinte forma:

$$A_t = \frac{s + p.n}{s + p}$$

(a) Responda as questões (a), (b) e (c) do exercício anterior usando a equação de Gustafson.

(b) Compare os resultados obtidos para a aceleração teórica usando a equação de Amdahl e a equação de Gustafson.

Capítulo 2

Programação paralela em GPU

2.1 Introdução

As unidades de processamento gráfico (*Graphics Processing Units* — GPUs) estão cada vez mais sendo usadas em conjundo com as CPUs (*Central Processor Units*) para reduzir o tempo de processamento de aplicações em diversas áreas, incluindo computação científica, engenharias, bioinformática, análise numérica, física, entre outras. As GPUs modernas deixaram de ser apenas um dispositivo de processamento gráfico para se tornarem um novo tipo de processador programável massivamente paralelo, com um custo financeiro relativamente baixo [20]. Nos últimos 10 anos, com o rápido crescimento da capacidade de processamento e das facilidades para programar as GPUs, vários problemas com elevada demanda computacional foram mapeados com sucesso pela comunidade científica para executarem nas GPUs, reduzindo significantemente o tempo de processamento requerido para resolver esses problemas [18, 33].

A GPU é um tipo de **acelerador**, ou seja, um dispositivo para o qual a CPU pode delegar parte do processamento de uma aplicação com a finalidade de "acelerar" a sua computação. Os ambientes de programação paralela para GPUs oferecem um conjunto de funcionalidades para alocar e desalocar espaços de memória na GPU, transferir dados de entrada e saída entre a memória principal do computador e a memória da GPU, e disparar a execução de programas na GPU. Cabe ao programador decidir quais partes do código serão executadas na GPU e de que forma a execução dessas partes será dividida entre as milhares de unidades de processamento da GPU.

Como as aplicações gráficas — propulsoras das GPUs — possuem certas demandas e características que as diferenciam das aplicações de propósito geral (por exemplo, manipular e alterar rapidamente posições de memória para acelerar a criação de imagens), as GPUs evoluíram adotando uma terminologia e um modelo de processamento próprios [34]. Por isso, há vários aspectos em que as GPUs se distinguem das CPUs, fazendo com que certos padrões de programação paralela normalmente adotados para as CPUs não necessariamente se apliquem para as GPUs. Da mesma forma, há aplicações que se beneficiem mais da computação paralela em GPU do que outras. As aplicações com forte **paralelismo de dados** são as principais candidatas para execução nas GPUs. Como vimos no Capítulo 1,

essa propriedade permite que sequências de operações aritméticas sejam executadas ao mesmo tempo para processar parcelas distintas dos dados do programa.

Diferentes linguagens e ambientes de programação para GPUS têm sido desenvolvidos. Entre eles destacam-se: CUDA [26, 41] e OpenCL [10, 17]. CUDA (*Compute Unified Device Architecture*) engloba um modelo de programação e uma plataforma de computação paralela específicos para GPUs NVIDIA. OpenCL (*Open Computing Language*) é um arcabouço para programação de diversos tipos de processadores (CPU, FPGA, DSP, GPU) que podem estar disponíveis em uma mesma máquina. A principal vantagem de OpenCL é sua independência de plataforma, isto é, programas em OpenCL podem ser executados em diferentes tipos de processadores e, no caso particular de GPUs, podem executar em GPUs desenvolvidas por diferentes fabricantes, não apenas pela NVIDIA. Entretanto, em função da sua generalidade, a programação com OpenCL é, de forma geral, mais complexa que a programação com CUDA.

Neste capítulo, mostramos como escrever programas paralelos para serem executados em uma GPU. Descrevemos as principais características do hardware das GPUs e como essas características devem ser levadas em conta pelos desenvolvedores de programas para explorar ao máximo a capacidade de processamento paralelo desses dispositivos. Adotamos a plataforma e o modelo de programação CUDA como referências para os conceitos e exemplos de programas que serão apresentados. Acreditamos, porém, que uma vez compreendido os aspectos fundamentais do modelo de programação de CUDA, a curva de aprendizado para outros ambientes de programação, em particular o OpenCL, diminui, uma vez que grande parte dos conceitos adotados por esses ambientes são bastante semelhantes.

2.2 Principais aspectos do hardware das GPUs

O hardware das GPUs é otimizado para tratar problemas com grande demanda computacional e elevada intensidade aritmética (razão entre operações aritméticas e operações de memória), onde um mesmo programa (ou uma mesma sequência de instruções) pode ser executado ao mesmo tempo para processar partes distintas dos dados de entrada. Essas características permitem ocultar a latência de acesso à memória com computações, ao invés do uso de *caches* sofisticadas (memórias de acesso mais rápido) [21]. O hardware aproveita o grande número de fluxos de execução independentes para encontrar trabalho para fazer quando parte desses fluxos está esperando por dados da memória. Como o mesmo programa pode ser executado ao mesmo tempo para processar dados distintos, a unidade de controle é bastante simplificada. Diferente da CPU, a GPU não implementa estratégias de gerência de fluxo mais complexas, como previsão de desvio ou execução especulativa [34].

O projeto das CPUs modernas, por outro lado, mantém ao longo dos anos o propósito de otimizar o desempenho de programas sequenciais. Em função disso, a lógica de controle é mais complexa para que as instruções de um mesmo fluxo de execução sejam processadas em paralelo (sempre que possível) ou fora da sua ordem sequencial, enquanto mantém a aparência de execução sequencial. Para reduzir a latência de acesso aos dados e instruções, uma grande área de memória *cache* é usada. Assim, boa parte do *chip* da CPU é ocupada pela unidade de controle e pela memória *cache*, como ilustra a Figura 2.1.

As CPUs implementam a arquitetura MIMD (*multiple instruction, multiple data*) e visam oferecer um equilíbrio entre capacidade de computação e funcionalidades de propósito geral. As GPUs, por sua vez, implementam a arquitetura SIMT

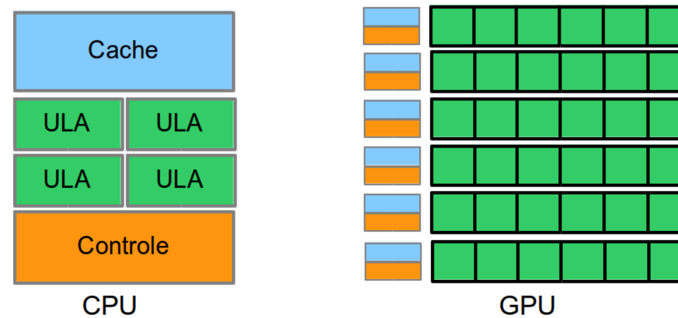


Figura 2.1: Arquitetura CPU versus arquitetura GPU. O projeto da GPU inclui um grande número de pequenas unidades de processamento com pouco espaço para unidades de controle e de cache.

(*single instruction, multiple thread*) caracterizada por permitir que uma mesma instrução seja executada ao mesmo tempo por vários fluxos de execução que atuam sobre elementos distintos do conjunto de dados, visando obter bom desempenho com operaçõs aritméticas massivamente paralelas.

Em razão das suas diferenças arquiteturais, as GPUs são muito mais restritivas que as CPUs, porém muito mais poderosas quando os algoritmos paralelos são cuidadosamente projetados e implementados para a sua arquitetura [20]. Um aspecto importante em relação ao desenvolvimento de aplicações paralelas para GPUs é que, embora as facilidades de programação desses dispositivos tenham evoluído bastante nos últimos anos, o seu modelo de execução e de organização da memória tem grande impacto no desempenho das aplicações. Por isso é fundamental conhecer como se organiza o hardware das GPUs para explorá-las bem.

2.3 Ambiente de programação CUDA

CUDA foi introduzida em 2006 como uma plataforma de computação paralela de propósito geral e um modelo de programação para GPUs NVIDIA [26, 41]. Os códigos que executam na GPU podem ser escritos usando diretamente o conjunto de instruções de máquina de CUDA, chamado **PTX** [25]. 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 [21, 30].

Há ainda versões CUDA para bibliotecas populares, como por exemplo a versão CUDA para a a biblioteca de álgebra linear BLAS [5], denominada `cuBLAS` (*CUDA Basic Linear Algebra Subroutines*) [27]; a biblioteca `CUDA Math` [28] que é uma coleção de funções matemáticas; e `CUSP` uma biblioteca de funções C++ com implementações paralelas para algoritmos de manipulação de matrizes esparsas e de resolução de sistemas lineares esparsos [31, 3]. Uma lista de bibliotecas complementares de CUDA pode ser encontrada em [32]. Diversos cursos sobre programação paralela com CUDA são oferecidos por diferentes universidades [29].

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.

Para apresentar o ambiente de programação de CUDA, vamos usar sua extensão para a linguagem C, denominada **CUDA C**. CUDA C oferece um conjunto de extensões para a linguagem C e uma biblioteca de execução construída acima de uma API C de nível mais baixo (denominada *CUDA driver*).

Nas próximas subseções, apresentaremos três exemplos de aplicações em CUDA C. Cada exemplo será usado para introduzir, de forma incremental, os conceitos básicos da programação paralela em CUDA.

2.3.1 Primeiro exemplo de aplicação em CUDA

Para introduzir os conceitos básicos da programação paralela para GPU, vamos tomar novamente como exemplo a operação SAXPY (soma de vetores): `C = A*k + B`, sendo `A`, `B` e `C` vetores de tamanho `N` e `k` um número.

Adotando a mesma estratégia de divisão do problema em subproblemas menores apresentada no código 1.2, vamos implementar uma função em CUDA para calcular um elemento do vetor de saída: `C[i] = A[i] * k  + B[i], 0 <= i < N`. Em seguida, para calcular todos os elementos de `C`, vamos criar `N` fluxos de execução paralela para executar essa função, cada um deles processando um elemento distinto do vetor. O trecho de código 2.1 mostra como implementar essa solução em CUDA C.

```
1  #define N 1000 //N igual ao número de elementos do vetor
2  //kernel para execução paralela na GPU
3  __global__ void somaVetoresPar(const float *a, const float *b,
       float *c, float k) {
4     int i = threadIdx.x;
5     c[i] = a[i] * k + b[i];
6  }
7  int main() {
8     float a[N], b[N], c[N];
9     //inicializa os vetores a e b
10    ...
11    // invoca o kernel com um bloco de n threads
12    somaVetoresPar<<<1, N>>>(a, b, c, k);
13    ...
14 }
```

Código 2.1: Soma de vetores em CUDA. Versão inicial.

Threads e Kernel Em CUDA, os fluxos de execução independentes que são disparados para executarem na GPU são chamados **threads**. Para codificarmos a tarefa que deverá ser executada por cada thread na GPU, definimos uma função especial — denominada **kernel**. Todas as threads executam a mesma função. O que caracteriza uma função como um kernel é o qualificador `__global__` que precede a assinatura da função. O tipo de retorno de uma função kernel deve sempre ser `void`.

Nas linhas 3 a 6 do código 2.1, temos a definição de uma função kernel denominada `somaVetoresPar`. Essa função recebe como argumentos os ponteiros para os vetores de entrada e saída e o valor da constante `k`. Dentro da função distinguimos qual elemento do vetor cada thread deverá calcular, usando como índice o valor da variável `threadIdx.x`. Essa variável é pre-definida pelo ambiente de execução e consiste de uma estrutura de 3 elementos que permite identificar unicamente cada

thread, usando até três dimensões (x, y e z). Dessa forma, é possível facilmente associar uma thread para cada elemento de um domínio de dados, o qual pode ser unidimensional, bidimensional ou tridimensional. Nesse exemplo, como estamos trabalhando com vetores, usamos apenas a dimensão `x`.

O paralelismo irá aparecer, de fato, quando invocarmos a função kernel explicitando quantas threads deverão executá-la simultaneamente na GPU. Toda chamada para uma função kernel deve especificar sua configuração de execução dentro de uma expressão na forma $<<< D_g, D_b, N_s, S >>>$ que precede a lista de argumentos da função. Por enquanto, vamos precisar usar apenas os dois primeiros argumentos (D_g e D_b). Os outros dois argumentos são opcionais e quando não definidos assumem o valor padrão 0. Juntos, D_g e D_b indicam a quantidade de threads (ou fluxos de execução paralelos) que devem ser criados na GPU para executar a função kernel que está sendo invocada.

As threads criadas na GPU são organizadas em **blocos** (de mesmo tamanho) e os blocos são organizados em uma **grade** (*grid*). O argumento D_b define a quantidade de threads por bloco, enquanto o argumento D_g define o número de blocos na grade. Portanto, $D_g * D_b$ será a quantidade total de threads que executarão o kernel. Na linha 12 do código 2.1 temos a chamada da função kernel com `1` bloco de `N` threads, ou seja, disparamos uma thread para calcular cada elemento do vetor.

A chamada para a função kernel retornará com falha se D_g ou D_b forem maiores que o tamanho máximo permitido. De acordo com o manual de programação de CUDA, o tamanho máximo de um bloco é de 1024 threads [21] (esse valor pode variar de um modelo de placa GPU para outro). Então, no código 2.1, o tamanho máximo dos vetores (valor de `N`) deverá ser de 1024 elementos, uma vez que cada elemento será processado por uma thread. Entretanto, podemos ter vários blocos de até 1024 threads, atribuindo um valor maior que 1 para o argumento D_g. Se, por exemplo, definirmos D_g como sendo 1000, poderemos ter vetores de até 1.024.000 elementos. Mas para isso, precisamos alterar também a função kernel. Como vimos, usamos o identificador da thread (`threadIdx.x`) para definir qual elemento do vetor a thread deverá processar. O que ocorre é que esse identificador é único apenas dentro de um bloco. Se tivermos dois blocos de 1024 threads, por exemplo, os valores de `threadIdx.x` em ambos os blocos irão variar de 0 a 1023. Sendo assim, para identificar cada thread unicamente, precisamos identificar em qual bloco ela está.

Da mesma forma que as threads, cada bloco possui um identificador único dentro da grade, chamado `blockIdx`, que é definido em tempo de execução e pode ser acessado de dentro do kernel. Assim como os blocos, uma grade pode ser unidimensional, bidimensional ou tridimensional. Desse modo, cada bloco dentro da grade pode ser identificado por até três dimensões (x, y, z). Por fim, a informação sobre o tamanho de cada bloco também é acessível de dentro do kernel através da variável `blockDim`.

O código 2.2 altera o código 2.1 permitindo somar vetores com dimensão superior ao limite inicial de 1024 elementos. Nessa nova versão do programa, fixamos o número de threads em cada bloco em 1024 (linha 12) e calculamos o número de blocos na grade de acordo com o valor de `N` (linha 13). Na função kernel, usamos o identificador do bloco (`blockIdx.x`), a dimensão de cada bloco (`blockDim.x`) e o identificador da thread dentro do bloco (`threadIdx.x`) para calcular o identificador único da thread na grade (linha 3).

```
1 //kernel para execução paralela na GPU
2 __global__ void somaVetoresPar(const float *a, const float *b,
```

```
    float *c, float k, int n) {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     if(i < n) {
5        c[i] = a[i] * k + b[i];
6     }
7  }
8  int main() {
9     float a[N], b[N], c[N];
10    //inicializa os vetores a e b
11    ...
12    int n_threads = 1024; //número de threads por bloco
13    int n_blocos = (N + n_threads-1) / n_threads; //número de
         blocos
14    somaVetoresPar<<<n_blocos, n_threads>>>(a, b, c, k, N);
15    ...
16 }
```

Código 2.2: Soma de vetores em CUDA. N é definido com a dimensão dos vetores. Versão que usa vários blocos de threads.

Uma vez que fixamos o tamanho de cada bloco e definimos o número de blocos de acordo com a dimensão dos vetores de entrada (valor de `N`), pode ocorrer que algumas threads não tenham elementos para processar. Por exemplo, se o valor de `N` for 2000, serão criados dois blocos de 1024 threads, totalizando 2048 threads. Então, as últimas 48 threads não terão elementos para processar. Para evitar erros de acesso a posições inválidas da memória, na linha 4 do código 2.2 verificamos se o identificador da thread é válido para a execução corrente, ou seja, se há elemento para processar nessa posição do vetor de saída. Se houver, o elemento será calculado, caso contrário a thread não executará nenhum processamento.

Espaço de memória da GPU Todas as variáveis acessadas pelas threads dentro da função kernel precisam estar alocadas no espaço de memória da GPU. No código 2.2, por exemplo, as threads recebem como parâmetros da função kernel um ponteiro para cada um dos vetores `a`, `b` e `c` (indicando o endereço de memória onde o primeiro byte de cada vetor está armazenado). Para que as threads possam acessar (ler ou escrever) qualquer elemento desses vetores, é necessário que o endereço apontado seja um endereço válido dentro espaço de memória da GPU. As demais variáveis (k e n), passadas por valor, serão automaticamente copiadas para a área de memória local das threads.

Assumindo que os vetores de entrada `A` e `B` serão inicializados na CPU e que queremos usar o vetor de saída `C` para continuar o processamento na CPU, é preciso que os vetores `A` e `B` sejam previamente transferidos da memória da CPU para a memória da GPU, antes da chamada do kernel. Posteriormente, quando todas as threads finalizarem suas execuções, será necessário transferir os resultados armazenados no vetor `C` da memória da GPU para a memória da CPU. Dado esse cenário, o fluxo de execução do programa CUDA deve seguir os seguintes passos:

1. reservar espaço de memória na CPU para os dados de entrada e saída e executar as inicializações necessárias;
2. reservar espaço de memória na GPU para os dados de entrada e saída;
3. transferir os dados de entrada da memória da CPU para a memória da GPU;
4. disparar o kernel;

5. transferir os dados processados da memória da GPU para a memória da CPU;

6. exibir os resultados e executar as finalizações necessárias.

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. O código 2.3 completa o código 2.2 com os passos 1, 2, 3, 5 e 6 listados acima.

```
1  #include <stdio.h>
2  //kernel para execução paralela na GPU
3  __global__ void somaVetoresPar(const float *a, const float *b,
       float *c, float k, int n) {
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5     if(i < n) {
6        c[i] = a[i] * k + b[i];
7     }
8  }
9  int main() {
10    float *h_a, *d_a;
11    float *h_b, *d_b;
12    float *h_c, *d_c;
13    n_bytes = N * sizeof(float); //N é a dimensão dos vetores

15    //aloca memória na CPU (host)
16    h_a = (float*) malloc (n_bytes);
17    h_b = (float*) malloc (n_bytes);
18    h_c = (float*) malloc (n_bytes);

20    //inicializa os vetores h_a e h_b
21    ...
22    //aloca espaço para os vetores na GPU
23    cudaMalloc((void**) &d_a, n_bytes);
24    cudaMalloc((void**) &d_b, n_bytes);
25    cudaMalloc((void**) &d_c, n_bytes);

27    //copia os vetores de entrada da CPU para a GPU (host para
         device)
28    cudaMemcpy(d_a, h_a, n_bytes, cudaMemcpyHostToDevice);
29    cudaMemcpy(d_b, h_b, n_bytes, cudaMemcpyHostToDevice);

31    //dispara o kernel
32    int n_threads = 1024; //número de threads por bloco
33    int n_blocos = (N + n_threads-1) / n_threads; //número de
         blocos
34    somaVetoresPar<<<n_blocos,n_threads>>>(d_a, d_b, d_c, k, N);

36    //copia o resultado da GPU para a CPU (device para host)
37    cudaMemcpy(h_c, d_c, n_bytes, cudaMemcpyDeviceToHost)

39    //libera a memória na GPU
40    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

42    //usa o vetor C
43    ...
44    //libera a memória na CPU
45    free(h_a); free(h_b); free(h_c);
```

```
47     //limpa o estado da GPU e termina
48     cudaDeviceReset();
49     return 0;
50 }
```

Código 2.3: Soma de vetores em CUDA. Versão completa do código.

Nesse código, criamos variáveis específicas para armazenar os vetores `A`, `B` e `C` na memória da CPU (denominadas `h_a`, `h_b` e `h_c`) e na memória da GPU (denominadas `d_a`, `d_b` e `d_c`) (linhas 10, 11 e 12). Na linha 13 calculamos o tamanho (em bytes) de cada um dos vetores.

Nas linhas 16, 17 e 18 alocamos espaço de memória para os vetores na CPU, usando a função `malloc` de C. Nas linhas 23, 24 e 25 usamos uma função similar de CUDA, chamada `cudaMalloc`, para alocar espaço de memória para os vetores na GPU. A função `cudaMalloc` permite alocar uma sequência contínua de bytes na memória da GPU. Sua assinatura completa é:

`cudaError_t cudaMalloc (void **devPtr, size_t size)`

O parâmetro `devPtr` — passado por referência — recebe o endereço do primeiro byte da sequência de bytes alocada na memória da GPU. A quantidade de bytes que serão alocados é definida pelo parâmetro `size`.

Para transferir/copiar dados entre as memórias da CPU e GPU, CUDA oferece uma única função chamada `cudaMemcpy`, cuja assinatura é:

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

Essa função recebe como argumentos o endereço de destino (`dst`) e o endereço de origem dos dados (`src`), a quantidade de bytes que deve ser copiada (`count`) e a direção da cópia (`kind`) (por exemplo, da CPU para a GPU ou da GPU para a CPU). Para copiar dados da CPU (*host*) para a GPU (*device*), o parâmetro `kind` deve receber o valor `cudaMemcpyHostToDevice`. Para copiar dados na direção inversa, da GPU para a CPU, o parâmetro `kind` deve receber o valor `cudaMemcpyDeviceToHost`. Também é possível usar a função `cudaMemcpy` para copiar dados entre áreas distintas da GPU (nesse caso `kind` recebe o valor `cudaMemcpyDeviceToDevice`) ou para copiar dados entre áreas distintas da CPU (nesse caso `kind` recebe o valor `cudaMemcpyHostToHost`).

Nas linhas 28 e 29 do código 2.3, usamos a função `cudaMemcpy` para copiar os dados dos vetores de entrada `A` e `B` da memória da CPU para a memória da GPU. Posteriormente, na linha 37 usamos a mesma função para copiar os dados do vetor de saída `C` da memória da GPU para a memória da CPU. Importante observar que na linha 34 chamamos a função kernel passando como parâmetros para a função os endereços dos vetores de entrada e saída que foram alocados na memória da GPU (`d_a`, `d_b` e `d_c`).

Uma vez que os dados carregados para a GPU não serão mais usados, podemos liberar os espaços de memória alocados usando a função `cudaFree`, como mostrado na linha 45 do código 2.3. A assinatura da função `cudaFree` é a seguinte: `cudaError_t cudaFree(void *devPtr)`. Ela recebe como único parâmentro o endereço inicial da área de memória que deve ser liberada (valor que foi devolvido pela função `cudaMalloc`). Antes do programa terminar, os espaços de memória alocados na CPU também são liberados usando a função `free` da linguagem C.

Verificação de erros Todas as funções de CUDA que vimos até agora retornam códigos de erros para indicar se a operação requerida foi executada com sucesso ou não. Recomenda-se sempre verificar o retorno dessas funções.

No código 2.4 apresentamos uma nova versão do código 2.3 onde implementamos uma função *wrapper* (função que serve para encapsular as chamadas de outras funções) para verificar e exibir os erros ocorridos em todas as chamadas das funções de CUDA. Essa função foi definida na forma de uma macro, com o nome `CUDA_SAFE_CALL` (linhas 2 a 9).

A macro `CUDA_SAFE_CALL` invoca a função recebida como argumento e armazena o código de erro recebido como retorno (linha 3). Quando a operação em CUDA é realizada com sucesso, as funções retornam o código `cudaSuccess`. Em qualquer outro caso (linha 4), diferentes códigos de erros podem ser emitidos para distinguir o tipo de erro ocorrido. A macro `CUDA_SAFE_CALL` imprime na saída padrão de erro as seguintes informações: o arquivo fonte e a linha de código onde o erro ocorreu e uma descrição do erro que foi detectado (linhas 5 e 6). Em seguida, a macro encerra o programa (linha 7).

```
1  //wrapper para checar erros nas chamadas de funções de CUDA
2  #define CUDA_SAFE_CALL(call) { \
3     cudaError_t err = call;     \
4     if(err != cudaSuccess) {    \
5        fprintf(stderr,"Erro no arquivo '%s', linha %i: %s.\n", \
6                __FILE__, __LINE__, cudaGetErrorString(err));   \
7        exit(EXIT_FAILURE);
8     }
9  }
10 //kernel para execução paralela na GPU
11 __global__ void somaVetoresPar(const float *a, const float *b,
      float *c, float k, int n) {
12    int i = blockIdx.x * blockDim.x + threadIdx.x;
13    if(i < n) {
14       c[i] = a[i] * k + b[i];
15    }
16 }
17 int main() {
18    float *h_a, *d_a;
19    float *h_b, *d_b;
20    float *h_c, *d_c;
21    n_bytes = N * sizeof(float); //N é a dimensão dos vetores

23    //aloca memória na CPU (host)
24    h_a = (float*) malloc(n_bytes);
25    h_b = (float*) malloc(n_bytes);
26    h_c = (float*) malloc(n_bytes);
27    if(h_a==NULL)||(h_b==NULL)||(h_c==NULL) exit(EXIT_FAILURE);

29    //inicializa os vetores h_a e h_b
30    ...
31    //aloca espaço para os vetores na GPU
32    CUDA_SAFE_CALL(cudaMalloc((void**) &d_a, n_bytes));
33    CUDA_SAFE_CALL(cudaMalloc((void**) &d_b, n_bytes));
34    CUDA_SAFE_CALL(cudaMalloc((void**) &d_c, n_bytes));

36    //copia os vetores da CPU para a GPU (host para device)
```

```
37    CUDA_SAFE_CALL(cudaMemcpy(d_a,h_a,n_bytes,
          cudaMemcpyHostToDevice));
38    CUDA_SAFE_CALL(cudaMemcpy(d_b,h_b,n_bytes,
          cudaMemcpyHostToDevice));

40    //dispara o kernel
41    int n_threads = 1024; //número de threads por bloco
42    int n_blocos = (N+n_threads-1)/n_threads;//número de blocos
43    somaVetoresPar<<<n_blocos,n_threads>>>(d_a, d_b, d_c, k, N);
44    CUDA_SAFE_CALL(cudaGetLastError());

46    //copia o resultado da GPU para a CPU (device para host)
47    CUDA_SAFE_CALL(cudaMemcpy(h_c,d_c,n_bytes,
          cudaMemcpyDeviceToHost))

49    //libera a memória na GPU
50    CUDA_SAFE_CALL(cudaFree(d_a));
51    CUDA_SAFE_CALL(cudaFree(d_b));
52    CUDA_SAFE_CALL(cudaFree(d_c));
53    ...
54 }
```

Código 2.4: Soma de vetores em CUDA. N é definido com a dimensão dos vetores. Versão com tratamento de erros.

Para verificar se ocorreu algum erro ao disparar o kernel, acrescentamos na linha 44 do código 2.4 uma chamada para a função `cudaGetLastError`. Essa função não recebe argumentos e retorna o último erro que foi produzido pelas chamadas de função executadas na GPU. Como a função kernel não tem retorno, essa é uma forma de verificar se houve algum erro na sua invocação. Por exemplo, se chamarmos a função kernel com um número de threads por bloco acima do valor máximo permitido, o kernel não será executado e a função `cudaGetLastError` devolverá o erro encontrado. Para funções assíncronas (como é o caso da função kernel), que retornam para a CPU antes da execução finalizar na GPU, o código de erro reporta apenas erros relacionados à passagem de parâmetros ou que ocorreram antes da chamada da função (gerados por tarefas enfileiradas antes e que já terminaram). A operação de funções síncronas e assíncronas em CUDA é explicada mais adiante nesta seção.

Tempo de execução do kernel Diferentes métricas podem ser usadas para avaliar o desempenho de uma aplicação paralela, como discutido na seção 1.5. A medida que serve de base para todas as métricas é o tempo de execução do programa. No caso do problema de soma de vetores, estaríamos interessados em avaliar o quanto a GPU permite melhorar o desempenho da execução da função central, que dados os vetores de entrada, calcula o vetor de saída. No código CUDA, essa função está implementada no kernel e o seu tempo de execução corresponde ao intervalo de tempo decorrido desde o lançamento do kernel até todas as threads completarem a sua execução.

Uma questão importante aqui é como saber quando todas as threads terminaram. Isso porque quando disparamos o kernel a partir da CPU, essa chamada é **assíncrona** (do ponto de vista da CPU), ou seja, a CPU não fica bloqueada até que o kernel termine. Ao invés disso, tão logo a GPU enfileira a tarefa de processar o kernel, o controle retorna para a CPU que pode continuar sua execução realizando

outras tarefas que não dependem da informação que será devolvida pelo kernel. A ideia é permitir o máximo de paralelismo entre a CPU e a GPU.

Por outro lado, há funções de CUDA que são bloqueantes ou **síncronas** (aguardam a conclusão das operações anteriores lançadas na GPU e só retornam depois de concluírem suas operações). Um exemplo são as operações de alocação de memória e transferência de dados. Nesse caso, a CPU não pode prosseguir sua execução enquanto as funções chamadas não retornarem. Se voltarmos ao código 2.4, podemos verificar que a sequência de execução do programa funcionará corretamente pois as operações de alocação de memória (linhas 32 a 34) e de transferência de dados da CPU para a GPU (linhas 37 e 38) serão necessariamente concluídas antes da CPU disparar o kernel (linha 43). Por outro lado, como o disparo do kernel é assíncrono, a CPU irá executar logo em seguida as próximas sentenças do código (linhas 44 e 47). No entanto, na linha 47 a CPU novamente terá que aguardar pois a função `cudaMemcpy` só começa a transferência de dados depois que a função kernel é concluída na GPU.

Se medirmos o tempo imediatamente antes de lançar o kernel (entre as linhas 42 e 43) e depois da transferência de dados da linha 47, teremos certeza de termos englobado o tempo total de execução do kernel. No entanto, não teremos como separar o tempo de transferência de dados da GPU para a CPU (que pode ser bastante significativo).

A alternativa recomendada no manual de programação de CUDA para medir o tempo de execução de um kernel é fazer uso de **eventos** [21]. Uma aplicação pode registrar um evento em qualquer parte do código e depois verificar em qual instante de tempo ele foi concluído. Um evento em CUDA é concluído quando todas as tarefas que precederam o evento e aguardam execução pela GPU na mesma fila são finalizadas.

CUDA define um tipo especial para eventos, denominado `cudaEvent_t`, e várias funções para manipulação de eventos. A função `cudaEventCreate` cria o evento que é passado por referência. A função `cudaEventRecord` registra um evento para execução na fila da GPU, enquanto a função `cudaEventSynchronize` espera até que todas as tarefas que precederam o evento na fila da GPU sejam completadas.

Para calcular o tempo decorrido entre o registro do primeiro evento e a conclusão do segundo, CUDA oferece a função `cudaEventElapsedTime`. A assinatura dessa função é a seguinte:

```
cudaError_t cudaEventElapsedTime(float *ms, cudaEvent_t start,
                                 cudaEvent_t end)
```

No primeiro parâmetro será retornado o tempo transcorrido entre a execução dos dois eventos, em milisegundos. O segundo e terceiro parâmetros recebem, respectivamente, o evento que foi registrado primeiro e o evento que foi registrado por último. A função calcula o intervalo de tempo transcorrido entre eles.

No código 2.5 destacamos o trecho de código que foi acrescentado ao código 2.4 para medir o tempo de execução do kernel.

```
1    ...
2    //dispara o kernel
3    int n_threads = 1024; //número de threads por bloco
4    int n_blocos = (N+n_threads-1)/n_threads;//número de blocos

6    //cria eventos auxiliares para tomada de tempo de execução
         do kernel
```

```
7     cudaEvent_t start, stop;
8     CUDA_SAFE_CALL(cudaEventCreate(&start));
9     CUDA_SAFE_CALL(cudaEventCreate(&stop));

11    //dispara o primeiro evento antes de disparar o kernel
12    CUDA_SAFE_CALL(cudaEventRecord(start));

14    somaVetoresPar<<<n_blocos,n_threads>>>(d_a, d_b, d_c, k, N);
15    CUDA_SAFE_CALL(cudaGetLastError());

17    //dispara o segundo evento após disparar o kernel
18    CUDA_SAFE_CALL(cudaEventRecord(stop));

20    //aguarda até o segundo evento ser completado
21    CUDA_SAFE_CALL(cudaEventSynchronize(stop));

23    //calcula o tempo decorrido entre os dois eventos que
          englobaram a execução do kernel
24    float delta_eventos = 0;
25    CUDA_SAFE_CALL(cudaEventElapsedTime(&delta_eventos, start,
          stop));
26    ...
```

Código 2.5: Trecho do programa de soma de vetores em CUDA. Versão com tomada do tempo de execução do kernel.

Compilação e execução Como vimos nos exemplos de códigos mostrados nesta seção, um programa CUDA é um código unificado que engloba as partes do programa que executarão na CPU e na GPU. Para gerar os códigos executáveis apropriadamente, o processo de compilação envolve vários passos. A ferramenta `nvcc` (NVIDIA C) [24] controla a execução dessa sequência de passos, ocultando do programador detalhes intrínsecos do processo de compilação de CUDA.

Os trechos de código com a sintaxe <<< ... >>> (que estão associados às chamadas de kernels) são substituídos por chamadas de funções especiais para compilar e disparar os códigos dos kernels. O código fonte modificado é então compilado com ferramentas de compilação apropriadas para gerar instruções de máquina para a CPU.

Essa opção por gerar o código para a GPU em tempo de execução (chamado *just-in-time compilation*) demanda um tempo maior para iniciar de fato a execução do programa quando ele é disparado, mas por outro lado permite que a aplicação se beneficie de novas melhorias providas pelo controlador do dispositivo. Para evitar a recompilação do código em execuções consecutivas do programa, o controlador guarda uma cópia do código binário gerado. Essa cópia só é invalidada quando o controlador do dispositivo sofre alguma atualização [21].

O compilador `nvcc` pode ser invocado diretamente da linha de comando ou a partir de um ambiente de desenvolvimento integrado (IDE), como o *Nsight* [23] (uma IDE para desenvolver aplicações CUDA em Linux e MacOS-X). Normalmente, o `nvcc` e outras ferramentas auxiliares para compilação e execução de aplicações CUDA são instaladas junto com o *toolkit* de CUDA.

O `nvcc` oferece várias opções de compilação. Informações mais detalhadas sobre essas opções e sobre todas as etapas do processo de compilação de programas CUDA C podem ser obtidas no manual do `nvcc` [24]. Informações sobre como instalar o *toolkit* de CUDA podem ser encontradas no manual de instalação de CUDA [22].

Avaliação do desempenho da aplicação Para finalizar a discussão sobre a paralelização do problema de soma de vetores em CUDA, vamos avaliar qual foi o desempenho obtido com a última versão apresentada (código 2.5).

Primeiro, criamos um arquivo chamado `somaVetores.cu` com a versão completa do código. (Os arquivos fonte com aplicações CUDA C devem ter a extensão `.cu`.) Nessa versão, permitimos que o usuário informe o tamanho dos vetores e o valor da constante `k` como argumentos para a chamada do programa. Por simplicidade, os vetores são inicializados dentro da aplicação. Compilamos o programa usando o `nvcc` a partir da linha de comando: `nvcc somaVetores.cu -o somaVetores`. O arquivo executável gerado (`somaVetores`) foi então invocado vários vezes, com diferentes valores para o tamanho dos vetores de entrada.

Para essa avaliação, fixamos o tamanho dos blocos de threads em 256 e variamos o valor de N (tamanho dos vetores) de 10^2 até 10^7. Usamos uma máquina com processador Intel i7-4770 com o sistema operacional Ubuntu 14.04.3 e uma GPU NVIDIA GeForce GTX 770 (8 multiprocessadores de 192 *cores*) com a versão 7.5 de CUDA instalada. Todas as métricas de avaliação usadas foram apresentadas na seção 1.5.

Para avaliar o ganho de desempenho da versão paralela, implementamos uma função em C que soma os vetores de forma sequencial. A Tabela 2.1 apresenta os resultados obtidos. A coluna `Ts` contém o tempo de execução da função sequencial na CPU, enquanto a coluna `Tp` contém o tempo de execução do kernel na GPU. Podemos observar, pelos valores de **aceleração** obtidos, que a versão paralela só começa a ter resultados melhores que a versão sequencial quando o tamanho dos vetores de entrada é maior que 10^3. Essa é uma situação bastante comum na programação paralela. Os custos para criar e gerenciar vários fluxos de execução paralela só são compensados quando o domínio dos dados de entrada ultrapassa um determinado limiar, o qual depende da aplicação e do hardware utilizado.

N	**Ts (mseg)**	**Tp (mseg)**	**Aceleração**	**Instruções**	**Vazão (seq) (GFlops)**	**Vazão (par) (GFlops)**
10^2	0,000001	0,000019	0,05	2×10^2	0,2097	0,0105
10^3	0,000011	0,000021	0,52	2×10^3	0,1824	0,0952
10^4	0,000136	0,000028	4,86	2×10^4	0,1469	0,7143
10^5	0,001382	0,000042	32,91	2×10^5	0,1447	4,7619
10^6	0,002903	0,000073	39,77	2×10^6	0,6889	27,3973
10^7	0,024477	0,000663	36,92	2×10^7	0,8171	30,1659

Tabela 2.1: Medidas de desempenho do programa de soma de vetores sem considerar o tempo de transferência de dados entre a memória da CPU e a memória da GPU.

No cálculo da **vazão computacional** (ver equação (2.3.1)), contabilizamos 2 instruções de ponto flutuante para cada elemento do vetor de saída (uma operação de multiplicação e outra de adição). A penúltima coluna na tabela mostra a vazão computacional da função sequencial, enquanto a última coluna mostra a vazão computacional do kernel. Como exemplo, para calcular a vazão computacional do kernel quando $N = 10^2$ (primeira linha da tabela), fizemos:

$$V = 2 \cdot 10^2/(0.000019 \cdot 10^9) = 0,0105 GFlops \tag{2.3.1}$$

Podemos observar que para a versão sequencial, a vazão varia entre $0,2$ e $0,8$, já para a versão paralela a vazão cresce com o aumento da carga de trabalho. Isso mos-

tra que a GPU funciona melhor com cargas de trabalho mais altas porque consegue obter índices mais elevados de ocupação das suas unidades de processamento.

A Tabela 2.2 apresenta os resultados obtidos incluindo-se no tempo de execução paralela o tempo de transferência dos vetores de entrada da memória da CPU para a memória da GPU — antes da execução do kernel — e o tempo de transferência do vetor de saída da memória da GPU para a memória da CPU — depois da execução do kernel. Neste caso, observamos que o tempo da execução paralela aumenta consideravelmente, fazendo com que o ganho de desempenho seja quase nulo para o problema em questão. Isso mostra que o custo de transferência de dados entre as memórias da CPU e da GPU pode ser bastante significativo, devendo ser levado em conta no projeto de algoritmos paralelos para GPU.

N	**Ts (mseg)**	**Tp (mseg)**	**Aceleração**	**Instruções**	**Vazão (seq) (GFlops)**	**Vazão (par) (GFlops)**
10^2	0,000001	0,000050	0,02	2×10^2	0,2097	0,0040
10^3	0,000011	0,000070	0,16	2×10^3	0,1824	0,0286
10^4	0,000136	0,000083	1,64	2×10^4	0,1469	0,2410
10^5	0,001382	0,000359	3,85	2×10^5	0,1447	0,5571
10^6	0,002903	0,002165	1,34	2×10^6	0,6889	0,9238
10^7	0,024477	0,018532	1,32	2×10^7	0,8171	1,0792

Tabela 2.2: Medidas de desempenho do programa de soma de vetores considerando o tempo de transferência de dados entre a memória da CPU e a memória da GPU.

Os resultados apresentados na Tabela 2.2 servem para mostrar que além do volume de dados processados, a complexidade do processamento (em termos do número de operações realizadas para cada dado de entrada) é também um aspecto relevante para que um problema seja um candidato para paralelização em GPU. O volume de dados processados e a complexidade do processamento realizado por cada thread deve compensar os custos associados com a execução paralela e com as transferências de dados.

De forma geral, operações básicas como essa de soma de vetores (SAXPY) que tomamos como exemplo introdutório de paralelização não aparecem de forma isolada, mas como parte de problemas maiores. Nesse caso, a sua paralelização pode se justificar (de acordo com os dados da Tabela 2.1), uma vez que os dados de entrada já estão na memória da GPU.

2.3.2 Sincronização de threads em CUDA

CUDA oferece um mecanismo de **sincronização por barreira** (veja seção 1.3) no nível de **blocos**, por meio do comando `__syncthreads`. Quando esse comando é chamado dentro de um kernel, ele define um ponto de bloqueio/barreira para todas as threads do mesmo bloco. Isso significa que as threads só podem avançar, ou seja, continuar a execução da sua sequência de instruções, depois que todas as demais threads do bloco tiverem chegado a esse ponto do código.

Como vimos, a sincronização por barreira é necessária para modelar condições lógicas de um problema. Por exemplo, em métodos numéricos iterativos, cada iteração deve ser completamente concluída antes do processamento avançar para a próxima iteração. Há outras situações em que a sincronização por barreira não é

uma demanda do problema em si, mas torna-se necessária para garantir o funcionamento correto do algoritmo paralelo proposto para resolver esse problema.

Para mostrar o uso das alternativas de sincronização das threads em CUDA, vamos tomar novamente como exemplo o problema de **somar os elementos de um vetor**, discutido na seção 1.3. Trata-se de um problema bastante simples mas que pode aparecer como parte de um problema maior, como por exemplo, o cálculo do produto interno de dois vetores. Dependendo da dimensão do vetor de entrada, a paralelização desse subproblema tem potencial para trazer ganhos de desempenho significativos para a aplicação mais geral.

No código 1.7, mostramos um algoritmo sequencial para somar os elementos de um vetor. A variável `soma` vai acumulando os valores dos elementos do vetor do primeiro ao último. Na seção 1.3 chegamos a discutir uma estratégia para paralelizar esse algoritmo, dividindo o conjunto de valores do vetor em grupos consecutivos de elementos que seriam somados separadamente por cada fluxo de execução. Ao final, usando algum mecanismo de sincronização, cada fluxo de execução deveria somar seu resultado na variável global. Em princípio, essa é uma boa estratégia de paralelização para a execução em CPUs *multicore*, pois todos os processadores ficarão ocupados e poderão trabalhar simultaneamente. Além disso, como são poucas unidades de processamento, o custo da sincronização no final fica reduzido. Então, nesse caso, dividir a tarefa total igualmente entre os processadores garante um nível adequado de paralelismo.

No caso das GPUs temos centenas ou milhares de unidades de processamento. Podemos pensar inicialmente que adotar a mesma estratégia de dividir os elementos do vetor igualmente entre as unidades de processamento seria uma boa alternativa. Entretanto, como vimos na seção 2.2, as GPUs ocultam a latência de acesso à memória mantendo muitos fluxos de execução ativos e alternando rapidamente a execução entre eles. Sendo assim, o ideal é disparar um número de threads com subtarefas independentes bastante maior que o número de unidades de processamento para que a GPU sempre encontre trabalho para elas. A pergunta a ser feita então é: qual é a menor tarefa para esse problema? Podemos considerar a **soma de um par de elementos do vetor** como a tarefa mais elementar desse problema. Seguindo esse raciocínio, vamos dividir o problema inicial em $N/2$ subtarefas (de somar dois elementos), que serão alocadas para $N/2$ threads. Na etapa seguinte, vamos continuar trabalhando com tarefas elementares de somar dois valores, mas agora usando como entrada os resultados das somas anteriores. Esse processo deve continuar se repetindo até que reste apenas um valor que será a soma total. A Figura 2.2 ilustra essa ideia. A linha horizontal tracejada destaca a necessidade de se aguardar a conclusão de todas as somas de pares de valores da etapa atual antes de avançar para a etapa seguinte, uma vez que os resultados dessas somas são usados como valores de entrada para as somas realizadas na próxima etapa.

O valor final da soma é armazenado na última posição do vetor. Por simplicidade, assume-se que o número de elementos do vetor é uma potência de 2. No primeiro passo, os elementos são somados par-a-par e o resultado é armazenado nos campos de índice ímpar. No segundo passo, apenas os elementos de índice ímpar (que armazenam as somas da etapa anterior) são somados par-a-par e o resultado é armazenado no elemento de maior índice. No terceiro passo, os elementos dos índices 3 e 7 são somados e o resultado é armazenado no índice 7, o qual passa a ter a soma de todos os elementos do vetor. (Nesse caso, o vetor de entrada foi modificado para armazenar as somas intermediárias. Outra alternativa seria usar um vetor auxiliar para não perder o conteúdo do vetor original.)

Se olharmos do ponto de vista do último elemento — o qual deverá conter a soma

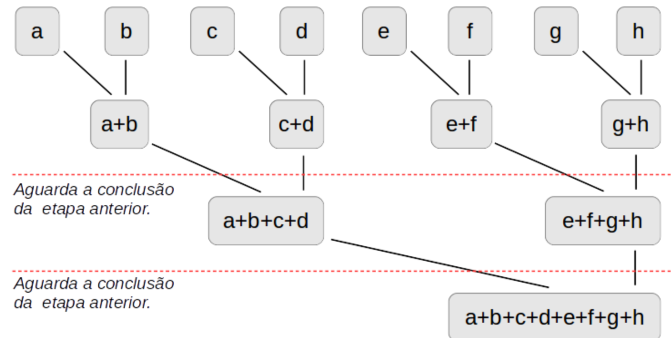


Figura 2.2: Estratégia para somar os elementos de um vetor de forma paralela.

total dos elementos ao final da execução — foram necessárias 3 iterações sobre o vetor para somar todos os elementos, ao invés de 7 do algoritmo sequencial. Isso porque durante essas 3 iterações, outras posições do vetor estavam sendo processadas ao mesmo tempo e os resultados parciais dessas somas foram sendo incorporados a cada iteração. Assim, reduzimos a complexidade do algoritmo (em termos do número de passos necessários para concluir a soma total) de $O(N)$ para $O(\log_2 N)$, considerando vários fluxos de execução trabalhando ao mesmo tempo.

A necessidade de **sincronizar a execução das threads** aparece nesse problema porque precisamos garantir que os cálculos das somas feitos em uma etapa tenham sido concluídos para então serem usados como entrada na etapa seguinte, ou seja, as threads não podem avançar para a etapa seguinte enquanto todas elas não tiverem concluído a etapa atual (como ilustrado na Figura 2.2 pelas linhas horizontais tracejadas). Isso é necessário porque o valor calculado por uma thread será usado na etapa seguinte por outra thread.

O código 2.6 apresenta uma implementação aproximada desse algoritmo em CUDA, usando o comando `__syncthreads`.

```
1  //kernel para execução paralela na GPU
2  __global__ void somaElementosVetorPar(float *a) {
3     int i = threadIdx.x;
4     int n = blockDim.x;
5     int salto;
6     for(salto=1; salto<n; salto*=2) {
7        float aux;
8        if(i >= salto) {
9           aux = a[i-salto];
10       }
11       __syncthreads();
12       if(i >= salto) {
13          a[i] = aux + a[i];
14       }
15       __syncthreads();
16    }
17 }
```

Código 2.6: Kernel para somar os elementos de um vetor em CUDA. Esse kernel deverá ser disparado com 1 bloco de N threads (N igual ao número de elementos do vetor de entrada).

O comando `for` da linha 6 calcula o número total de iterações necessárias para se chegar ao valor final da soma ($\log_2 n$). A variável `salto` começa com o valor 1 e é multiplicada a cada iteração por 2. Enquanto o valor de `salto` for menor que n (número de elementos do vetor), mais uma iteração do algoritmo deve ser realizada.

Dentro de cada iteração, o valor contido na variável `salto` é usado pelas threads para selecionar o elemento que deverá ser somado com o valor na posição do vetor referente ao seu índice. Na primeira iteração, quando `salto` é igual a 1, as threads devem somar o elemento que está uma posição antes da sua. Na segunda iteração, quando `salto` é igual a 2, as threads devem somar o elemento que está duas posições antes da sua, e assim sucessivamente enquanto existirem posições válidas na distância definida por `salto`.

Podemos observar que o processamento realizado por um subconjunto das threads no código 2.6 não será aproveitado para o cálculo do resultado final. Na primeira iteração do algoritmo, por exemplo, o processamento das threads de índices pares (correspondentes aos elementos a, c, e e g da Figura 2.2) não será utilizado, uma vez que os elementos somados por essas threads já foram incluídos nas somas realizadas pelas threads de índices ímpares (correspondentes aos elementos b, d, f e h da Figura 2.2). Discutiremos mais adiante porque optamos por essa estratégia de implementação.

O processamento de cada iteração é dividido em duas partes. Na primeira parte, cada thread lê o valor atual do elemento na distância determinada pelo valor de `salto` e armazena esse valor na variável auxiliar `aux`. Esse armazenamento temporário é necessário para garantir que as threads não leiam valores incorretos, que já foram ou estão sendo atualizados por outras threads ao mesmo tempo. Além disso, para garantir que todas as threads terminaram de ler as posições devidas antes delas serem atualizadas, é necessário incluir uma sincronização de barreira ao final dessa parte, o que é feito com o comando `__syncthreads` da linha 11. Assim, nenhuma thread pode executar a instrução seguinte, que altera as posições do vetor, antes das demais terem concluído a leitura dos valores correntes.

Na segunda parte, cada thread atualiza a posição do vetor correspondente ao seu índice somando o valor atual dessa posição com o valor armazenado anteriormente na variável `aux`. Agora, as threads não podem avançar para a próxima iteração antes de todas as threads concluírem as somas parciais, uma vez que esses valores serão usados no próximo passo. O comando `__syncthreads` da linha 15 garante essa sincronização.

Como a cada iteração o valor do salto é multiplicado por 2, as threads terminarão suas tarefas em momentos diferentes pois não terão elementos para somar com as suas posições. Entretanto, mesmo sem tarefa, as threads precisam continuar invocando o comando `__syncthreads` pois a barreira requer que todas as threads do bloco passem por ela, ou seja, uma vez que uma thread do bloco invocou o comando `__syncthreads`, todas as demais threads do bloco também devem fazê-lo para que a barreira seja completada e todas as threads sejam desbloqueadas. Por isso foi necessário incluir os comandos `if` das linhas 8 e 12 e deixar as chamadas `__syncthreads` fora do corpo desses `if`.

Se analisarmos com mais profundidade o kernel do código 2.6, veremos que ele faz mais do que deixar a soma de todos os elementos do vetor na última posição. Como todas as threads estão processando ao mesmo tempo, ao final de $\log_2 N$ iterações, cada posição do vetor guardará a soma de todos os elementos antecessores. Essa foi uma opção que fizemos: deixar todas as threads fazendo a mesma tarefa e no final do processamento pegar apenas o resultado que nos interessa (nesse caso, o valor armazenado na última posição do vetor). Com essa decisão, foi possível simplificar

o código do kernel usando sempre o próprio índice da thread para determinar a posição do vetor que ela deverá alterar a cada iteração (assumindo que teremos uma thread para cada elemento do vetor de entrada). Sabemos que já na primeira iteração do algoritmo, apenas os resultados processados pelas threads de índices ímpares serão necessários. Entretanto, se disparássemos o kernel com apenas $N/2$ threads perderíamos a comodidade de usar diretamente os seus índices para acessar as posições correspondentes do vetor.

Outra simplificação que fizemos foi deixar todas as threads trabalhando até o final, mesmo sabendo que a cada iteração o número de threads que precisam continuar processando de fato cai pela metade. Novamente, optamos por manter o código do kernel mais simples ao invés de determinar a cada iteração quais threads precisam executar a soma. Veremos adiante, quando discutirmos o modelo de execução de CUDA, que essa decisão não trará impactos para o desempenho do programa.

Sincronização entre chamadas de kernels Uma questão importante relacionada ao mecanismo de sincronização de barreira de CUDA (`__syncthreads`), como já foi mencionado, é que ele é válido apenas para threads do mesmo bloco, isto é, não é possível sincronizar threads que estão em blocos distintos. Assim, o tamanho máximo do vetor de entrada para a solução implementada no código 2.6 fica limitado ao número máximo de threads de um bloco, que no nosso caso é de 1024.

A alternativa usual para sincronizar todas as threads de uma grade é dividir o kernel em mais de uma parte ou invocar o mesmo kernel mais de uma vez, ou seja, fazer a sincronização com a ajuda da CPU. Isso é possível porque a execução de um kernel só termina quando todas as threads de todos os blocos completam as suas execuções. Além disso, quando enfileiramos dois ou mais kernels para execução na GPU (em um mesmo *stream*), eles são executados em sequência, um após o outro. Os dados armazenados na memória global da GPU (alocados via `cudaMalloc`) são mantidos entre as chamadas de kernels. Isso nos permite invocar kernels distintos, mas continuar operando sobre o mesmo conjunto de dados.

O código 2.7 estende a solução anterior fazendo uso de dois níveis de sincronização: uma no nível de blocos e outra entre chamadas de kernels. O mesmo kernel é invocado duas vezes. Na primeira vez (linha 42), cria-se uma thread para cada elemento do vetor e cada bloco calcula a soma dos valores dentro da sua subparte do vetor total. O valor final calculado em cada bloco é armazenado em outro vetor, de tamanho igual ao número de blocos inicial. Nesse ponto, o kernel termina garantindo que todos os blocos concluíram a primeira rodada do processamento, isto é, somaram todos os elementos do vetor na subparte do bloco e preencheram a posição correspondente a cada bloco no vetor de saída (linhas 18 e 19).

Na segunda vez que o kernel é invocado (linha 50), o vetor de entrada passa a ser o vetor de saída da execução anterior e o mesmo processo é repetido. Cada bloco é responsável por calcular uma subparte do novo vetor e deixar o resultado da última thread no vetor auxiliar, na posição correspondente ao índice do bloco. Se após as duas execuções do kernel ainda tivermos elementos para somar, o processamento final será realizado na CPU.

```
1  //kernel para execução paralela na GPU
2  __global__ void somaElementosVetorPar(float *a, float *b) {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     int n = blockDim.x;
5     int salto;
```

```
6       for(salto=1; salto<n; salto*=2) {
7          float aux;
8          if(threadIdx.x >= salto) {
9             aux = a[i-salto];
10         }
11         __syncthreads();
12         if(threadIdx.x >= salto) {
13            a[i] = aux + a[i];
14         }
15         __syncthreads();
16      }
17      //última thread do bloco, salva seu valor no outo vetor
18      if (threadIdx.x == (n-1)) {
19         b[blockIdx.x] = a[i];
20      }
21   }
22   int main() {
23      float *h_s, *h_a, *d_a, *h_b, *d_b;
24      int n, n_threads;
25      int n_blocos1, n_blocos2;
26      double soma=0;
27      //n e n_threads (threads por bloco) são lidos da entrada do
           usuário

29      //número de blocos (primeira rodada)
30      n_blocos1 = (n + n_threads-1) / n_threads;

32      //faz as alocações de memória na CPU e inicializa o vetor de
           entrada
33      ...
34      //aloca espaço para os vetores na GPU
35      CUDA_SAFE_CALL(cudaMalloc((void**)&d_a, n*sizeof(float);));
36      CUDA_SAFE_CALL(cudaMalloc((void**) &d_b, n_blocos1*sizeof(
           float)));

38      //copia o vetor de entrada da CPU para a GPU (host para
           device)
39      CUDA_SAFE_CALL(cudaMemcpy(d_a, h_a, n_bytes,
           cudaMemcpyHostToDevice));

41      //dispara o kernel pela primeira vez
42      somaElementosVetorPar<<<n_blocos1, n_threads>>>(d_a, d_b);
43      CUDA_SAFE_CALL(cudaGetLastError());

45      //número de blocos (segunda rodada)
46      n_blocos2 = n_blocos1 / n_threads;

48      if (n_blocos2 > 0) {
49         //dispara o kernel pela segunda vez
50         somaElementosVetorPar<<<n_blocos2,n_threads>>>(d_b, d_a);
51         CUDA_SAFE_CALL(cudaGetLastError());

53         //copia resultado da GPU para a CPU (device para host)
54         CUDA_SAFE_CALL(cudaMemcpy(h_b, d_a, n_blocos2 * sizeof(
              float), cudaMemcpyDeviceToHost))
```

```
56         //termina os cálculos na CPU
57         for (int i = 0; i < n_blocos2; i++) {
58            soma += h_b[i];
59         }
60      } else {
61         //copia resultado da GPU para a CPU (device para host)
62         CUDA_SAFE_CALL(cudaMemcpy(h_b, d_b, n_blocos1 * sizeof(
               float), cudaMemcpyDeviceToHost))

64         //termina os cálculos na CPU
65         for (int i = 0; i < n_blocos1; i++) {
66            soma += h_b[i];
67         }
68      }
69      ...
```

Código 2.7: Soma os elementos de um vetor em CUDA. Versão com vários blocos de threads.

Por exemplo, se tivermos como entrada um vetor com 2^{20} elementos e fixarmos o tamanho dos blocos em 1024 (2^{10}) threads, na primeira rodada teremos 1024 (2^{10}) blocos e na segunda rodada apenas um bloco que devolverá o valor final da soma. Agora, se tivermos como entrada um vetor com 2^{25} elementos e mantivermos o tamanho dos blocos em 1024 (2^{10}) threads, na primeira rodada teremos 32768 (2^{15}) blocos e na segunda rodada 32 (2^{5}) blocos. Os 32 valores devolvidos no final serão somados na CPU. Por outro lado, se o vetor de entrada tiver 2^{15} elementos, teremos 32 (2^{5}) blocos de 1024 threads. Nesse caso, o kernel será invocado apenas uma vez e o restante do processamento será concluído na CPU.

2.3.3 Modelo de programação de CUDA

O modelo de programação de CUDA se apresenta como uma camada de abstração de nível mais alto, acima da arquitetura da GPU, que nos permite projetar e escrever algoritmos paralelos independente da maneira como eles serão processados. Isso quer dizer que o programador não precisa mapear os fluxos de execução lógica do algoritmo para as unidades de execução físicas e nem gerenciar a operação dessas unidades. Como veremos adiante (na seção 2.3.4), o modelo de execução de CUDA se encarrega dessas operações.

Podemos assumir então que existe um espaço discreto de computação onde uma grande quantidade de threads estão organizadas. Esse espaço de computação é caracterizado como uma **grade**, composta de **blocos de threads**. Para modelar um problema nesse espaço de computação é possível particioná-lo primeiro em subproblemas maiores que podem ser resolvidos independentemente e de forma paralela por blocos de threads, e depois dividir cada subproblema em partes menores que podem ser resolvidas cooperativamente em paralelo por todas as threads dentro de um bloco. Comparando com o método proposto por Foster [8] (seção 1.4), essa estratégia engloba os três primeiros passos — particionamento, comunicação e aglomeração — mas em uma ordem invertida.

Como vimos, a grade e os blocos são estruturas discretas de k dimensões ($k = 1, 2, 3$). Cada thread é unicamente identificada dentro do espaço de computação por meio das suas coordenadas dentro do bloco e das coordenadas do bloco dentro da grade. Essa organização do espaço de computação simplifica a tarefa de mapear

as threads para processarem diferentes localizações (partes menores) do problema a ser tratado. Por exemplo, se o domínio de dados do problema é unidimensional, as threads dos blocos podem ser organizadas como um vetor de tamanho igual à quantidade de dados de entrada, e a tarefa básica de processar cada elemento da entrada pode ser mapeada para a thread de mesmo índice global do elemento (como fizemos no código 2.3).

Se o domínio de dados é bidimensional, as threads podem ser organizadas em matrizes, de modo que cada thread poderá usar seus dois índices (i, j) para processar o elemento equivalente do domínio de entrada. O mesmo pode ser feito no caso de domínios de dados tridimensionais.

A Figura 2.3 ilustra a organização de um espaço de computação bidimensional. As threads dentro de cada bloco são organizadas em linhas e colunas e indexadas por um par de coordenadas (i, j). Os blocos, por sua vez, também são organizados em linhas e colunas formando uma grade bidimensional.

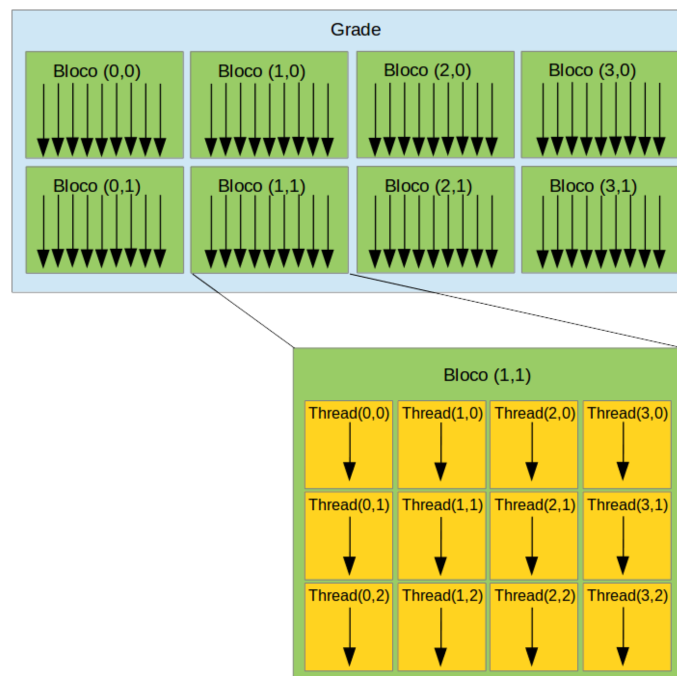


Figura 2.3: Espaço discreto de computação: uma grade bidimensional com blocos de threads bidimensionais.

Para acessar as coordenadas da thread dentro do kernel, usamos os campos `x` e `y` da variável `threadIdx`. Assim, uma thread é unicamente identificada dentro de um bloco usando o par de coordenadas (`threadIdx.x`, `threadIdx.y`). A dimensão do bloco também passa a ser definida com um par de valores, usando os campos `x` e `y` da variável `blockDim`. Os blocos são identificados unicamente dentro da grade por meio do par de coordenadas (`blockIdx.x`, `blockIdx.y`). Para identificar unicamente uma thread dentro da grade, precisamos mapear os índices da thread dentro do bloco para a posição correspondente na matriz da grade, fazendo:

```
i = (blockIdx.x * blockDim.x) + threadIdx.x
j = (blockIdx.y * blockDim.y) + threadIdx.y
```

Exemplo de mapeamento de um problema bidimensional Para mostrar o uso de um espaço de computação bidimensional em CUDA, vamor tomar o problema de multiplicação de matrizes como exemplo. Na seção 1.2 discutimos uma estratégia de paralelização desse problema, atribuindo para cada fluxo de execução a tarefa de calcular um elemento da matriz de saída. No código 1.6 mostramos o algoritmo proposto.

Para implementar esse algoritmo em CUDA, podemos seguir a mesma estratégia de dividir a tarefa de calcular a matriz de saída em subtarefas que consistem em calcular um único elemento dessa matriz. Na Figura 2.4 ilustramos novamente essa ideia. Para calcular o elemento (2,3) da matriz de saída `C`, precisaremos acessar a linha 2 da matriz de entrada `A` e a coluna 3 da matriz de entrada `B`.

Figura 2.4: Multiplicação de matrizes.

Para facilitar o mapeamento das threads para os elementos que cada uma delas deverá processar, vamos definir o espaço de computação como uma matriz com a mesma dimensão da matriz de saída `C`. O código 2.8 apresenta a primeira versão do kernel para lançamento de um único bloco com dimensão igual à dimensão da matriz de saída. Por simplicidade, assumimos que as matrizes de entrada são quadradas com dimensão N. Usamos a notação convencional de matrizes para definir os parâmetros de entrada do kernel e, dentro da função, usamos os índices da thread (coordenadas `x` e `y`) para definir o elemento que a thread deve processar (linhas 4 e 5). Como lançaremos apenas um bloco, os índices das threads podem ser mapeados diretamente para os índices dos elementos. Em seguida, computamos o valor do elemento lendo a linha e a coluna correspondente nas matrizes A e B (linhas 9 e 10).

```
//kernel para execução paralela na GPU
__global__ void multMatriz(float a[N][N], float b[N][N], float
    c[N][N]) {
   //coordenadas da thread
   int i = threadIdx.x;
   int j = threadIdx.y;

   //calcula o elemento C(i,j)
```

```
8      float valor = 0;
9      for (int k = 0; k < N; k++) {
10        valor += A[i][k] * B[k][j];
11     }
12     //escreve o valor calculado na matriz de saida
13     C[i][j] = valor;
14  }
15  int main() {
16     ...
17     //invoca o kernel com um bloco de n*n threads
18     dim3 threadsBloco(N, N);
19     multMatriz<<<1, threadsBloco>>>(A, B, C);
20     ...
21  }
```

Código 2.8: Multiplicação de matrizes em CUDA. Versão com um bloco de threads.

Para configurar as dimensões do bloco bidimensional, usamos o tipo `dim3`. Criamos uma variável chamada `threadsBloco` e inicializamos seus campos com as dimensões da matriz (linha 18). Na linha 19, disparamos o kernel usando essa variável para definir a dimensão dos blocos. O tipo `dim3` é um vetor de inteiros com três valores (`x, y, z`) que é usado para especificar dimensões. Quando um componente do vetor não é especificado, seu valor é inicializado com 1.

Usando o código 2.8, ficamos limitados a processar matrizes cujo número de elementos seja no máximo igual a 1024 (limite máximo de threads por bloco). Considerando matrizes quadradas, a maior dimensão permitida seria 32×32. Para resolver essa limitação, vamos estender esse código mapeando a matriz de saída em vários blocos de threads. O código 2.9 apresenta essa nova versão. Os valores dos índices `i` e `j` de cada thread são agora calculados levando-se em conta o índice e a dimensão de cada bloco (linhas 4 e 5). Dessa forma, cada thread endereçará um elemento único da matriz.

```
1   //kernel para execução paralela na GPU
2   __global__ void multMatriz(float a[N][N], float b[N][N], float
        c[N][N]) {
3      //coordenadas globais da thread
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;

7      //calcula o elemento C(i,j)
8      float valor = 0;
9      for (int k = 0; k < N; k++) {
10        valor += A[i][k] * B[k][j];
11     }
12     //escreve o valor calculado na matriz de saida
13     C[i][j] = valor;
14  }
15  int main() {
16     ...
17     //invoca o kernel com blocos de tamanhos fixos
18     dim3 threadsBloco(TAM_BLOCO, TAM_BLOCO);
19     dim3 blocosGrade(N/threadsBloco.x, N/threadsBloco.y);
20     multMatriz<<<blocosGrade, threadsBloco>>>(A, B, C);
21     ...
```

```
22 }
```

Código 2.9: Multiplicação de matrizes em CUDA. Versão com vários blocos de threads.

Para completar o mapeamento do espaço de dados para o espaço de computação, fixamos as dimensões dos blocos (linha 18) e calculamos a dimensão da grade com o número de blocos necessário para se ter uma thread por elemento da matriz (linha 19). Para simplificar o código, assumimos que a dimensão N das matrizes é sempre múltipla da dimensão dos blocos (`TAM_BLOCO`). Por exemplo, considerando as matrizes da Figura 2.4, a constante `TAM_BLOCO` poderia assumir os valores 2 ou 3.

2.3.4 Modelo de execução de CUDA

Na arquitetura das GPUs NVIDIA, as unidades de processamento mais elementares — denominadas *cores* — são agrupadas em unidades básicas de processamento paralelo que recebem o nome de **Streaming Multiprocessors** (**SMs**). Assim, podemos dizer que um SM é a menor unidade de processamento paralelo da GPU, sendo constituído por várias unidades elementares de processamento (ou *cores*). Por exemplo, a placa de vídeo GeForce GTX 770 (da NVIDIA) possui 8 SMs, com 192 *cores* em cada um deles, totalizando 1536 unidades de processamento elementares.

Todas as threads de um bloco são alocadas para execução no mesmo multiprocessador (SM) e compartilham os recursos de memória disponíveis nesse multiprocessador. Essa é a razão para o limite máximo do número de threads por bloco.

Uma questão bastante importante do modelo de execução de CUDA é que a execução de um bloco de threads é completamente independente da execução dos demais blocos. Assim, os blocos podem ser designados para execução em qualquer multiprocessador da GPU, em qualquer ordem, de forma sequencial ou paralela. Além disso, quando um bloco de threads é mapeado para execução em um SM, ele executa até terminar.

Dependendo do tamanho do bloco e da capacidade da GPU, mais de um bloco de threads pode ser alocado para executar concorrentemente no mesmo SM. Por exemplo, na GPU GeForce GTX 770 o número máximo de threads por SM é limitado em 2048. Assim, se os blocos tiverem tamanho 512, no máximo 4 blocos poderão ser mapeados para o mesmo SM ao mesmo tempo. À medida que os blocos terminam suas execuções é que novos blocos de threads podem ser alocados para os SMs disponíveis.

Warps Para gerenciar a execução de centenas de threads concorrentemente, os SMs adotam o modelo de arquitetura SIMT (*Single-Instruction, Multiple-Thread*). Diferente das CPUs, as instruções são executadas em ordem e não há predição de desvios e execução especulativa (técnica de antecipar a execução de instruções que estão dentro de fluxos de desvio condicional).

Em cada SM, as threads são criadas, gerenciadas, escalonadas e executadas em grupos de 32 threads chamados **warps**. Todas as threads de um mesmo **warp** começam a executar juntas a partir de um mesmo endereço de programa, mas cada thread tem seu próprio contador de endereço de instrução e registrador de estado, o que permite a execução de desvios condicionais.

Quando o SM recebe um ou mais blocos de threads para executar, ele agrupa as threads em warps seguindo a ordem consecutiva dos identificadores das threads. Todas as threads de um warp devem pertencer ao mesmo bloco. Assim, o primeiro

warp de cada bloco será constituído pelas threads de índice 0 a 31, o segundo warp será constituído pelas threads de índice 32 a 63, e assim sucessivamente. Se o tamanho do bloco não for divisível por 32, o último warp de cada bloco ficará com um número de threads ativas menor que 32.

A execução dos warps é gerenciada por **escalonadores de warps**. O número de escalonadores de warps por SM pode variar de acordo com o modelo da GPU. Um warp executa uma instrução de máquina por vez. Sendo assim o ideal — em termos de uso eficiente do hardware disponível — é que todas as 32 threads do warp sigam o mesmo fluxo de instruções. Quando as threads do warp **divergem** a partir de um desvio de fluxo condicional, a execução do warp é serializada: as threads do mesmo fluxo seguem a execução enquanto as demais ficam desabilitadas. A Figura 2.5 ilustra a ocorrência de divergência entre threads do mesmo warp. A condição avaliada (`(threadIdx.x % 2) == 0`) avaliará verdade (`true`) para as threads de índice par, e falso (`false`) para as threads de índice ímpar.

Quando todos os fluxos terminam, as threads voltam a executar a mesma instrução concorrentemente. A divergência (e serialização) de fluxo ocorre apenas dentro de um warp, ou seja, warps distintos executam concorrentemente. Se um bloco não possui número de threads múltiplo de 32, as últimas threads do último warp permanecerão desabilitadas durante toda a execução do bloco.

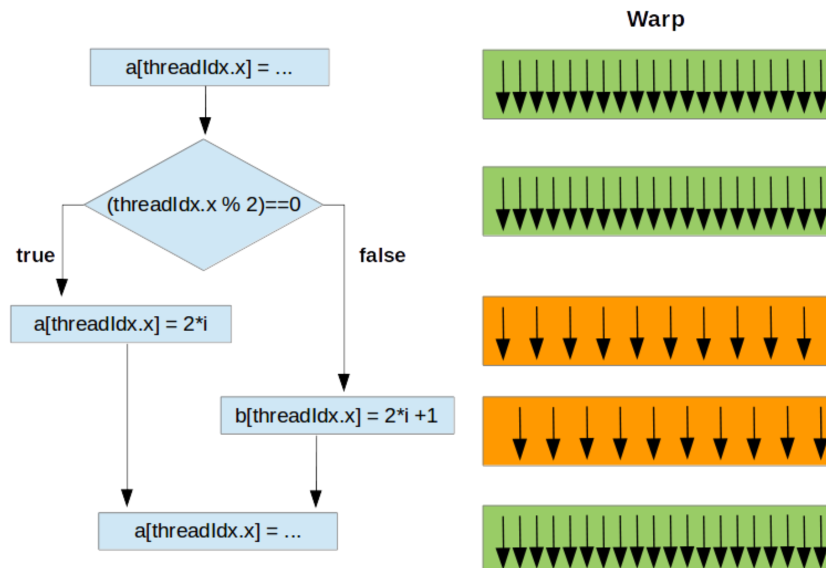


Figura 2.5: O warp executa os fluxos distintos de forma serial.

Cada warp processado por um SM possui um contexto de execução próprio mantido pelo hardware durante todo seu ciclo de vida. Por isso, a troca de contexto entre warps não tem custo e a cada instrução o escalonador de warps seleciona um warp cujas threads estejam prontas para executar a próxima instrução.

Para escrever programas corretos, não é necessário que o programador tenha consciência do modelo SIMT (escalonamento de warps). O seu modo de operação é completamente transparente durante a execução dos programas. Entretanto, para questões de eficiência, projetar algoritmos que evitem a divergência entre threads de um mesmo warp é sempre recomendável. Essa situação é análoga ao papel das memórias cache na programação para CPUs. Em termos de corretude do código, o

funcionamento e a operação dos vários níveis de memória cache podem ser completamente ignorados pelo programador. Entretanto, eles devem ser considerados no projeto dos algoritmos quando se deseja otimizar o desempenho da aplicação [21].

Instruções que alteram uma variável compartilhada Na seção 1.3, quando apresentamos uma implementação paralela do problema de somar os elementos de um vetor, esbarramos com uma situação onde todos os fluxos de execução escreviam/alteravam uma mesma variável global (compartilhada). Para garantir a corretude do código, destacamos a necessidade de usar algum mecanismo de exclusão mútua.

Na GPU, quando o warp executa uma instrução de escrita em uma posição de memória compartilhada pelas threads, e mais de uma thread do warp executa essa instrução, o número de escritas executadas de forma serial para essa localização de memória pode variar dependendo do modelo da GPU. Além disso, qual das threads executará a última escrita é indefinido.

Se a instrução for **atômica** (isto é, o programador explicitamente define que ela deve ser executada por uma thread de cada vez), seja ela de leitura ou de escrita, cada leitura/escrita para essa localização será realizada e elas serão executadas de forma serial, mas a ordem na qual elas ocorrem é indefinida.

2.3.5 Hierarquia de memória em CUDA

As threads CUDA acessam diferentes espaços de memória. Cada thread tem um espaço de **memória privada** para armazenar as variáveis automáticas (aquelas criadas dentro da função kernel). Dentro de um bloco, as threads dispõem de um espaço de **memória compartilhada** exclusivo de cada bloco e visível para todas as threads do mesmo bloco. A **memória global** pode ser acessada por todas as threads de uma grade e também pela CPU. Trata-se da memória de acesso mais lento e de maior capacidade de armazenamento.

O espaço de memória privada é alocado em registradores ou na memória global da GPU. Os registradores são a memória de acesso mais rápido, por isso seu uso é prioritário. Entretanto há situações em que o espaço de memória privada de cada thread precisa ser completado usando a memória global. Isso ocorre, por exemplo, quando as estruturas de dados são muito grandes e por isso consumiriam muito espaço nos registradores ou quando o kernel já está usando todos os registradores disponíveis para cada thread.

Há ainda dois espaços de memória, com acesso apenas para leitura, visíveis para todas as threads de uma grade: **memória constante** e **memória de textura**. O uso dessas memórias pode oferecer taxas de acesso bastante elevadas, dependendo do padrão de acesso das threads à memória. O espaço de memória constante reside na memória da GPU mas seu acesso é feito via uma área de memória cache constante. Uma requisição é dividida em várias requisições separadas, uma para cada endereço de memória contido na requisição inicial, reduzindo o tempo de acesso total. As requisições são servidas na taxa de acesso à memória constante, quando há acerto (o endereço solicitado já estava carregado na cache), ou na taxa de acesso à memória da GPU, caso contrário. O espaço de memória de textura reside na memória da GPU e é copiado para a cache de textura. Assim, o custo de uma leitura nessas memórias é o custo de uma leitura na memória da GPU apenas se houver falha de cache. De outro modo, o custo será de um acesso à memória cache de textura. A cache de textura é otimizada para estruturas de dados 2D. Mais detalhes sobre as

características e formas de uso desses espaços de memória podem ser encontrados no manual de programação de CUDA [21].

Os espaços de memória global, constante e de textura são persistentes entre chamadas de kernels da mesma aplicação. Já o conteúdo da memória compartilhada e da memória privada de cada thread é reiniciado a cada invocação de um novo kernel.

Memória compartilhada A memória compartilhada é específica de cada SM e reside no chip da GPU. Por isso o acesso a essa memória é mais rápido do que o acesso à memória global. A memória compartilhada pode ser vista como uma memória cache gerenciada pelo programador, uma vez que seu acesso deve ser feito de forma explícita dentro do código da aplicação.

Como não é possível transferir dados diretamente da memória da CPU para a memória compartilhada dos blocos, o uso desse espaço de memória requer sempre uma etapa adicional de leitura/escrita de dados de/para a memória global. Desse modo, a seguinte sequência de passos precisa ser implementada quando a memória compartilhada é usada:

1. carregar o dado da memória global para a memória compartilhada;
2. sincronizar as threads do bloco para garantir que toda a carga foi concluída;
3. processar o dado na memória compartilhada;
4. sincronizar novamente as threads, se necessário, para garantir que a memória compartilhada foi completamente atualizada;
5. escrever os resultados na memória global.

Assim como ocorre com as memórias cache, o uso da memória compartilhada é especialmente vantajoso quando o dado que foi copiado da memória global é usado mais de uma vez antes de ser devolvido para a memória global.

Para alocar uma variável no espaço de memória compartilhada, o programador deve usar explicitamente o qualificador `__shared__` na definição da variável. O qualificador `__shared__` declara variáveis com as seguintes propriedades: (i) residem na memória compartilhada de um bloco de threads; (ii) existem enquanto o bloco estiver ativo; (iii) são acessíveis apenas por threads do mesmo bloco.

Para mostrar um exemplo de uso da memória compartilhada, vamos voltar ao problema de **multiplicação de matrizes**. No código 2.9, atribuímos para cada thread a tarefa de calcular um elemento da matriz de saída. Para isso, cada thread precisará ler todos os elementos da linha e coluna correspondentes ao elemento nas matrizes A e B. O que ocorre nesse caso é que cada linha e coluna dessas matrizes serão lidas por mais de uma thread. A Figura 2.6 ilustra essa situação.

Se observarmos, por exemplo, a submatriz que está destacada dentro da matriz de saída C (com os elementos (2,2), (2,3), (3,2) e (3,3)), podemos notar que para calcular seus 4 elementos vamos usar as linhas 2 e 3 da matriz A e as colunas 2 e 3 da matriz B duas vezes. Nesse caso, vale a pena copiar o conteúdo dessas linhas e colunas da memória global para a memória compartilhada e reutilizar seus valores.

Para fazer isso, primeiro precisamos declarar o espaço de memória compartilhada dentro do kernel. Depois copiar os elementos da memória global para a memória compartilhada, e finalmente usar os seus valores. Como o espaço de memória compartilhada disponível para cada bloco é relativamente pequeno, nem sempre é possível copiar todos os valores que serão compartilhados de uma vez. Então, a

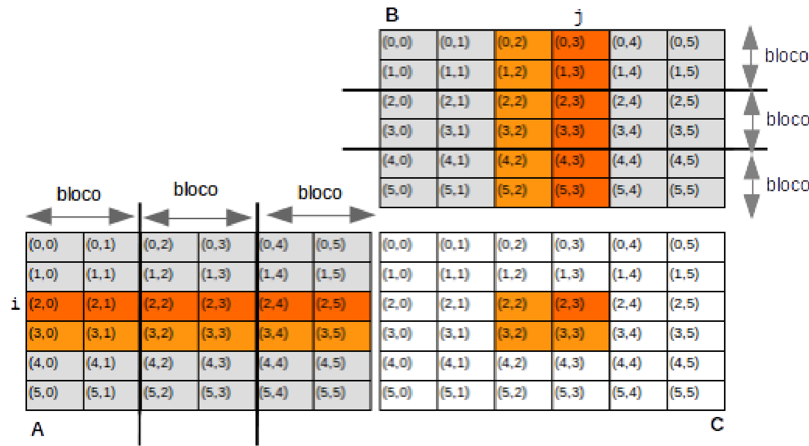


Figura 2.6: Multiplicação de matrizes com memória compartilhada.

alternativa mais comum é copiar os dados em blocos, utilizá-los e em seguida trazer um novo bloco até que todos os elementos tenham sido processados.

Para o problema de multiplicação de matrizes, vamos alocar na memória compartilhada duas submatrizes de dimensão igual à dimensão do bloco de threads: uma para copiar os elementos da matriz A e outra para copiar os elementos da matriz B. A cada passo, cada thread do bloco copia para a memória compartilhada um elemento da matriz A e um elemento da matriz B. Em seguida, todas as threads usam os elementos copiados para computar uma parte do seu elemento de saída. Concluída a iteração, as threads prosseguem copiando as submatrizes subsequentes de A e de B.

O código 2.10 implementa essa estratégia. Cada bloco de threads é responsável por computar uma submatriz de C e cada thread do bloco é responsável por computar um elemento dessa submatriz.

```
//kernel para execução paralela na GPU
__global__ void multMatriz(float a[N][N], float b[N][N], float
    c[N][N]) {
  //coordenadas globais da thread
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  //coordenadas locais da thread
  int i_bloco = threadIdx.x;
  int j_bloco = threadIdx.y;

  //aloca memória compartilhada para armazenar a submatriz de
      A
  __shared__ float Asub[TAM_BLOCO][TAM_BLOCO];
  //aloca memória compartilhada para armazenar a submatriz de
      B
  __shared__ float Bsub[TAM_BLOCO][TAM_BLOCO];

  //calcula o elemento C(i,j)
  float valor = 0;
```

```
18      for(int passo = 0; passo < N; passo += TAM_BLOCO) {
19         //carrega as submatrizes da memória global para a memória
               compartilhada, cada thread carrega um elemento da
              matriz
20         Asub[i_bloco][j_bloco] = A[i][passo + j_bloco];
21         Bsub[i_bloco][j_bloco] = B[passo + i_bloco][j];

23         //sincroniza para todas as threads terminarem a cópia
              antes de começarem a computação
24         __syncthreads();

26         //multiplica as submatrizes, cada thread computa um
              elemento
27         for (int k = 0; k < TAM_BLOCO; k++) {
28            valor += Asub[i_bloco][k] * Bsub[k][j_bloco];
29         }

31         //sincroniza para todas as threads terminarem a sua
              computação antes de carregar a nova submatriz
32         __syncthreads();
33      }
34      //escreve o valor calculado na matriz de saida
35      C[i][j] = valor;
36   }
37   int main() {
38      ...
39      //invoca o kernel com blocos de tamanhos fixos
40      dim3 threadsBloco(TAM_BLOCO, TAM_BLOCO);
41      dim3 blocosGrade(N/threadsBloco.x, N/threadsBloco.y);
42      multMatriz<<<blocosGrade, threadsBloco>>>(A, B, C);
43      ...
44   }
```

Código 2.10: Multiplicação de matrizes em CUDA. Versão com vários blocos de threads e memória compartilhada.

Nas linhas 12 e 14, declaramos duas submatrizes no espaço de memória compartilhada para copiar os elementos das matrizes de entrada. O comando `for` da linha 18 determina quantas iterações serão necessárias para que as threads do bloco carreguem e computem todos os dados das matrizes de entrada (em blocos de tamanho `TAM_BLOCO`).

Dentro de cada iteração, as threads primeiro copiam os elementos das matrizes A e B para as submatrizes `Asub` e `Bsub` (linhas 20 e 21). Em seguida, as threads se sincronizam chamando `__syncthreads` (linha 24) para garantir que todas as threads terminaram a cópia antes de começarem a computação. Nas linhas 27 e 28, cada thread computa mais uma parte do seu elemento de saída, usando os dados que foram copiados anteriormente. Por fim, as threads se sincronizam novamente (linha 32) para garantir que todas terminaram a computação anterior antes de começarem a copiar as novas submatrizes.

Quando todas as iterações são concluídas, cada thread copia para a matriz C o valor computado (linha 35). Com essa implementação, aproveitamos o acesso mais rápido da memória compartilhada e reduzimos o acesso à memória global, uma vez que as matrizes de entrada serão lidas apenas `N/TAM_BLOCO` vezes.

Voltando para a Figura 2.6, vamos considerar como exemplo que cada bloco tenha dimensão 2×2 e que queremos calcular a submatriz que está destacada em

C. Na primeira iteração, as threads deverão carregar para a memória compartilhada os elementos das células (2,0), (2,1), (3,0) e (3,1) da matriz A, e os elementos (0,2), (0,3), (1,2) e (1,3) da matriz B. Na segunda iteração as threads deverão carregar para a memória compartilhada os elementos das células (2,2), (2,3), (3,2) e (3,3) da matriz A e da matriz B. Finalmente, na terceira iteração, as threads deverão carregar para a memória compartilhada os elementos das células (2,4), (2,5), (3,4) e (3,5) da matriz A, e os elementos (4,2), (4,3), (5,2) e (5,3) da matriz B. As barras verticais e horizontais na figura indicam os pontos de barreira a cada iteração do algoritmo.

Memória compartilhada com alocação dinâmica A declaração das variáveis compartilhadas no código 2.10 foi feita de forma estática, isto é, definimos dentro do kernel o tamanho das submatrizes `Asub` e `Bsub`. Há outra forma de declarar variáveis compartilhadas, onde o tamanho da estrutura de dados é definido no lançamento do kernel. Para isso, a declaração da variável compartilhada deve ser precedida pelo qualificador `extern`, como ilustrado a seguir [21]:

`extern __shared__ float vetor_compartilhado[];`

Por exemplo, se quisermos definir os vetores

```
1 short vetor0[128];
2 float vetor1[64];
3 int vetor2[256];
```

na memória compartilhada, então devemos declará-los da seguinte forma dentro da função kernel:

```
1 extern __shared__ float vetor_compartilhado[];
2 short* vetor0 = (short*) vetor_compartilhado;
3 float* vetor1 = (float*) &vetor0[128];
4 int* vetor2 = (int*) &vetor1[64];
```

Todas as variáveis declaradas dessa forma começam no mesmo endereço na memória, então a organização das variáveis dentro do vetor deve ser explicitamente gerenciada através de deslocamentos a partir do endereço inicial, como mostra o exemplo acima.

Importante destacar que os ponteiros precisam estar alinhados para o tamanho do tipo que eles apontam. O código abaixo ilustra uma situação de erro pois o vetor `vetor1` não está alinhado para 4 bytes (tamanho do tipo `float`).

```
1 extern __shared__ float vetor_compartilhado[];
2 short* vetor0 = (short*) vetor_compartilhado;
3 float* vetor1 = (float*) &vetor0[127];
```

Quando o kernel é lançado, deve-se acrescentar na sua lista de parâmetros de configuração a quantidade de bytes que deverão ser alocados para a memória compartilhada. Como vimos no início da seção 2.3.1, a configuração de execução do kernel é definida dentro de uma expressão na forma `<<< Dg , Db , Ns , S >>>` que precede a lista de argumentos da função. O terceiro parâmetro, `Ns`, é usado para definir a quantidade de bytes da memória compartilhada que será dinamicamente alocada para cada bloco. Essa memória será usada pelas variáveis declaradas como vetores e precedidas pelo qualificador `extern` dentro do kernel. A chamada para a função kernel retornará com falha se a soma de `Ns` com a quantidade de memória compartilhada requerida para alocação estática dentro do kernel for maior que a quantidade máxima de memória compartilhada disponível para cada SM.

No capítulo 3 mostraremos exemplos de uso da memória compartilhada com alocação dinâmica.

2.3.6 Estratégias para otimizar o desempenho das aplicações

A NVIDIA adota um modelo de versionamento para representar a capacidade de computação de suas placas de vídeo. O número de versão identifica as características oferecidas pelo hardware da GPU e pode ser usado pelas aplicações em tempo de execução para tomar decisões sobre quais características explorar. O número de versão possui duas partes denotadas por $X.Y$. O valor de X indica a arquitetura da GPU (5, arquitetura *Maxwell*; 3, arquitetura *Kepler*; 2, arquitetura *Fermi*; e 1, arquitetura *Tesla* (descontinuada a partir da versão 7.0 de CUDA). O valor de Y indica melhorias incrementais dentro da arquitetura.

Independente da capacidade de computação de uma GPU, algumas questões básicas devem ser levadas em conta quando projetamos e implementados um programa em CUDA para que o seu desempenho seja o máximo possível. Discutiremos nessa seção algumas dessas questões.

Maximizando a execução paralela na GPU Para maximizar a utilização da capacidade de processamento da GPU, o primeiro passo é estruturar o algoritmo paralelo de forma que ele exponha o máximo de paralelismo possível. O segundo passo consiste em mapear esse paralelismo de forma eficiente para os vários componentes do sistema, visando mantê-los ocupados o máximo de tempo.

Quando o paralelismo é quebrado em pontos que requerem a sincronização das threads para compartilhar dados entre elas, dois casos podem ocorrer: (i) as threads pertencem ao mesmo bloco, nesse caso deve-se usar `__syncthreads` e compartilhar os dados por meio da memória compartilhada do bloco dentro da mesma invocação do kernel; ou (ii) as threads pertencem a blocos distintos e nesse caso as threads devem compartilhar dados por meio da memória global usando duas (ou mais) invocações separadas do kernel (uma para escrever para a memória global e outra para ler da memória global). O segundo caso pode ter um custo computacional mais elevado, dada a necessidade de invocar mais de um kernel e fazer mais uso da memória global.

Como os SMs operam com o paralelismo no nível de warps, quanto maior o número de warps alocados/residentes em um SM ao mesmo tempo, melhor. Desse modo, aumenta-se a chance do escalonador de warps encontrar um warp pronto para executar. Se sempre houver um warp disponível para alocação, a latência de execução dos warps (intervalo de tempo em que um warp precisa aguardar por alguma condição para então poder executar a próxima instrução) pode ser completamente escondida.

Cada SM possui um conjunto de registradores de 32 bits que são divididos entre os warps e uma área de memória compartilhada que é particionada entre os blocos de threads. Desse modo, a quantidade total de memória compartilhada usada por um bloco de threads e o número de registradores usados pelo kernel limita o número de warps residentes em um SM. Por exemplo, para GPUs de capacidade 2.x, o número de registradores por SM é de 1024×32, então, se o kernel usar 32 registradores (o que significa que cada thread precisará de 32 registradores) e cada bloco de threads for definido com 512 threads, no máximo 2 blocos (ou seja, 32 warps) poderão ser alocados para cada SM ao mesmo tempo. Se o kernel usar um registrador a mais, apenas um bloco de threads (ou seja, 16 warps) poderá residir no SM [21].

Uma das razões para um warp ficar suspenso é quando os operandos de entrada da próxima instrução não estão disponíveis (ainda presisam ser escritos nos registradores pela instrução anterior ou estão na memória). Outra razão para um warp não estar pronto para executar é quando ele está esperando em algum ponto de sincronização. Um ponto de sincronização força o multiprocessador a suspender mais e mais warps até que todos os warps do mesmo bloco cheguem no mesmo ponto de sincronização. Nesse caso, ter vários blocos alocados/residentes no mesmo multiprocessador ajuda a minimizar o impacto da latência da sincronização, uma vez que warps de blocos distintos não compartilham a mesma sincronização.

CUDA oferece várias funções em sua API para auxiliar os programadores a escolher o melhor tamanho de bloco com base na quantidade de espaço de memória compartilhada e número de registradores requeridos pelo kernel [21].

A função `cudaOccupancyMaxActiveBlocksPerMultiprocessor`, por exemplo, retorna uma previsão de ocupação em termos do número de blocos de threads por multiprocessador, levando-se em conta o tamanho do bloco e os recursos de memória requeridos pelo kernel. Algumas métricas adicionais podem ser obtidas a partir dessa informação:

- *número de warps concorrentes por multiprocessador*: número de warps por bloco multiplicado pelo número de blocos de threads por multiprocessador;
- *percentual de ocupação de warps*: número de warps concorrentes por multiprocessador dividido pelo número máximo de warps por multiprocessador.

O ideal é que o percentual de ocupação de warps seja igual a 1, ou seja, o máximo permitido pela GPU.

Se não houver número de registradores ou espaço de memória compartilhada no SM para processar ao menos um bloco, o kernel não é executado. (A quantidade de registradores e memória compartilhada usada por um bloco é reportada pelo compilador usando a opção `--ptxas-options=-v`.) Por último, o número de threads por bloco deveria ser escolhido sempre que possível como um múltiplo de 32 (tamanho do warp) para evitar a subutilização de recursos.

Maximizando a taxa de transferência da memória A taxa de transferência no acesso à memória pelo kernel pode variar de uma ordem de grandeza dependendo do padrão de acesso para cada tipo de memória. Assim, organizar o acesso à memória da forma mais otimizada possível, de acordo com os padrões de uso recomendados, é um importante passo para maximizar o desempenho da aplicação. As recomendações mais gerais para maximizar a taxa de transferência da memória são:

1. minimizar as transferências de dados entre CPU e GPU;
2. minimizar as transferências de dados da memória global da GPU para as memórias dos multiprocessadores (memória compartilhada e caches).

Acesso coalescente à memória global A memória global reside na memória da GPU e é acessada por meio de transações de memória de 32, 64 ou 128 bytes. Assim, para otimizar o acesso à memória, o ideal é carregar segmentos de 32, 64 ou 128 bytes onde o primeiro endereço é múltiplo do tamanho do segmento, de forma que eles poderão ser lidos ou escritos em uma transação de memória [21].

Quando um warp executa uma instrução que acessa a memória global, ele **aglutina** (coalesce) os acessos de memória das threads dentro do warp em uma ou mais

dessas transações de memória, dependendo do tamanho da sequência de bytes requisitada por cada thread e da distribuição dos endereços de memória entre as threads. Por exemplo, se uma thread requisita um acesso de memória para ler uma variável de 4 bytes (inteiro ou float), será necessário realizar uma transação de memória de 32 bytes (8 vezes o tamanho da variável). Desse modo, 28 bytes serão transferidos mas não serão usados pela thread. Agora, se outras sete threads do mesmo *warp* solicitarem um acesso de memória para ler uma variável de 4 bytes, sendo que as oito variáveis estão em posições consecutivas da memória e o endereço da primeira variável é múltiplo do tamanho do segmento, todos os 32 bytes trazidos pela transação de memória de 32 bytes serão usados. Nesse caso, teremos uma ótima relação entre o número de instruções executadas para cada transferência de memória (todos os bytes transferidos foram utilizados).

Alinhamento dos dados na memória global As instruções de acesso à memória global suportam requisições de leitura ou escrita de palavras (sequência de bytes) de tamanho igual a 1, 2, 4, 8 ou 16. Qualquer acesso para dados na memória global gera uma instrução simples de acesso à memória se e somente se o tamanho do tipo de dado é de 1, 2, 4, 8 ou 16 bytes e o dado está alinhado (ou seja, seu endereço inicial é um múltiplo do seu tamanho). Se os requisitos de tamanho ou de alinhamento não são atendidos, o compilador gera várias instruções simples de acesso à memória com padrões de acesso intercalados que dificultam a aglutinação dessas instruções pelos warps.

Recomenda-se, portanto, o uso de tipos de dados que atendam aos requisitos de tamanho e alinhamento. O alinhamento é automaticamente satisfeito para os tipos predefinidos: `char, short, int, long, long long, float, double`.

Quando é necessário usar estruturas de dados, os requisitos de tamanho e alinhamento podem ser aplicados pelo compilador usando os especificadores de alinhamento, como ilustrado no trecho de código abaixo:

```
struct __align__(16) {
   float x;
   float y;
   float z;
};
```

Nesse caso, a estrutura de dados ocuparia 12 bytes (considerando que cada variável do tipo float ocupa 4 bytes) o que requereria mais de um acesso à memória desnecessariamente, uma vez que é possível acessar blocos de 16 bytes contínuos na memória em uma única requisição. Com o uso do especificador de alinhamento `__align__`, indicamos ao compilador que a estrutura de dados deve ser alinhada em 16 bytes. O custo associado a essa decisão é que a estrutura de dados ocupará um espaço de memória extra (nesse exemplo de 4 bytes). Entretanto, o ganho de desempenho obtido pode compensar esse consumo adicional de espaço de memória.

Memória local A memória local reside na memória da GPU, então os acessos à memória local sofrem a mesma latência e possuem a mesma taxa de transferência dos acessos à memória global. Em função disso, os requerimentos para acesso aglutinado à memória são os mesmos.

Divergência de instruções Os comandos `if, switch, do, for, while` — de controle de fluxo — podem impactar de forma significativa na taxa de execução

das instruções de um programa em razão da possibilidade de causarem divergências entre as threads de um mesmo warp (ou seja, fazer as threads seguirem fluxos de execução distintos). As divergências dentro de um warp fazem com que os diferentes fluxos precisem ser serializados, aumentando o número total de instruções executadas pelo warp.

A Figura 2.5 ilustrou essa situação: o código executado pelas threads passa por um desvio de fluxo que distingue threads com índices pares de threads com índices ímpares, implementado pelo comando `if ((threadIdx.x % 2) == 0)`. Dentro de cada warp, as threads com índices pares executarão o fluxo que avalia a condição lógica do comando `if` como verdadeira (`a[threadIdx.x] = 2*i`), enquanto as threads de índices ímpares ficarão ociosas. Em seguida, a situação se inverte, as threads com índices ímpares executarão o fluxo que avalia a condição lógica do comando `if` como falsa (`a[threadIdx.x] = 2*i+1`), enquanto as threads de índices pares ficarão ociosas. O efeito no desempenho da aplicação é que todas as threads consumirão o tempo requerido para executar os dois fluxos determinados pelo comando `if`, quando na verdade só executaram um dos fluxos.

Nos casos de kernels em que o fluxo de controle depende do ID da thread, a condição de controle deveria ser escrita de forma a minimizar o número de warps divergentes, isto é, garantir que as divergências somente ocorram no início de novos warps (a partir de threads múltiplas de 32). Isso é possível, uma vez que a distribuição dos warps ao longo dos blocos é determinística. No exemplo da Figura 2.5, corresponderia a avaliar a possibilidade da divergência não ser implementada entre threads pares e ímpares, mas entre blocos contínuos de 32 threads. Por exemplo, as primeiras 32 threads executariam o primeiro fluxo, as 32 threads seguintes o segundo fluxo, as próximas 32 threads o primeiro fluxo e assim sucessivamente, alternando os fluxos a cada bloco de 32 threads.

Operações de ponto flutuante As GPUs NVIDIA implementam o padrão IEEE-754 [15] para realizar as operações de ponto flutuante com precisão simples e dupla. Compreender como o hardware da GPU trata essas operações é de grande importância para conseguir implementar programas com bom desempenho e com a precisão requerida. Para uma discussão mais aprofundada sobre essa questão, sugerimos como ponto de partida o trabalho de Whitehead e Fit-Florea [39].

2.4 Programação de GPUs com OpenCL

OpenCL [1] [10] é um padrão aberto para programação paralela de sistemas heterogêneos. Um sistema heterogêneo é um sistema de computação que inclui uma CPU tradicional (*host*) e um ou mais dispositivos (*devices*), também chamados aceleradores ou processadores massivamente paralelos, com várias unidades aritméticas (por exemplo, FPGAs, GPUs, e outros).

Diferente de CUDA, OpenCL foi projetado para o desenvolvimento de aplicações paralelas portáveis, isto é, independentes de plataforma. OpenCL oferece uma linguagem *kernel* e duas APIs: *C Platform Layer API* (para buscar, selecionar e inicializar dispositivos) e *C Runtime API* (para construir e executar kernels em vários dispositivos).

O projeto do OpenCL foi iniciado pela empresa Apple, em colaboração com outras empresas fabricantes de hardware (AMD, IBM, Intel e NVIDIA). Seu desen-

[1] https://www.khronos.org/opencl

volvimento é gerenciado pelo grupo Khronos. A primeira especificação do OpenCL foi lançada em 2008. A versão atual é a 2.2, lançada em maio de 2016.

OpenCL é compatível com as CPUs `x86, ARM, PowerPC` e pode ser usada em GPUs da AMD, Apple e NVIDIA. Algumas características oferecidas pelo OpenCL são opcionais e não são suportadas por todos os tipos de dispositivos.

Assim como CUDA, um programa OpenCL consiste de duas partes: funções que executam em um ou mais dispositivos OpenCL (*kernel*) e o programa principal que gerencia a execução dos *kernels*. De forma geral, o modelo de programação de OpenCL é bastante similar ao modelo de programação de CUDA. Porém, como OpenCL é um padrão independente de plataforma, os programas OpenCL tendem a ser mais complexos uma vez que precisam tratar características distintas de diversas plataformas.

2.5 Exercícios

1. A distância euclidiana entre um par de pontos com coordenadas (x_i, y_i, z_i) em um espaço tridimensional é dada por: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$. Considere um conjunto de N pontos em um espaço euclidiano tridimensional. Dado um ponto qualquer nesse espaço, escreva um programa em CUDA para calcular a distância entre esse ponto e todos os pontos do conjunto dado. Implemente uma função em C para checar se o resultado calculado está correto. Experimente seu programa com diferentes valores de N.

2. Sejam $(a_1, a_2, \ldots, a_N)^T$ e $(b_1, b_2, \ldots, b_N)^T$ vetores em um espaço de dimensão N expressos em termos de um sistema ortogonal de coordenadas cartesianas. O produto interno desses dois vetores é um valor real dado pela equação: $a_1b_1 + a_2b_2 + \cdots + a_Nb_N$. Escreva um programa em CUDA para calcular o produto interno de dois vetores de dimensão N. Implemente uma versão sequencial para resolver o mesmo problema e avalie o desempenho do programa em CUDA comparando os resultados obtidos com a versão sequencial. Use as métricas de aceleração e vazão computacional para a sua análise. Para a tomada de tempo da versão paralela, inclua o tempo de transferência dos dados da memória da CPU para a GPU.

3. Como foi discutido brevemente na seção 2.3.2, o código 2.6 implementa uma solução para o problema de **soma de prefixos paralela** (dado um vetor de entrada, cada elemento do vetor de saída deve conter a soma de todos os seus antecessores). No entanto, o código 2.6 limita o tamanho do vetor de entrada para o tamanho máximo de um bloco de threads (1024 elementos). Proponha uma extensão desse código para executar a soma de prefixos para vetores com qualquer tamanho. Compare o desempenho obtido com uma versão sequencial do problema, usando a métrica de aceleração.

4. O kernel implementado no código 2.9 para multiplicação de matrizes com vários blocos de threads faz duas simplificações: assume que as matrizes de entrada são quadradas, de dimensão N, e que o valor de N é sempre múltiplo do tamanho dos blocos (`TAM_BLOCO`). Altere esse kernel para que ele processe matrizes de qualquer dimensão, não apenas matrizes quadradas, e trate os casos em que N não é múltiplo de `TAM_BLOCO`.

5. Aplique as mesmas extensões do exercício anterior no código 2.10 que multiplica duas matrizes usando a memória compartilhada dos blocos. Avalie

o desempenho desse programa experimentando com diferentes dimensões de matrizes de entrada e de tamanho dos blocos de threads. Analise o comportamento da métrica de aceleração mantendo o tamanho das matrizes de entrada fixo e alterando o tamanho dos blocos.

Capítulo 3

Solução de equações lineares usando GPUs

3.1 Introdução

Sistemas de equações lineares aparecem com muita frequência na solução de problemas das engenharias e das ciências.

Em geral, um sistema de n equações lineares com n incógnitas pode ser escrito na forma

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + & \cdots & + a_{1n}x_n & = & d_1 \\ a_{21}x_1 + a_{22}x_2 + & \cdots & + a_{2n}x_n & = & d_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + & \cdots & + a_{nn}x_n & = & d_n \end{cases} \tag{3.1.1}$$

Usando a notação matricial, temos a equação vetorial

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{d} \tag{3.1.2}$$

onde a matriz dos coeficientes $\mathbf{A}$, o vetor das incógnitas $\mathbf{x}$ e o vetor do lado direito $\mathbf{d}$ são dados por

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2,n-1} & a_{2n} \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{e} \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

Assim, para se obter a solução de (3.1.2) devemos calcular $\mathbf{A}^{-1} \cdot \mathbf{d}$, em que $\mathbf{A}^{-1}$ representa a matriz inversa de $\mathbf{A}$. Na prática, para resolver esse sistema são usados métodos em que não é preciso calcular explicitamente a matriz inversa.

Um dos métodos mais usados para a resolução de sistemas lineares é o método de eliminação gaussiana com retrosubstituição [6, 11]. Uma das situações mais simples em que o método de eliminação gaussiana com retrosubstituição mostra sua grande utilidade é no caso de sistemas de equações tridiagonais. Nesse caso, o método pode ser organizado de forma eficiente e o método resultante recebe a denominação de algoritmo de Thomas [6].

O método de fatoração LU é uma forma de obter uma decomposição da matriz de um sistema de equações lineares aplicando o método de eliminação gaussiana

[6, 11]. Esse método é muito vantajoso nas aplicações onde é preciso resolver vários sistemas de equações que compartilham a mesma matriz dos coeficientes.

Neste capítulo, discutimos algumas estratégias de implementação paralela de diferentes métodos de solução de sistemas de equações tridiagonais e da fatoração LU, usando GPU.

3.2 Sistemas de equações tridiagonais

Sistemas de equações tridiagonais aparecem em muitas aplicações, como por exemplo no cálculo de funções *spline* e na solução de problemas de valor de contorno para equações diferenciais [6].

Quando os coeficientes do sistema (3.1.1) satisfazem as condições

$$a_{ij} = 0 \quad \text{se} \quad |i - j| > 1,$$

dizemos que o sistema é tridiagonal. Assim sendo, temos que na matriz dos coeficientes todos os elementos que se encontram fora da diagonal principal ou das duas diagonais exatamente acima e abaixo da diagonal principal, são nulos.

Nesse caso, (3.1.1) pode ser re-escrito na forma

$$\left\{ \begin{array}{lcl} b_1 x_1 + c_1 x_2 & = & d_1, \\ \quad \ddots & & \\ \quad a_i x_{i-1} + b_i x_i + c_i x_{i+1} & = & d_i, \\ \qquad \qquad \ddots & & \\ \qquad \qquad a_n x_{n-1} + b_n x_n & = & d_n. \end{array} \right. \tag{3.2.3}$$

Um dos métodos mais simples para a solução do sistema (3.2.3) é o *algoritmo de Thomas*. Este algoritmo consiste basicamente na aplicação do método de eliminação de Gauss (sem pivoteamento) com a respectiva substituição reversa.

Após o processo de eliminação de Gauss é obtido um sistema triangular da forma

$$\left\{ \begin{array}{lcl} x_1 + c'_1 x_2 & = & d'_1, \\ \quad \ddots & & \\ \quad x_i + c'_i x_{i+1} & = & d'_i, \\ \qquad \qquad \ddots & & \\ \qquad \qquad x_n & = & d'_n. \end{array} \right. \tag{3.2.4}$$

Para isto primeiro transformamos a primeira equação de (3.2.3) de acordo com a transformação

$$E_1/b_1 \rightarrow E'_1$$

e assim obtemos que

$$c'_1 = \frac{c_1}{b_1}, \quad d'_1 = \frac{d_1}{b_1}.$$

A partir daí as outras equações são tratadas de forma recursiva. Se a equação $i-1$ já foi reduzida à forma

$$E'_{i-1}: \quad x_{i-1} + c'_{i-1} x_i = d'_{i-1}$$

a transformação da i-ésima equação

$$E_i: \quad a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

será feita eliminando a variável x_{i-1}, ou seja, aplicando a transformação

$$E_i - a_i \cdot E'_{i-1} \to E_i$$

e seguidamente dividindo a equação resultante pelo coeficiente que acompanha a variável x_i. Assim obtemos a nova equação

$$E'_i : x_i + c'_i x_{i+1} = d'_i$$

onde

$$c'_i = \frac{c_i}{b_i - a_i c'_{i-1}}, \quad d'_i = \frac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}}.$$

Na segunda fase o sistema (3.2.4) é resolvido aplicando substituição reversa, ou seja, calculando primeiro

$$x_n = d'_n,$$

e depois obtendo recursivamente $x_{n-1}, x_{n-2}, \ldots, x_1$, fazendo

$$x_i = d'_i - c'_i x_{i+1}, \qquad i = n-1, \ldots, 1.$$

Observamos que o método será bem sucedido se e somente se $b_1 \neq 0$ e $b_i \neq a_i c'_{i-1}$ para $i = 2, \ldots, n$. Isso acontece por exemplo se a matriz for estritamente diagonal dominante, ou se for simétrica e definida positiva [6].

O algoritmo resultante é muito simples mas fortemente sequencial. Isso é consequência de que no cálculo de c'_i e d'_i usamos diretamente os coeficientes c'_{i-1} e d'_{i-1} que são determinados apenas no passo anterior. A mesma observação pode ser realizada em relação ao cálculo de x_i e sua dependência de x_{i+1}.

3.2.1 Método de redução cíclica

O método de redução cíclica é muito útil na solução de sistemas de equações tridiagonais. Esse método também pode ser aplicado quando a matriz do sistema é bloco-tridiagonal ou bloco-Toeplitz. Esse algoritmo se destaca pela sua simplicidade e por ser facilmente paralelizável [9, 14].

A ideia principal do método de redução cíclica consiste em eliminar a metade das variáveis do sistema, reagrupar as equações em um sistema com apenas metade das incógnitas e continuar repetindo esse processo até se chegar em um sistema com apenas duas incógnitas. Esse procedimento pode ser desenvolvido de forma muito simples no caso de um sistema tridiagonal já que as incógnitas podem ser eliminadas de modo tal que o novo sistema (com metade das variáveis) também é tridiagonal.

Considere as três equações consecutivas

$$\begin{aligned} E_{j-1}: &\quad a_{j-1}x_{j-2} + b_{j-1}x_{j-1} + c_{j-1}x_j = d_{j-1}, \\ E_j: &\quad a_j x_{j-1} + b_j x_j + c_j x_{j+1} = d_j, \\ E_{j+1}: &\quad a_{j+1}x_j + b_{j+1}x_{j+1} + c_{j+1}x_{j+2} = d_{j+1}. \end{aligned}$$

Aplicando a transformação

$$E_j - \alpha_j E_{j-1} - \beta_j E_{j+1} \to E'_j \tag{3.2.5}$$

em que

$$\alpha_j = \frac{a_j}{b_{j-1}}, \quad \beta_j = \frac{c_j}{b_{j+1}}, \tag{3.2.6}$$

obtemos a nova equação

$$E'_j:\ a'_j x_{j-2} + b'_j x_j + c'_j x_{j+2} = d'_j \tag{3.2.7}$$

onde

$$\begin{aligned} a'_j &= -\alpha_j a_{j-1} \\ b'_j &= b_j - \alpha_j c_{j-1} - \beta_j a_{j+1} \\ c'_j &= -\beta_j c_{j+1} \\ d'_j &= d_j - \alpha_j d_{j-1} - \beta_j d_{j+1}. \end{aligned} \tag{3.2.8}$$

É importante notar que na nova equação E'_j aparecem explicitamente apenas as incógnitas com índices $j-2$, i, $j+2$.

Temos que observar que no caso $j = 1$ ou $j = n$ a transformação envolverá somente duas equações E_1 e E_2 ou E_{n-1} e E_n. Nesses casos, nas equações (3.2.5)–(3.2.8), temos que desconsiderar os termos e as parcelas contendo α_1 quando $j = 1$ e β_n quando $j = n$. Como consequência, a nova equação obtida (que seria análoga à (3.2.7)) contem apenas as incógnitas x_1 e x_3 se $j = 1$ ou x_{n-2} e x_n se $j = n$.

Considerando que n é par, quando aplicamos essas transformações para $j = 2, 4, \ldots, n-2, n$ chegamos em um novo sistema tridiagonal para as incógnitas $x_2, x_4, \ldots, x_{n-2}, x_n$ constituído pelas $n/2$ equações: $E'_2, E'_4, \ldots, E'_{n-2}, E'_n$. Além disso, quando as soluções desse novo sistema forem conhecidas, poderemos usar as equações ímpares (E_1, E_3, $\ldots$, E_{n-3}, E_{n-1}) do sistema inicial para calcular x_1, x_3, $\ldots$, x_{n-3}, x_{n-1}. De fato, se x_{j-1} e x_{j+1} são conhecidos, então podemos calcular x_j diretamente por substituição na equação E_j

$$x_j = \frac{d_j - a_j x_{j-1} - c_j x_{j+1}}{b_j}. \tag{3.2.9}$$

Veja que quando $j = 1$, devemos desconsiderar a parcela associada com a_1.

Assim, para se obter a solução do sistema inicial com n incógnitas basta resolver um sistema reduzido contendo apenas $n/2$ incógnitas e depois aplicar um simples processo de substituição para calcular as $n/2$ incógnitas restantes. Notamos também que um raciocínio semelhante pode ser aplicado no caso de n ímpar, em que temos que resolver primeiramente um sistema reduzido com apenas $(n-1)/2$ incógnitas e depois, através do processo de substituição, obter as $(n+1)/2$ incógnitas restantes.

Aplicando esses processos recursivamente chegamos ao *método de redução cíclica*. Para simplificar nossa notação, vamos considerar o caso em que $n = 2^p$ com $p \geq 1$. O método é constituído por duas etapas: redução e substituição.

Durante a redução, na primeira iteração obtemos um sistema tridiagonal reduzido contendo as $n/2$ equações E'_{2j} para as incógnitas x_{2j} em que $j = 1, 2, \ldots, n/2$. Na segunda iteração, a partir do sistema reduzido resultante na primeira iteração, obtemos um novo sistema reduzido com $n/4$ equações E''_{4j} para as incógnitas x_{4j} em que $j = 1, 2, \ldots, n/4$. Finalmente, após $p-1$ iterações, como resultado do processo de redução obtemos um sistema com duas equações para calcular as incógnitas $x_{n/2}$ e x_n.

De modo geral, na r-ésima iteração da etapa de redução obtemos um sistema reduzido da forma

$$E^{(r)}_{j\delta}: \quad a^{(r)}_{j\delta} x_{(j-1)\delta} + b^{(r)}_{j\delta} x_{j\delta} + c^{(r)}_{j\delta} x_{(j+1)\delta} = d^{(r)}_{j\delta}$$

em que $\delta = 2^r$ e $j = 1, 2, \ldots, n/2^r$. Observe que o caso $r = 0$ corresponde ao sistema inicial, mas quando $r \geq 1$ os coeficientes são determinados de acordo com

as equações

$$\begin{aligned}
a_{j\delta}^{(r)} &= -\alpha_j a_{j\delta-\delta/2}^{(r-1)} \\
b_{j\delta}^{(r)} &= b_{j\delta}^{(r-1)} - \alpha_j c_{j\delta-\delta/2}^{(r-1)} - \beta_j a_{j\delta+\delta/2}^{(r-1)} \\
c_{j\delta}^{(r)} &= -\beta_j c_{j\delta+\delta/2}^{(r-1)} \\
d_{j\delta}^{(r)} &= d_{j\delta}^{(r-1)} - \alpha_j d_{j\delta-\delta/2}^{(r-1)} - \beta_j d_{j\delta+\delta/2}^{(r-1)}.
\end{aligned} \tag{3.2.10}$$

em que

$$\alpha_j = \frac{a_{j\delta}^{(r-1)}}{b_{j\delta-\delta/2}^{(r-1)}}, \quad \beta_j = \frac{c_{j\delta}^{(r-1)}}{b_{j\delta+\delta/2}^{(r-1)}}. \tag{3.2.11}$$

Notamos que quando $r = 1$, essas equações coincidem com (3.2.8) e (3.2.11).

A etapa de substituição começa após o cálculo de $x_{n/2}$ e x_n. Na primeira iteração dessa etapa, calculamos as duas incógnitas $x_{n/4}$ e $x_{3n/4}$, substituindo $x_{n/2}$ e x_n no sistema reduzido obtido na iteração $p-2$. Na próxima iteração, substituímos as 4 incógnitas conhecidas $x_{n/4}$, $x_{n/2}$, $x_{3n/4}$ e x_n no sistema reduzido obtido na $(p-3)$-ésima iteração da etapa de redução para calcular mais 4 incógnitas: $x_{n/8}$, $x_{3n/8}$, $x_{5n/8}$ e $x_{7n/8}$. Finalmente, após concluirmos a $(p-1)$-ésima iteração da etapa de substituição, teremos calculado todas as incógnitas correspondentes ao sistema inicial.

Em geral, durante a r-ésima iteração do processo de substituição $(1 \leq r \leq p-1)$ calculamos 2^r novas incógnitas $x_{j\delta}$, $j = 1, 3, \ldots, 2^{(r+1)}-1$, a partir das já conhecidas 2^r incógnitas $x_{2j\delta}$, $j = 1, 2, \ldots, 2^r$, em que $\delta = n/2^{(r+1)} = 2^{p-r-1}$. Esse processo consiste na substituição das incógnitas conhecidas no sistema reduzido resultante da $(p-r-1)$-ésima iteração da etapa de redução, ou seja

$$x_{j\delta} = \frac{d_{j\delta}^{(p-r-1)} - a_{j\delta}^{(p-r-1)} x_{(j-1)\delta} - c_{j\delta}^{(p-r-1)} x_{(j+1)\delta}}{b_{j\delta}^{(p-r-1)}},$$

para $j = 1, 3, \ldots, 2^{(r+1)} - 1$. Note que no caso $j = 1$ devemos desconsiderar a parcela $a_{\delta}^{(p-r-1)} x_0$.

Uma representação esquemática do método de redução cíclica mostrando as dependências entre os dados é mostrada na Figura 3.1, para o caso $n = 8$. É fácil perceber que as transformações das equações a serem realizadas em cada iteração da etapa de redução podem ser feitas em paralelo, mas é preciso concluir todos os cálculos correspondentes a uma iteração antes de começar a trabalhar na próxima iteração. Uma observação análoga pode ser feita em relação à etapa de substituição.

O método pode ser aplicado quando os elementos da diagonal principal são diferentes de zero em cada iteração da fase de redução. Uma condição suficiente para isso é que a matriz inicial seja estritamente diagonal dominante, pois essa propriedade é conservada a cada iteração, veja [40]. É importante também destacar que sob essas condições o método é numericamente estável [2].

Implementação sequencial

Por simplicidade, vamos considerar o caso em que a dimensão do sistema é uma potência de 2 com expoente $p \geq 2$. Antes de discutir a implementação paralela em CUDA, vamos apresentar uma implementação sequencial do método (veja o código 3.1). A função, `alg_rc()` mostrada abaixo recebe como entrada ponteiros para os vetores dos coeficientes das diagonais (`a`, `b`, `c`), o vetor do termo constante

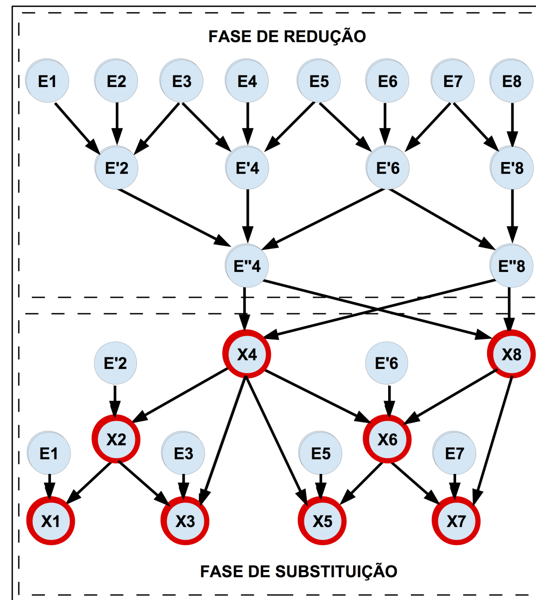


Figura 3.1: Relações e dependências do método de redução cíclica para o caso de um sistema tridiagonal de 8 equações.

(`d`) e o vetor das incógnitas (`x`) e também a dimensão do sistema (`dim_sist`). Ao concluir a chamada dessa função, a solução do sistema estará armazenada no vetor de incógnitas (`x`).

Durante a etapa de redução, a função vai atualizando as diferentes componentes dos vetores dos coeficientes e do termo constante. São usados dois laços `for` aninhados, o externo (linhas 9–28) está relacionado com as iterações da etapa e o interno (linhas 13–26) com o cálculo dos coeficientes de cada equação que precisa ser transformada (linhas 19–25). Na etapa de substituição, também são usados dois laços `for` aninhados, o externo (linhas 39–51) para realizar as iterações dessa etapa e o interno (linhas 41–48) para o cálculo de cada uma das incógnitas correspondentes a essa iteração.

```
1  void alg_rc(double *a, double *b, double *c, double *d, double
       *x, int dim_sist) {
2     int num_iter = (int) ((log2((double)dim_sist/2) + log2((
          double)dim_sist))/2);
3     int num_eq_iter = dim_sist/2;
4     int delta = 1;
5     int var, i1, i2, i, i_esq, i_dir;
6     double alfa, beta, det;

8     //etapa de redução
9     for (int j = 0; j < num_iter; j++) {
10       var = delta;
11       delta *= 2;

13       for (int l = 0; l < num_eq_iter; l++) {
14          i = delta * l + delta - 1;
15          i_esq = i - var;
```

```
16          i_dir = i + var;
17          if (i_dir >= dim_sist) i_dir = dim_sist - 1;

19          // calculo dos coeficientes do sistema reduzido
20          alfa = a[i] / b[i_esq];
21          beta = c[i] / b[i_dir];
22          b[i] = b[i] - c[i_esq] * alfa - a[i_dir] * beta;
23          d[i] = d[i] - d[i_esq] * alfa - d[i_dir] * beta;
24          a[i] = -a[i_esq] * alfa;
25          c[i] = -c[i_dir] * beta;
26       }
27       num_eq_iter /= 2;
28    }

30    i1 = delta - 1;
31    i2 = 2 * delta - 1;
32    // calcula x[i1] e x[i2] resolvendo o sistema 2x2
33    det = b[i2] * b[i1] - c[i1] * a[i2];
34    x[i1] = (b[i2] * d[i1] - c[i1] * d[i2]) / det;
35    x[i2] = (d[i2] * b[i1] - d[i1] * a[i2]) / det;

37    //etapa de substitução
38    int num_sol_iter = 2;
39    for (int j = 0; j < num_iter; j++) {
40       var = delta/2;
41       for (int l = 0; l < num_sol_iter; l++ ) {
42          int i = delta * l + var - 1;
43          if (l==0) {
44             x[i] = (d[i] - c[i] * x[i+var]) / b[i];
45          } else {
46             x[i] = (d[i] - a[i] * x[i-var] - c[i] * x[i+var]) /
                    b[i];
47          }
48       }
49       delta /= 2;
50       num_sol_iter *= 2;
51    }
52 }
```

Código 3.1: Implementação sequencial do método de redução cíclica.

Implementação paralela em CUDA

A seguir, apresentamos uma implementação paralela relativamente simples do método de redução cíclica para GPU, considerando que usaremos apenas um bloco unidimensional de threads. Como precisamos transformar, no máximo, `dim_sist`/2 equações, usaremos exatamente essa quantidade de threads por bloco.

O kernel `rc_kernel()` é apresentado no código 3.2. Ele recebe como entrada: os vetores dos coeficientes das três diagonais (`d_a`, `d_b`, `d_c`), o vetor do termo constante (`d_d`) e o vetor das incógnitas (`d_x`). Apenas os elementos do vetor solução `d_x` serão modificados no kernel.

De forma geral, o kernel `rc_kernel()` não é muito diferente da função `rc_seq()`. A primeira parte do código (linhas 3–32) é preparatória e nela são definidas e inicializadas algumas variáveis auxiliares que não aparecem no código sequencial. Na

linha 3, a variável `i` que identifica cada thread é inicializada. De certo modo, essa variável serve para substituir a variável `l` que aparece nos laços `for` mais internos da implementação sequencial. Além disso, como cada thread realiza as tarefas de forma independente, observamos que no kernel os laços `for` internos foram substituídos pelos comandos condicionais `if` (linhas 39 e 73) que decidem quais threads devem ficar ociosas.

Outra parte em que chamamos a atenção são as linhas 12–32. Nelas são definidas e inicializadas as variáveis necessárias para fazermos uso da memória compartilhada do bloco (veja seção 2.3.5). Os dados são copiados da memória global da GPU para a memória compartilhada, sendo cada thread responsável pela cópia de duas equações do sistema (note como a variável `i`, que armazena o índice da thread, é convenientemente usada para indexar os elementos dos vetores de dados). Como a dimensão dos vetores pode variar a cada execução do kernel, utilizamos a forma dinâmica de alocação da memória compartilhada (veja a seção 2.3.5). Dessa forma, todas as transformações do sistema são realizadas usando os dados armazenados na memória compartilhada para ganhar em eficiência.

Chamamos a atenção também para a necessidade de uso do comando de sincronização `__syncthreads` (veja seção 2.3.2) para garantir que todas as threads do bloco já possuem as informações necessárias antes de executar os próximos passos (veja as linhas 9, 31, 52, 70 e 83).

```
1  __global__ void rc_kernel(double *d_a, double *d_b, double *d_c
       , double *d_d, double *d_x) {
2      int i = threadIdx.x;
3      int num_eq_iter = blockDim.x;
4      const int dim_sist = blockDim.x * 2;
5      int num_iter = (int) ((log2((double)dim_sist/2) +  log2((
           double)dim_sist))/2);
6      int delta = 1;
7      int var;

9      __syncthreads();

11     //memória compartilhada pelas threads do bloco
12     extern __shared__ char mem_do_bloco[];

14     //ponteiros para acessar a memória compartilhada do bloco
15     //alocada externamente através da configuração do kernel
16     double* a = (double*)mem_do_bloco;
17     double* b = (double*)&a[dim_sist];
18     double* c = (double*)&b[dim_sist];
19     double* d = (double*)&c[dim_sist];
20     double* x = (double*)&d[dim_sist];

22     //copiando dados para a memória do bloco
23     a[i] = d_a[i];
24     a[i + blockDim.x] = d_a[i + blockDim.x];
25     b[i] = d_b[i];
26     b[i + blockDim.x] = d_b[i + blockDim.x];
27     c[i] = d_c[i];
28     c[i + blockDim.x] = d_c[i + blockDim.x];
29     d[i] = d_d[i];
30     d[i + blockDim.x] = d_d[i + blockDim.x];
31     __syncthreads();
```

```
33     //etapa de eliminação
34     for (int j = 0; j < num_iter; j++) {
35        var = delta;
36        delta *= 2;

38        if (i < num_eq_iter)  {
39           int i1 = delta * i + delta - 1;
40           int i_esq = i1 - var;
41           int i_dir = i1 + var;

43           if (i_dir >= dim_sist) i_dir = dim_sist - 1;
44           double alfa = a[i1] / b[i_esq];
45           double beta = c[i1] / b[i_dir];
46           b[i1] = b[i1] - c[i_esq] * alfa - a[i_dir] * beta;
47           d[i1] = d[i1] - d[i_esq] * alfa - d[i_dir] * beta;
48           a[i1] = -a[i_esq] * alfa;
49           c[i1] = -c[i_dir] * beta;
50        }
51        num_eq_iter /= 2;
52        __syncthreads();
53     }

55     //solução do sistema 2x2
56     //precisamos de apenas uma thread para fazer isso
57     if (i==0)  {
58        int i1 = delta - 1;
59        int i2 = 2 * delta - 1;
60        // calcula x[i1] e x[i2] resolvendo o sistema 2x2
61        double det = b[i2] * b[i1] - c[i1] * a[i2];
62        x[i1] = (b[i2] * d[i1] - c[i1] * d[i2]) / det;
63        x[i2] = (d[i2] * b[i1] - d[i1] * a[i2]) / det;
64     }

66     //etapa de substituição
67     int num_sol_iter = 2;
68     for (int j = 0; j < num_iter; j++) {
69        var = delta/2;
70        __syncthreads();

72        if (i < num_sol_iter) {
73           int i1 = delta * i + var - 1;
74           if (i==0) {
75              x[i1] = (d[i1] - c[i1] * x[i+var]) / b[i1];
76           } else {
77              x[i1] = (d[i1] - a[i1] * x[i-var] - c[i] * x[i+var
                    ]) / b[i1];
78           }
79        }
80        delta /= 2;
81        num_sol_iter *= 2;
82     }
83     __syncthreads();

85     //copiando o resultado da memória do bloco para a GPU
```

```
86    //cada thread é responsável por dois valores
87    d_x[i] = x[i];
88    d_x[i + blockDim.x] = x[i + blockDim.x];
89 }
```

Código 3.2: Kernel para implementação paralela do método de redução cíclica.

O código 3.3 mostra um exemplo de implementação da função `main()` para um programa que resolve um sistema tridiagonal pelo método de redução cíclica, usando o kernel apresentado acima (código 3.2). Para o programa funcionar corretamente, é preciso adicionar as linhas de código para o preenchimento dos coeficientes e do termo livre do sistema. Chamamos a atenção para a linha 45, onde lançamos o kernel. Nos parâmetros de configuração temos: 1 bloco, `dim_sist`/2 threads e 5*`quant_mem` *bytes* na memória compartilhada desse bloco. Lembramos que o tratamento dessa memória compartilhada no kernel aparece nas linhas 16–20 do código 3.2.

```
1  int main() {
2     int dim_sist;

4     //adicionar código para inicializar a variável dim_sist
5     //contendo a dimensão do sistema

7     size_t quant_mem = dim_sist*sizeof(double);

9     //variáveis para os dados usados pela CPU
10    double* a = (double*)malloc(quant_mem);
11    double* b = (double*)malloc(quant_mem);
12    double* c = (double*)malloc(quant_mem);
13    double* d = (double*)malloc(quant_mem);
14    double* x = (double*)malloc(quant_mem);

16    if (a == NULL || b == NULL || c == NULL || d == NULL || x ==
         NULL) {
17       fprintf(stderr, "Memoria insuficiente\n");
18       exit(EXIT_FAILURE);
19    }

21    //inserir código para preencher os vetores de dados

23    //variáveis para os dados usados pela GPU
24    double* d_a;
25    double* d_b;
26    double* d_c;
27    double* d_d;
28    double* d_x;

30    //reservar memória na GPU
31    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_a,quant_mem) );
32    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_b,quant_mem) );
33    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_c,quant_mem) );
34    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_d,quant_mem) );
35    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_x,quant_mem) );

37    //copiar os dados da CPU para a GPU
38    CUDA_SAFE_CALL( cudaMemcpy( d_a, a, quant_mem,
```

```
        cudaMemcpyHostToDevice) );
39    CUDA_SAFE_CALL( cudaMemcpy( d_b, b, quant_mem,
        cudaMemcpyHostToDevice) );
40    CUDA_SAFE_CALL( cudaMemcpy( d_c, c, quant_mem,
        cudaMemcpyHostToDevice) );
41    CUDA_SAFE_CALL( cudaMemcpy( d_d, d, quant_mem,
        cudaMemcpyHostToDevice) );

43    //lançar o kernel usando um único bloco com dim_sist/2
        threads
44    //reserva  memória compartilhada  para o bloco
45    rc_kernel <<<1, dim_sist/2, 5*quant_mem>>> (d_a, d_b, d_c,
        d_d, d_x);

47    //certificar-se que o kernel foi lançado com sucesso
48    CUDA_SAFE_CALL( cudaGetLastError() );

50    //transferir o resultado da GPU para a CPU
51    CUDA_SAFE_CALL( cudaMemcpy(x, d_x, quant_mem,
        cudaMemcpyDeviceToHost) );

53    //limpar a memória  da GPU
54    CUDA_SAFE_CALL( cudaFree(d_a) );
55    CUDA_SAFE_CALL( cudaFree(d_b) );
56    CUDA_SAFE_CALL( cudaFree(d_c) );
57    CUDA_SAFE_CALL( cudaFree(d_d) );
58    CUDA_SAFE_CALL( cudaFree(d_x) );

60    //inserir código que faz uso da solução

62    free(a);
63    free(b);
64    free(c);
65    free(d);
66    free(x);

68    CUDA_SAFE_CALL( cudaDeviceReset() );
69    exit(EXIT_SUCCESS);
70 }
```

Código 3.3: Modelo de função `main()` para a solução de um sistema tridiagonal na GPU.

A implementação apresentada possui limitações devido ao fato de definir apenas um bloco de threads e fazer uso da memória compartilhada para armazenar os dados. De qualquer forma, a maior parte do código desenvolvido pode ser útil quando desejamos resolver um grande número de sistemas tridiagonais de mesma dimensão, não sendo esta muito grande (dimensão ≤ 1024). Vamos então descrever as mudanças que devemos fazer nos códigos 3.2 e 3.3 para contemplar esse cenário.

Note que vamos ter um bloco de threads para resolver cada um dos sistemas. Também vamos considerar que nos vetores de entrada do kernel `d_a`, ..., `d_d` se encontram os dados de todos os sistemas, formando um bloco contínuo para cada sistema em cada um desses vetores, e no vetor `d_x` vamos armazenar as soluções de acordo com essa ordem.

Logo, na parte inicial do código 3.2, devemos trocar as linhas 14–30 pelas se-

guintes:

```
14 // ponteiros para acessar a memória compartilhada do bloco
15 // alocada externamente na configuração do kernel
16 double* a = (double*)mem_do_bloco;
17 double* b = (double*)&a[dim_sist];
18 double* c = (double*)&b[dim_sist];
19 double* d = (double*)&c[dim_sist];
20 double* x = (double*)&d[dim_sist];

22 // copiando dados para a memória do bloco
23 a[i] = d_a[blockIdx.x*dim_sist + i];
24 a[i + blockDim.x] = d_a[blockIdx.x*dim_sist  + blockDim.x+ i];
25 b[i] = d_b[blockIdx.x*dim_sist + i];
26 b[i + blockDim.x] = d_b[blockIdx.x*dim_sist + blockDim.x + i];
27 c[i] = d_c[blockIdx.x*dim_sist + i];
28 c[i + blockDim.x] = d_c[blockIdx.x*dim_sist + blockDim.x + i];
29 d[i] = d_d[blockIdx.x*dim_sist + i];
30 d[i + blockDim.x] = d_d[blockIdx.x*dim_sist + blockDim.x + i];
```

Código 3.4: Primeira modificação do kernel 3.2.

No final do código, temos que trocar as linhas 85–88 pelas seguintes:

```
85 // copiando o resultado da memória do bloco para a GPU
86 // cada thread é responsável por dois valores
87 d_x[blockIdx.x*dim_sist + i] = x[i];
88 d_x[blockIdx.x*dim_sist + blockDim.x + i] = x[i + blockDim.x];
```

Código 3.5: Segunda modificação do kernel 3.2.

Na função `main()` do código 3.3, também devemos fazer algumas mudanças. Primeiramente é necessário adicionar uma nova variável substituindo a linha 2 pela seguinte:

```
2 int dim_sist, n_sists;
```

A nova variável `n_sists` deve ser inicializada com o número de sistemas a serem resolvidos. Também é preciso trocar a variável `quant_mem` nas linhas 10–14, 31–35, 38–41 e 51 pela expressão: `n_sists * quant_mem`. Finalmente, as linhas 43-45 (onde o kernel é lançado) devem ser substituídas pelas seguintes:

```
43 // lançar o kernel usando n_sists blocos de threads
44 // com dim_sist/2 threads cada
45 rc_kernel<<<n_sists, dim_sist/2, 5*quant_mem>>>(d_a, d_b, d_c,
       d_d, d_x);
```

3.2.2 Método de redução cíclica paralela

Este método é baseado nas mesmas transformações que o método de redução cíclica. Aqui também, por simplicidade, consideramos que $n = 2^p$ com $p > 1$.

Como foi visto na seção anterior, usando as transformações (3.2.5) podemos obter as equações $E'_2, E'_4, \ldots, E'_n$ que formam um sistema tridiagonal para determinar as $n/2$ incógnitas $x_2, x_4, \ldots, x_n$. De forma análoga, podemos chegar em outro sistema tridiagonal constituído pelas equações $E'_1, E'_3, \ldots, E'_{n-1}$ para determinar as $n/2$ incógnitas restantes x_1, x_3, $\ldots$, x_{n-1}.

Aplicando esse processo de redução recursivamente a cada sistema resultante, após um certo número de iterações, chegamos em $n/2$ sistemas de 2×2 que podem

ser resolvidos facilmente para se obter a solução do sistema original. Esse processo iterativo é o *método de redução cíclica paralela*. Observamos que nesse novo método não é preciso aplicar as substituições.

Uma representação esquemática do método mostrando as dependências entre os dados é dada na Figura 3.2 para o caso $n = 8$.

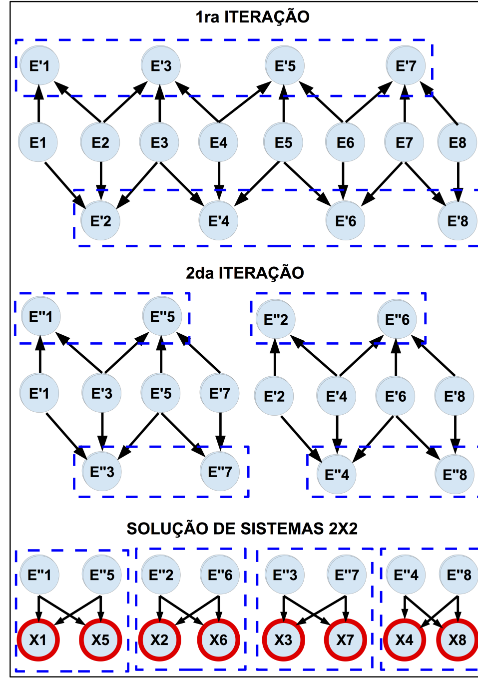


Figura 3.2: Relações e dependências do método de redução cíclica paralela para o caso de um sistema tridiagonal com 8 equações.

A cada iteração, a dimensão dos sistemas sendo transformados cai pela metade, logo são necessárias $p-1$ iterações para se chegar em um conjunto de sistemas com apenas duas incógnitas cada.

De modo geral, na r-ésima iteração obtemos as equações

$$E_j^{(r)}: \quad a_j^{(r)} x_{j-\delta} + b_j^{(r)} x_j + c_j^{(r)} x_{j+\delta} = d_j^{(r)} \tag{3.2.12}$$

em que $\delta = 2^r$ e $j = 1, 2, \ldots, n$.

Observe que o caso $r = 0$ corresponde ao sistema inicial, mas quando $r \geq 1$ os coeficientes devem ser calculados de acordo com as equações

$$\begin{aligned} a_j^{(r)} &= -\alpha_j a_{j-\delta/2}^{(r-1)} \\ b_j^{(r)} &= b_j^{(r-1)} - \alpha_j c_{j-\delta/2}^{(r-1)} - \beta_j a_{j+\delta/2}^{(r-1)} \\ c_j^{(r)} &= -\beta_j c_{j+\delta/2}^{(r-1)} \\ d_j^{(r)} &= d_j^{(r-1)} - \alpha_j d_{j-\delta/2}^{(r-1)} - \beta_j d_{j+\delta/2}^{(r-1)}. \end{aligned} \tag{3.2.13}$$

em que

$$\alpha_j = \frac{a_j^{(r-1)}}{b_{j-\delta/2}^{(r-1)}}, \quad \beta_j = \frac{c_j^{(r-1)}}{b_{j+\delta/2}^{(r-1)}}. \tag{3.2.14}$$

Observamos que nessas equações, as parcelas contendo termos com o índice fora do intervalo permissível (menor que 1 ou maior que n) devem ser desconsideradas.

Como é esperado, as n equações (3.2.12) podem ser reagrupadas para formar δ sistemas tridiagonais de dimensão n/δ da seguinte forma. Para cada $s \in \{1, 2, \dots, \delta\}$ o s-ésimo sistema está constituído pelas n/δ equações: $E^{(r)}_{s+(l-1)\delta}$ em que $l = 1, 2, \dots, n/\delta$.

Importante notar que durante a execução de cada iteração, o cálculo das novas equações pode ser realizado simultaneamente, mas é necessário obter todas essas equações antes de passar para a próxima iteração. Essa característica da etapa de redução torna possível a sua paralelização. Além disso, o resultado dessa etapa produz $n/2$ sistemas independentes que podem ser resolvidos simultaneamente, contribuindo também para a paralelização do método.

Quando comparamos este último método com o método de redução cíclica apresentado anteriormente, notamos várias diferenças. Em cada iteração da etapa de redução no novo método temos que transformar todas as n equações, enquanto que no primeiro método estudado a quantidade de equações a serem transformadas caia pela metade a cada iteração. No primeiro método, após a etapa de redução, temos que resolver um sistema de 2×2 e em seguida executar a etapa de substituição. Em contraposição, neste novo método, após a etapa de redução temos que resolver $n/2$ sistemas de 2×2 calculando-se assim todas as incógnitas do sistema inicial, portanto não é necessário incluir a etapa de substituições. Podemos concluir que em comparação ao método de redução cíclica, o método de redução cíclica paralela precisa de um número menor de passos, ainda que em cada passo seja necessário realizar um trabalho um pouco mais intenso. De qualquer forma, este novo método é mais adequado para o processamento em paralelo [14].

Implementação sequencial

Por simplicidade vamos considerar o caso em que a dimensão do sistema é uma potência de 2 com expoente $p \geq 2$.

Começamos discutindo uma implementação sequencial do método de redução cíclica paralela. Quando comparamos com o código 3.1, que implementa a função `alg_rc()`, percebemos que quase todas as variáveis possuem o mesmo significado nos dois casos. A única diferença é que no novo código é necessário introduzir os vetores temporários `t_a`, ..., `t_d` pois nesse método todos os dados são modificados simultaneamente. Também pela mesma razão, o laço `for` mais interno na etapa de redução sempre percorre todas equações. Além disso, as soluções são calculadas diretamente sem precisarmos de fazer um processo de substituição, por isso temos um único laço `for` onde elas são determinadas.

```
1  void alg_rcp(double *a, double *b, double *c, double *d, double
       *x, int dim_sist) {
2      size_t quant_mem = sizeof(double)*dim_sist;
3      int num_iter = (int) ((log2((double)dim_sist/2) + log2((
           double)dim_sist))/2);
4      //observe que var = delta/2
5      int var = 1;
6      int i1, i2, i_esq, i_dir;
7      double alfa, beta, det;

9      //variáveis temporárias para não sobreescrever os dados
           necessários
```

```
10     double * t_a = (double*)malloc(quant_mem);
11     double * t_b = (double*)malloc(quant_mem);
12     double * t_c = (double*)malloc(quant_mem);
13     double * t_d = (double*)malloc(quant_mem);

15     if (t_a==NULL || t_b==NULL || t_c==NULL || t_d==NULL )  {
16        fprintf(stderr, "Memoria insuficiente\n" );
17        exit(EXIT_FAILURE);
18     }

20     //fase de redução
21     for (int j = 0; j < num_iter; j++) {
22        for (int i = 0; i < dim_sist; i++) {
23           i_esq = i - var;
24           i_dir = i + var;
25           if (i_esq < 0) {
26              i_esq = 0;
27           }
28           if (i_dir >= dim_sist) {
29              i_dir = dim_sist - 1;
30           }

32           //calcular as equações transformadas
33           alfa = a[i] / b[i_esq];
34           beta = c[i] / b[i_dir];
35           t_b[i] = b[i] - c[i_esq] * alfa - a[i_dir] * beta;
36           t_d[i] = d[i] - d[i_esq] * alfa - d[i_dir] * beta;
37           t_a[i] = -a[i_esq] * alfa;
38           t_c[i] = -c[i_dir] * beta;
39        }
40        //atualizar os dados das equações
41        memcpy(a, t_a, quant_mem);
42        memcpy(b, t_b, quant_mem);
43        memcpy(c, t_c, quant_mem);
44        memcpy(d, t_d, quant_mem);

46        var *= 2;
47     }

49     //solução dos sistemas 2x2
50     for (int j = 0; j < var; j++)  {
51        i1 = j;
52        i2 = j + var;
53        det = t_b[i2] * t_b[i1] - t_c[i1] * t_a[i2];
54        x[i1] = (t_b[i2] * t_d[i1] - t_c[i1] * t_d[i2]) / det;
55        x[i2] = (t_d[i2] * t_b[i1] - t_d[i1] * t_a[i2]) / det;
56     }
57     //liberar os espaços de memória alocados
58     free(t_a); free(t_b); free(t_c); free(t_d);
59  }
```

Código 3.6: Implementação sequencial do método de redução cíclica paralela.

Implementação paralela em CUDA

Apresentamos no código 3.7 o kernel `rcp_kernel()` que implementa o método de redução cíclica paralela. A ideia básica para passar da implementação sequencial para a paralela é a mesma que utilizamos na implementação do método de redução cíclica (código 3.2). Usamos um único bloco de threads para resolver o sistema e cada thread é responsável pela transformação de uma única equação, de modo que precisamos de `dim_sist` threads. Assim, podemos observar que durante a etapa de redução todas as threads permanecem ativas, daí a ausência do laço mais interno no código. Por outro lado, a etapa de solução do sistema envolve apenas metade das threads (ou seja, `dim_sist`/2 threads). Observamos como a variável `i`, que identifica cada thread, é usada para desativar algumas delas durante essa etapa.

Chamamos a atenção para as linhas 26–48 em que são implementadas as transformações das equações. Como neste método todas as equações são modificadas, é preciso evitar a modificação indevida dos dados associados a uma thread que são necessários para uma outra thread efetuar seus cálculos. Isto é garantido com a introdução das variáveis `alfa`, `beta`, `v_dir` e `v_esq` (locais para cada thread) e a sincronização das threads.

```
1   __global__ void rcp_kernel(double *d_a, double *d_b, double *
        d_c, double *d_d, double *d_x) {
2      int i = threadIdx.x;
3      const int dim_sist = blockDim.x;
4      int num_iter = (int) ((log2((double)dim_sist/2) + log2((
          double) dim_sist))/2);
5      int var = 1;
6      int i_esq, i_dir;
7      double alfa, beta, v_dir, v_esq;

9      //usando a memória compartilhada
10     //reservada no lançamento do kernel
11     extern __shared__ char mem_do_bloco[];
12     double* a = (double*)mem_do_bloco;
13     double* b = (double*)&a[dim_sist];
14     double* c = (double*)&b[dim_sist];
15     double* d = (double*)&c[dim_sist];
16     double* x = (double*)&d[dim_sist];

18     a[i] = d_a[i];
19     b[i] = d_b[i];
20     c[i] = d_c[i];
21     d[i] = d_d[i];
22     __syncthreads();

24     //etapa de eliminação
25     for (int j = 0; j < num_iter; j++)  {
26        i_esq = i - var;
27        i_dir = i + var;

29        if (i_esq < 0) {
30           i_esq = 0;
31        }
32        if (i_dir >= dim_sist) {
33           i_dir = dim_sist - 1;
34        }
```

```
35        alfa = a[i] / b[i_esq];
36        beta = c[i] / b[i_dir];
37        v_dir = d[i_dir];
38        v_esq = d[i_esq];
39        __syncthreads();

41        b[i] = b[i] - c[i_esq] * alfa - a[i_dir] * beta;
42        d[i] = d[i] - v_esq*alfa-v_dir*beta;
43        v_esq = a[i_esq];
44        v_dir = c[i_dir];
45        __syncthreads();

47        a[i] = -v_esq * alfa;
48        c[i] = -v_dir * beta;
49        __syncthreads();
50        var *= 2;
51     }

53     //solução dos sistemas 2x2
54     if (i < var)  {
55        int i1 = i;
56        int i2 = i + var;
57        double det = b[i1]*b[i2]-c[i1]*a[i2];
58        x[i1] = (b[i2]*d[i1]-c[i1]*d[i2])/det;
59        x[i2] = (d[i2]*b[i1]-d[i1]*a[i2])/det;
60     }
61     __syncthreads();

63     d_x[i] = x[i];
64  }
```

Código 3.7: Kernel para implementação paralela do método de redução cíclica paralela.

Este kernel pode ser usado de forma análoga ao kernel `rc_kernel()` no código 3.3. Para isso, é suficiente trocar as linhas 43-45 pelas seguintes:

```
43  // lança o kernel usando um único bloco com dim_sist threads
44  // reserva  memória compartilhada para o bloco
45  rcp_kernel<<<1, dim_sist, 5*quant_mem >>>(d_a, d_b, d_c, d_d,
        d_x);
```

Não é dificil introduzir modificações nos códigos 3.7 e 3.3 para implementar a solução de vários sistemas tridiagonais pelo método de redução cíclica paralela usando a GPU. Isto pode ser feito de forma análoga às modificações introduzidas nos códigos 3.2 e 3.3 que foram apresentadas nas páginas 63–64.

Observamos que da mesma forma que os códigos do método de redução cíclica desenvolvidos para GPU (códigos 3.2, 3.3 e suas modificações), o código 3.7 e as modificações propostas acima apresentam limitações em relação à dimensão do sistema (ou dos sistemas de equações) a serem resolvidos devido ao lançamento do kernel com apenas um bloco de threads e à utilização da memória compartilhada. Para superar essa limitação, uma alternativa é usar mais de um bloco de threads na resolução do sistema, mas isso traz algumas dificuldades no tratamento da sincronização das threads e no acesso aos dados. Algumas estratégias para o tratamento dessas dificuldades são apresentadas no capítulo 2 do livro [18].

3.3 Fatoração LU

Como mencionamos na introdução deste capítulo, o método de fatoração LU consiste na aplicação do algoritmo de eliminação gaussiana para se obter uma representação especial da matriz do sistema. Em particular, se o processo de eliminação for realizado sem usar pivoteamento, então a matriz do sistema poderá ser escrita na forma

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{3.3.15}$$

em que a matriz $\mathbf{L}$ é triangular inferior com todos seus elementos da diagonal principal iguais a 1 e $\mathbf{U}$ é triangular superior. Na matriz $\mathbf{L}$ se encontram codificadas todas as transformações realizadas durante el processo de eliminação gaussiana e a matriz $\mathbf{U}$ é composta pelos coeficientes do sistema triangular superior resultante do processo de eliminação.

A grande utilidade dessa fatoração vem do fato de que a solução $\mathbf{x}$ pode ser calculada resolvendo primeiro o sistema $\mathbf{L} \cdot \mathbf{y} = \mathbf{d}$ e em seguida $\mathbf{U} \cdot \mathbf{x} = \mathbf{y}$. Mais especificamente, como as matrizes desses sistemas são triangulares, eles podem ser resolvidos eficientemente por substituição com um custo de $O(n^2)$ operações aritméticas, em que n é a dimensão do sistema. Temos que lembrar que o método de eliminação com retrosubstituição possui um custo de $O(n^3)$ operações. Assim sendo, o método de fatoração será muito útil quando temos que resolver vários sistemas de equações lineares que compartilham a mesma matriz dos coeficientes [6, 35].

Lembramos que o processo de eliminação gaussiana consiste de $n-1$ estágios e em cada estágio é obtido um novo sistema equivalente ao inicial, de forma tal que um certo número de variáveis foram eliminadas de algumas equações. O resultado final do processo é um sistema triangular superior de fácil resolução.

Representando por $\mathbf{A}^{(k)}$ a matriz dos coeficientes resultantes do k-ésimo estágio, sabemos que as linhas dessas matrizes satisfazem as relações

$$L_j^{(k)} = L_j^{(k-1)}, \quad j = 1, \ldots, k.$$

$$\left.\begin{aligned} L_j^{(k)} &= L_j^{(k-1)} - m_{jk} \cdot L_k^{(k-1)} \\ m_{jk} &= \frac{a_{jk}^{(k-1)}}{a_{kk}^{(k-1)}} \end{aligned}\right\} \quad j = k+1, \ldots, n.$$

Como consequência, temos que os elementos abaixo da diagonal principal da matriz $\mathbf{L}$ são dados por m_{jk} $(k = 1, \ldots, n-1;\ j = k+1, \ldots, n)$ e a matriz $\mathbf{U} = \mathbf{A}^{(k)}$ [6]. A partir daí, pode-se definir o seguinte algoritmo em que a matriz inicial é transformada de forma tal que nas entradas abaixo da diagonal principal são armazenados os elementos de $\mathbf{L}$ e nas entradas restantes os elementos de $\mathbf{U}$ (veja [11, 35]):

1. **para** $i = 1, \ldots, n-1$, **faça**
2. **para** $j = i+1, \ldots, n$, **faça**
3. $a_{ji} \leftarrow a_{ji}/a_{ii}$
4.   **para** $k = i+1, \ldots, n$, **faça**
5. $a_{jk} \leftarrow a_{jk} - a_{ji} * a_{ik}$
6.   **fim**
7. **fim**
8. **fim**

Esse algoritmo é conhecido como *algoritmo de fatoração LU com atualização imediata (right-looking)*. Observamos que os elementos são atualizados linha a linha sendo que o primeiro elemento atualizado na j-ésima linha (a_{ji}) é depois usado para modificar o restante dos elementos (a_{jk}, com $k > i$). Em princípio, as atualizações correspondentes a essa última fase podem ser realizadas independentemente, ou seja, em paralelo. Mas é possível modificar o algoritmo de forma a reduzir ainda mais as dependências entre os dados e aumentar o grau de paralelismo. Separando os cálculos dos primeiros elementos de cada linha do cálculo dos demais elementos nessas linhas, chegamos no seguinte algoritmo.

1. **para** $i = 1, \ldots, n-1$, **faça**
2.   **para** $j = i+1, \ldots, n$, **faça**
3. $a_{ji} \leftarrow a_{ji}/a_{ii}$
4.   **fim**
5.   **para** $j = i+1, \ldots, n$, **faça**
6.   **para** $k = i+1, \ldots, n$, **faça**
7. $a_{jk} \leftarrow a_{jk} - a_{ji} * a_{ik}$
8.   **fim**
9.   **fim**
10. **fim**

Dessa forma, as transformações realizadas dentro do laço mais externo ficam organizadas em duas etapas: re-escalonamento dos elementos da i-ésima coluna que se encontram abaixo da diagonal; e atualização da $(i+1)$-ésima submatriz principal. Nas Figuras 3.3 e 3.4 apresentamos representações esquemáticas dessas etapas e a dependência entre os dados. Este novo algoritmo é usado como base para a implementação em CUDA da fatoração LU.

Antes de discutirmos a implementação em CUDA, faremos algumas observações gerais sobre o método de fatoração LU. Notamos que para garantir a estabilidade numérica do método de eliminação gaussiana, é preciso usar alguma estratégia de pivotamento. A mais simples e comum delas é o pivotamento parcial (permutação de linhas), nesse caso a fatoração tem a forma

$$\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U} \tag{3.3.16}$$

em que $\mathbf{P}$ é a matriz de permutações [6, 11]. Observamos que nos algoritmos apresentados acima, a estratégia de pivoteamento parcial pode ser incluída sem dificuldades [11, 35].

Outro aspecto interessante é que o método de eliminação gaussiana, e consequentemente a fatoração LU, podem ser aplicados considerando que a matriz inicial é formada por blocos quadrados. A adaptação dos algoritmos básicos para o caso de matrizes em blocos é bastante direta [11]. Vale a pena destacar que esses algoritmos podem ser usados como base na implementação paralela da fatoração LU.

Implementação em CUDA

A organização dos cálculos introduzida no algoritmo modificado permite que em cada etapa a transformação dos elementos envolvidos possa ser feita de forma independente, proporcionando a possibilidade de executar tarefas em paralelo. Na

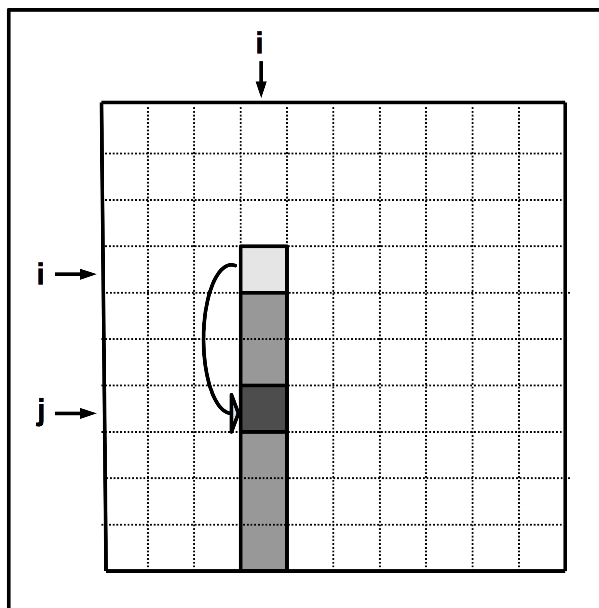


Figura 3.3: Esquema representativo do algoritmo de fatoração *LU* modificado. Primeira etapa: re-escalonamento dos elementos da i-ésima coluna que se encontram abaixo da diagonal principal.

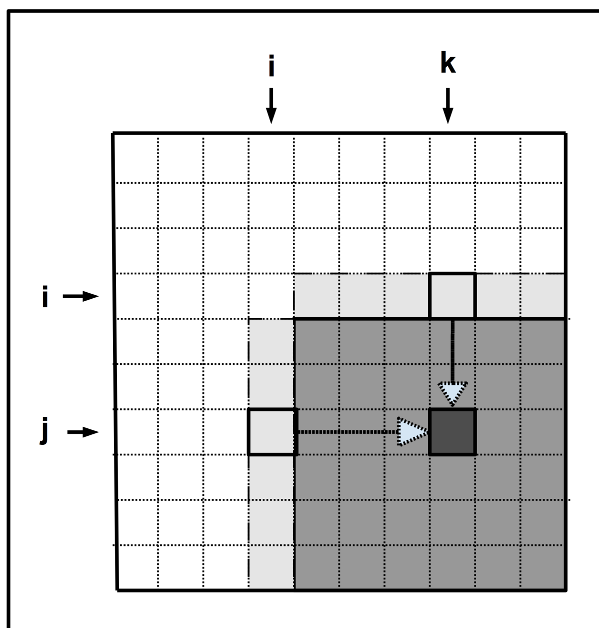


Figura 3.4: Esquema representativo do algoritmo de fatoração *LU* modificado. Segunda etapa: atualização dos elementos da $(i + 1)$-ésima submatriz principal.

implementação do método, definimos a função `alg_lu_gpu()` que será encarregada de trabalhar com os dados na GPU. Ela recebe no argumento `d_m` (um ponteiro do

tipo `double`) os dados da matriz inicial na memória da GPU, e no argumento `dim` a dimensão da matriz.

A matriz é armazenada na ordem lexicográfica, ou seja, uma linha após a outra. Observe no código 3.8 que cada uma das duas etapas do algoritmo é realizada por um kernel diferente e que cada kernel (`lu_calc_col()` e `lu_calc_subm()`) é lançado com sua própria configuração. No código temos a variável `TAM_BLOCO` que está relacionada com a distribuição das threads em cada kernel.

```
1  void alg_lu_gpu( double* d_m, int dim) {
2     int i, n_blocos, TAM_BLOCO = 32;
3
4     for (i = 0; i < dim-1; i++) {
5        n_blocos = ((dim-i-1) + TAM_BLOCO-1) / TAM_BLOCO;
6
7        dim3 g_blocos(n_blocos, n_blocos);
8        dim3 n_threads(TAM_BLOCO,TAM_BLOCO);
9
10       lu_calc_col<<< n_blocos, TAM_BLOCO >>>(d_m, dim, i);
11       CUDA_SAFE_CALL(cudaGetLastError());
12       lu_calc_subm<<< g_blocos, n_threads >>>(d_m, dim, i);
13       CUDA_SAFE_CALL(cudaGetLastError());
14    }
15 }
```

Código 3.8: Função para a fatoração *LU* na GPU.

A distribuição dos blocos de threads e sua correspondência com os elementos da matriz para os kernels `lu_calc_col()` e `lu_calc_subm()` são mostradas nas Figuras 3.5 e 3.6, respectivamente.

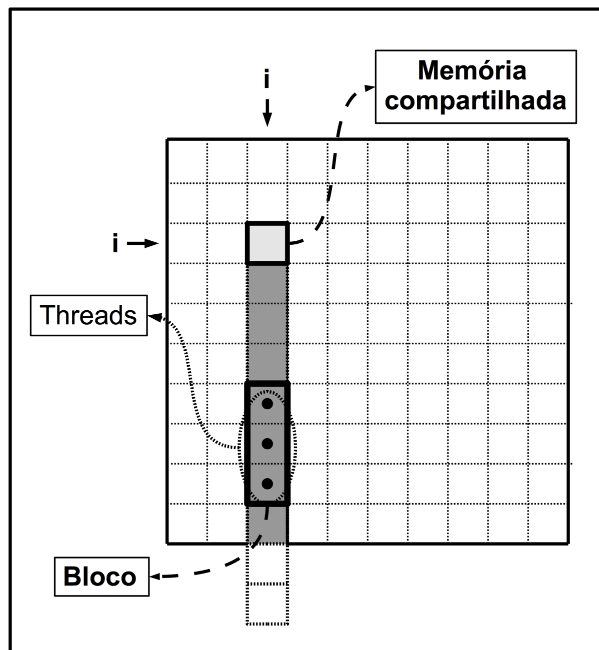


Figura 3.5: Organização dos blocos de threads para o kernel que implementa a etapa de re-escalonamento da coluna.

Na etapa de re-escalonamento implementada no kernel `lu_calc_col()`, quando trabalhamos na i-ésima coluna, precisamos atualizar seus elementos abaixo da diagonal principal (os mesmos são distribuídos em blocos contendo `TAM_BLOCO` threads e cada thread é responsável por re-escalonar um único elemento). A quantidade de blocos necessários para cobrir essa coluna é armazenada em `n_blocos` (calculado na linha 6 do código 3.8). A identificação dos blocos e das threads com as submatrizes e elementos da matriz é feita de cima para baixo.

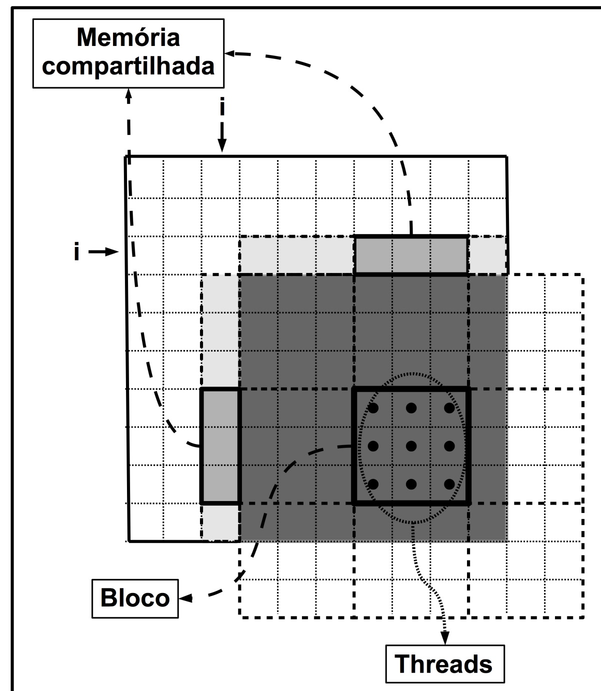


Figura 3.6: Organização dos blocos de threads para o kernel que implementa a etapa de atualização da submatriz.

No kernel `lu_calc_subm()` é implementada a etapa de atualização da submatriz, como mostrado na Figura 3.6. Quando trabalhamos na $(i + 1)$-ésima submatriz, uma grade bidimensional de blocos é usada. Cada bloco é também bidimensional possuindo tamanho `TAM_BLOCO` em cada direção (ou seja, contendo `TAM_BLOCO`×`TAM_BLOCO` threads). Cada thread é responsável por atualizar um único elemento. Para cobrir a submatriz, a grade de blocos é de dimensão `n_blocos` em cada direção (veja as linhas 8 e 9 do código 3.8). A identificação dos blocos e das threads com as submatrizes e elementos da matriz é feita de cima para baixo (para as linhas) e da esquerda à direita (para as colunas), começando pelo $(i + 1)$-ésimo elemento da diagonal principal.

```
1  __global__ void lu_calc_col( double* d_m, int dim, int i ) {
2     __shared__ double a_ii;

4     if (threadIdx.x == 0) {
5        a_ii = d_m[i*(dim+1)];
6     }
7     __syncthreads();
```

```
9     int j  = blockIdx.x * blockDim.x + threadIdx.x + i + 1;
10    if ( j < dim ) {
11       d_m[ j*dim+i ] /= a_ii;
12    }
13 }
```

Código 3.9: Kernel para o re-escalonamento da coluna.

No código 3.9, correspondente ao kernel `lu_calc_col()`, observamos que cada bloco armazena o elemento da coluna que se encontra na diagonal principal na memória compartilhada (assim, esse elemento é lido apenas uma vez por bloco) e a leitura é realizada pela primeira thread do bloco (linhas 3–6). Após a leitura, é necessário sincronizar as threads para garantir que a informação compartilhada estará disponível para todas elas. Em seguida, o elemento sob a responsabilidade da thread é atualizado (linhas 9–11). Note a presença do comando condicional `if` para desativar as threads associadas com elementos "fantasmas" que gerariam acessos a posições inválidas da memória.

```
1  __global__ void lu_calc_subm( double* d_m, int dim, int i) {
2     __shared__ double a_ji[TAM_BLOCO];
3     __shared__ double a_ik[TAM_BLOCO];

5     int j = blockDim.x * blockIdx.x + threadIdx.x + i + 1;
6     int k = blockDim.y * blockIdx.y + threadIdx.y + i + 1;

8     if ((threadIdx.y == 0) && (j < dim)) {
9        a_ji[threadIdx.x] = d_m[ j*dim + i ];
10    }
11    if ((threadIdx.x == 0) && (k < dim)) {
12       a_ik[threadIdx.y] = d_m[ i*dim + k ];
13    }
14    __syncthreads();

16    if ((j < dim) && (k < dim)) {
17       d_m[ j*dim + k ] -= a_ji[threadIdx.x] * a_ik[threadIdx.y
             ];
18    }
19 }
```

Código 3.10: Kernel para a atualização da submatriz.

No código 3.10, que implementa o kernel `lu_calc_subm()`, aplicamos ideias semelhantes ao caso anterior. Nele cada bloco armazena na memória compartilhada uma parte dos elementos das i-ésimas linha e coluna que se encontram à direita e abaixo da diagonal principal, respectivamente. A leitura é realizada apenas pelo grupo de threads do bloco que ficam paralelas à linha ou coluna sendo compartilhada e após a leitura fazemos a sincronização (linhas 3–13). Em seguida, o elemento sob a responsabilidade da thread é atualizado (linhas 15–16). Durante todo o processo é necessário deixar ociosas as threads associadas com elementos "fantasmas".

Apresentamos a seguir o código principal de um programa que faz a fatoração da matriz na GPU (código 3.11). Para o programa funcionar corretamente é preciso adicionar as linhas de código para o preenchimento da matriz. Nas linhas 20–34 mostramos os passos básicos para a fatoração *LU* na GPU, usando as funções discutidas acima e no Capítulo 2.

```
1  int main() {
```

```
    int dim_mat;
    double* m;

    //adicionar  código para inicializar a variável
    //dim_mat (dimensão da matriz)

    size_t quant_mem = dim_mat*dim_mat*sizeof(double);

    m = (double*)malloc(quant_mem);
    if ( m == NULL  ) {
       fprintf(stderr, "Memoria insuficiente\n");
       exit(EXIT_FAILURE);
    }

    //adicionar  código para preencher a matriz
    // e criar outros dados necessários para seu problema

    //alocar memória  na GPU para copiar a matriz
    double* d_m;
    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_m, quant_mem ));

    //copiar a matriz para a GPU
    CUDA_SAFE_CALL( cudaMemcpy(m, d_m, quant_mem,
        cudaMemcpyDeviceToHost));

    //executar a fatoração na GPU
    alg_lu_gpu(d_m, dim_mat);

    //copiar o resultado da GPU para a CPU
    CUDA_SAFE_CALL( cudaMemcpy(m, d_m, quant_mem,
        cudaMemcpyDeviceToHost));

    //limpar a memória da GPU
    CUDA_SAFE_CALL(cudaFree(d_m));

    //adicionar código para usar
    //as matrizes L e U (contidas em m)
    //e os outros dados

    free(m);

    CUDA_SAFE_CALL( cudaDeviceReset() );
    exit(EXIT_SUCCESS);
}
```

Código 3.11: Modelo de função `main()` para a fatoração LU de uma matriz usando a GPU.

Existem formas alternativas para implementar a fatoração LU que não diferem muito da que foi apresentada aqui. Iremos apresentar algumas delas como exercícios na seção 3.5.

A nossa implementação da fatoração LU é baseada em um algoritmo que não usa pivoteamento, mas é importante notar que a inclusão da estratégia de pivoteamento parcial nesse algoritmo pode ser feita de forma bastante direita [11].

3.4 Avaliação de desempenho

Nesta seção, apresentamos os resultados de diferentes testes realizados para avaliar os códigos apresentados anteriormente neste capítulo. Nosso primeiro objetivo é mostrar que a implementação de algoritmos paralelos apropriados na GPU pode contribuir a um aumento da eficiência computacional quando comparados com algoritmos sequenciais. Por conta disso, não discutiremos testes relacionados com a corretude e acurácia dos resultados numéricos obtidos com esses códigos.

Para avaliar o desempenho dessas implementações na GPU medimos os tempos de processamento e determinamos a aceleração em relação à implementações sequenciais para CPU desenvolvidas pelos autores (usando opções de otimização durante a compilação). Os resultados da avaliação devem ser interpretados com cuidado pois podem existir implementações sequenciais bem mais eficientes que a nossa. De qualquer forma, esses resultados fornecem informações interessantes sobre os fatores que limitam o desempenho dos códigos desenvolvidos.

Todos os testes foram realizados usando um computador Intel i7-4770, com o sistema operacional Ubuntu 14.04.3, uma placa NVIDIA GeForce GTX 770 e os compiladores: nvcc (versão 7.5) e gcc (versão 4.8.4).

Solução de sistemas tridiagonais

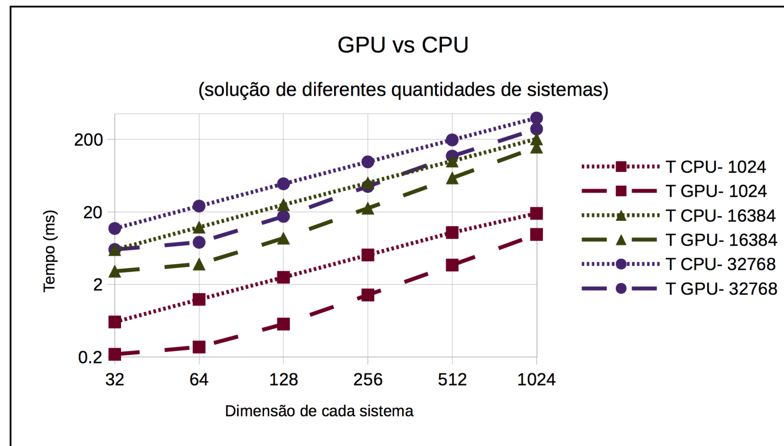
Lembramos que os códigos 3.2, 3.7 e as modificações propostas foram desenvolvidas com o objetivo de resolver vários sistemas tridiagonais simultaneamente na GPU.

Nos testes realizados, consideramos diferentes valores para a dimensão dos sistemas (n) e a quantidade deles a serem resolvidos (N_s). Para a dimensão, consideramos $n = 2^p$ com $p = 5, \ldots, 10$ e escolhemos $N_s = 32$, 512, 1024, 16384 e 32768. O valor máximo 1024 ($= 2^{10}$) para n é uma limitação da implementação.

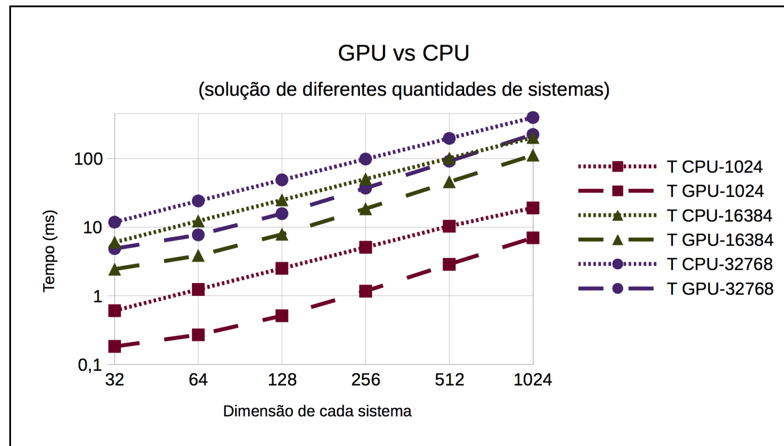
Para as diferentes escolhas de n e N_s foram executados vários testes. Em cada teste, primeiramente foram resolvidos N_s sistemas de dimensão n na CPU usando uma implementação do algoritmo de Thomas e o tempo de processamento foi medido. Em seguida, os mesmos N_s sistemas foram resolvidos usando as implementações para GPU dos métodos de redução cíclica e redução cíclica paralela (ou seja, incluindo as modificações propostas dos códigos 3.2 e 3.7) e os tempos de processamento correspondentes também foram medidos. Para cada escolha de n e N_s, foram realizados 5 testes para determinar os valores médios do tempo de processamento e das acelerações correspondentes a esses métodos. Esses resultados são mostrados nas figuras abaixo.

Destacamos que a medição dos tempos de processamento foi realizada utilizando as funções discutidas nas seções 1.5.1 e 2.3.1. Em cada medição apenas foi considerado o tempo associado ao processamento da matriz tridiagonal correspondente, sem incluir os tempos de inicialização ou transferência de dados. Por exemplo, no código 3.3 isto corresponder a medir apenas o tempo de execução do kernel (linha 45).

Na Figura 3.7 são mostrados os gráficos correspondentes ao tempo de resolução dos N_s sistemas na GPU e na CPU versus a dimensão dos sistemas (n). Na GPU foi usado o método de redução cíclica e na CPU o algoritmo de Thomas. Apresentamos os gráficos correspondentes a três dos valores de N_s testados. Observamos que as curvas correspondentes ao processamento na CPU e na GPU são bastante próximas de linhas retas, mas na GPU a inclinação é um pouco maior (eixos na escala logarítmica). De qualquer forma, para cada valor de N_s, as curvas correspondentes ao processamento na GPU sempre se encontram abaixo das curvas correspondentes



(a) GPU – método de redução cíclica e CPU – algoritmo de Thomas.

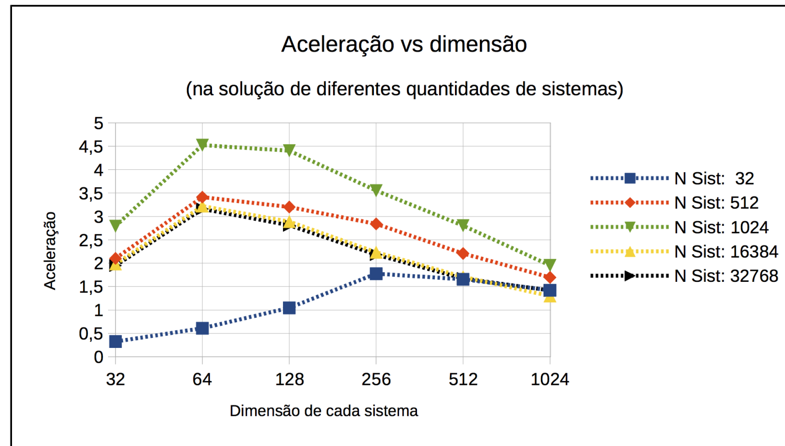


(b) GPU – método de redução cíclica paralela e CPU – algoritmo de Thomas.

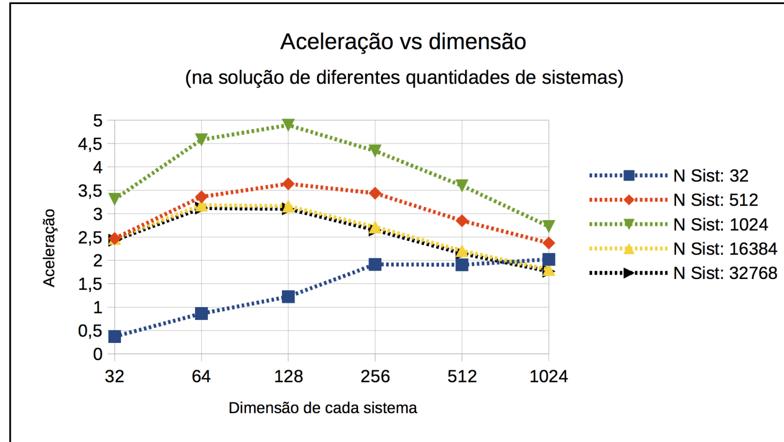
Figura 3.7: Tempo de processamento vs dimensão de cada sistema para diferentes valores da quantidade de sistemas (N_s). (Eixos na escala logarítmica.)

ao processamento na CPU. Comparando os gráficos a) e b), notamos que o tempo de processamento na GPU quando usamos o método de redução cíclica é maior do que quando usamos o método de redução cíclica paralela. Para os outros valores de N_s testados, apenas o caso $N_s = 32$ (que não foi mostrado na figura) teve um comportamento qualitativamente diferente. Nesse caso, os tempos de processamento na GPU para $n = 32$ e $n = 64$ foram maiores que os tempos de processamento na CPU.

A partir das medições de tempo realizadas podemos calcular a aceleração do processamento na GPU com relação ao processamento na CPU (veja a seção 1.5). Os resultados são mostrados na Figura 3.8. Com exceção da curva correspondente ao caso $N_s = 32$, o comportamento das outras curvas é qualitativamente o mesmo. Mais especificamente, em cada curva o valor da aceleração é sempre maior do que 1. Quando usamos o método de redução cíclica, o máximo é atingido em $n = 64$ (gráfico a)) e para o método de redução cíclica paralela em $n = 128$ (gráfico b)).



(a) GPU – método de redução cíclica e CPU – algoritmo de Thomas.



(b) GPU – método de redução cíclica paralela e CPU – algoritmo de Thomas.

Figura 3.8: Aceleração do processamento na GPU com relação ao processamento na CPU vs dimensão de cada sistema para diferentes valores da quantidade de sistemas (N_s). (Eixo das abscissas na escala logarítmica.)

No caso $N_s = 32$, o valor da aceleração é menor que 1 quando $n < 128$, indicando que na resolução de uma quantidade relativamente pequena de sistemas de baixa dimensão na GPU, o tempo associado ao gerenciamento das threads produz um aumento no tempo total de processamento que não é compensado pela execução paralela das threads.

Além disso, observamos que as curvas associadas com os dois maiores valores de N_s testados são quase iguais. Isto pode ser uma indicação de que a aceleração do processamento na GPU em relação ao processamento na CPU não varia muito quando a quantidade de sistemas é suficientemente grande.

O caso $N_s = 1024$ apresenta os melhores resultados. Isto está relacionado com o fato da GPU usada possuir 1536 núcleos. Como esses núcleos são responsáveis pela execução de $N_s \cdot n/2$ threads ($N_s \cdot n$, no caso da redução cíclica paralela), a grosso modo, temos que se n está fixo a capacidade de processamento será usada

ao máximo quando N_s estiver muito próximo de 1536.

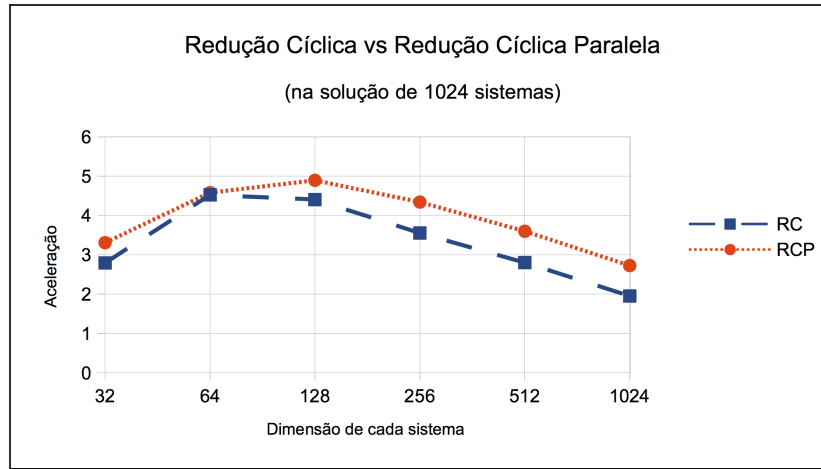


Figura 3.9: Comparação da aceleração do processamento na GPU (métodos de redução cíclica e redução cíclica paralela) em relação ao processamento na CPU usando o algoritmo de Thomas. (Eixo das abscissas na escala logarítmica.)

Finalmente, na Figura 3.9 apresentamos uma comparação das curvas da Figura 3.8 a) e b) correspondentes a $N_s = 1024$. Observamos que a curva correspondente ao método de redução cíclica paralela sempre está acima da curva associada ao método de redução cíclica. Isso é esperado pois o método de redução cíclica paralela possui um grau de paralelismo maior que o método de redução cíclica [14]. O melhor desempenho do método de redução cíclica paralela ocorre quando a dimensão do sistema $n = 128$. Notamos que quando o método de redução cíclica apresenta seu melhor desempenho, ou seja, para a dimensão do sistema $n = 64$, o método de redução cíclica paralela o supera apenas por uma margem muito estreita.

Para explicar o comportamento observado devemos levar em conta o grau de ocupação dos recursos da GPU (veja a seção 2.3.6). O grau de ocupação depende do número de threads por bloco e da quantidade de memória usada por cada bloco. Para valores pequenos da dimensão precisamos de poucas threads por bloco e também de pouca memória compartilhada, por tanto o grau de ocupação será baixo. Dessa forma, quando a dimensão cresce inicialmente o grau de ocupação também cresce. Porém, em algum momento a quantidade de memória compartilhada necessária será muito grande e consequentemente o grau de ocupação começará a diminuir.

Fatoração *LU*

Para avaliar o desempenho dos códigos 3.9, 3.10 e 3.11, usamos uma implementação sequencial do algoritmo para CPU. O código sequencial foi compilado com as opções de otimização do gcc (O0, O1, O2 e O3) e observamos que a opção *O3* gerou o programa mais eficiente.

Nos testes foram considerados os seguintes valores para a dimensão da matriz: $n = 64$, 128, 256, 512, 1024, 2048 e 4096. Também foram considerados dois tamanhos diferentes para os blocos da matriz associados com a grade de blocos de threads, $TB = 16$ e 32.

Para cada valor de n e TB rodamos 5 testes para determinar os valores médios do tempo de processamento (na GPU e na CPU) e da aceleração.

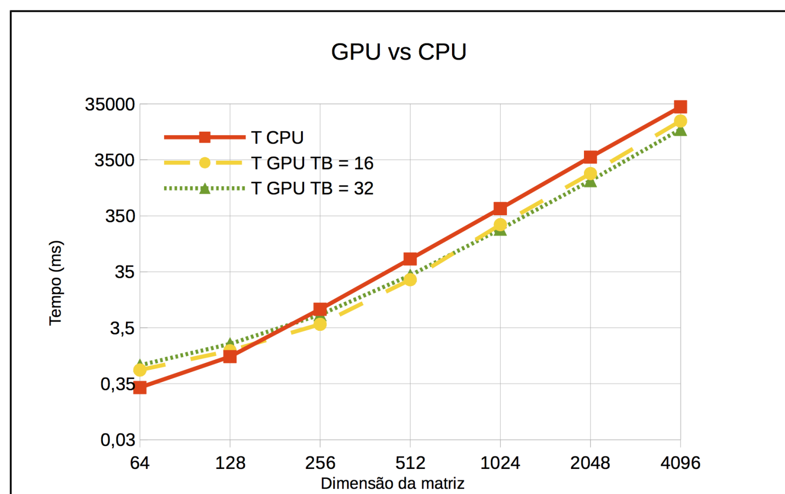


Figura 3.10: Tempo médio vs dimensão da matriz, para o processamento da fatoração da matriz na GPU e na CPU considerando o tamanho dos blocos $TB = 16$ e 32. (Eixos na escala logarítmica.)

Para cada escolha de n e TB, a fatoração da matriz foi realizada 5 vezes na CPU e também 5 vezes na GPU, e em seguida foi calculada a média do tempo de processamento na CPU e na GPU. O tempo medido em cada teste engloba o tempo de processamento associado com a fatoração da matriz e o tempo de transferência do resultado. Em geral, a transferência de dados da GPU para a CPU é mais lenta que a transferência de dados dentro da CPU o que leva a uma queda no desempenho da GPU.

Na Figura 3.10 apresentamos os resultados do tempo de processamento médio na GPU e na CPU para vários valores de n, considerando o tamanho dos blocos $TB = 16$ e 32. Notamos que as duas curvas correspondentes ao processamento na GPU estão relativamente próximas uma da outra e se cruzam para um valor de n entre 512 e 1024. Também observamos que para $n \geq 256$ ambas se encontram abaixo da curva correspondente ao processamento na CPU.

Na Figura 3.11 apresentamos as curvas da aceleração do processamento na GPU em relação ao processamento na CPU para os tamanhos dos blocos $TB = 16$ e 32. Observamos que para valores de $n < 1024$ o processamento usando blocos de tamanho $TB = 16$ apresenta um melhor desempenho e para $n \geq 1024$ o uso de blocos de tamanho $TB = 32$ acelera mais o processamento na GPU.

3.5 Exercícios

1. Faça modificações nos códigos 3.3 e 3.7 para desenvolver um programa em CUDA que determine a solução de vários sistemas tridiagonais pelo método de redução cíclica paralela.

2. Modifique o método de redução cíclica para considerar o caso de um sistema tridiagonal de dimensão n, em que n não é necessariamente uma potência de

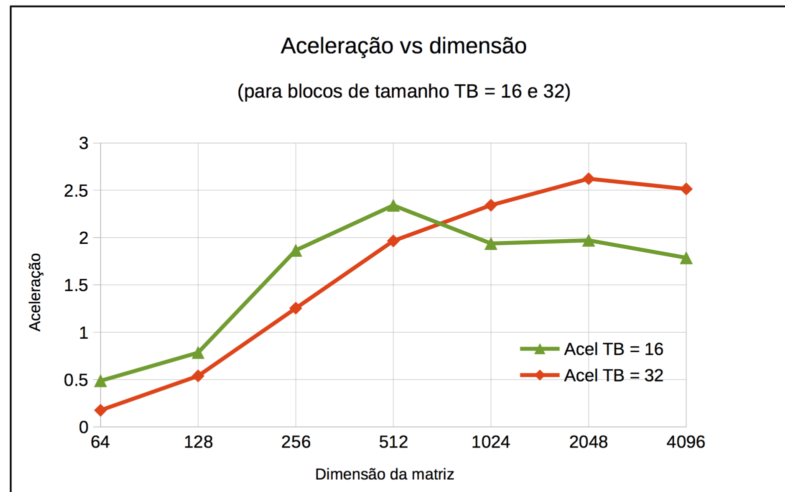


Figura 3.11: Aceleração do processamento na GPU em relação à CPU vs dimensão da matriz quando os blocos são de tamanho $TB = 16$ e 32. (Eixo das abscissas na escala logarítmica.)

2. Desenvolva um programa em CUDA para implementar esse método.

3. Desenvolva um programa em CUDA para implementar a solução de N_s sistemas de equações tridiagonais de dimensão n usando o algoritmo de Thomas.

4. Desenvolva um programa em CUDA para implementar a fatoração LU na GPU. Considere que na etapa de atualização da submatriz principal a relação entre os blocos de threads e os blocos da matriz está de acordo com o esquema da Figura 3.6, porém cada thread é responsável pela atualização dos elementos de:

   a) uma única linha do bloco da matriz associado ao bloco de threads;
   b) uma única coluna do bloco da matriz associado ao bloco de threads.

5. Desenvolva um programa em CUDA que implemente a solução de um sistema de equações lineares em que a matriz dos coeficientes é:

   a) triangular superior;
   b) triangular inferior com os elementos da diagonal principal todos iguais a 1.

   Para determinar a solução do sistema use um algoritmo de substituição baseado na representação da matriz inicial em blocos. Considere que a dimensão da matriz $n = b \cdot N$ em que b é a dimensão de cada bloco e $N \geq 1$ é um inteiro [11]. Escolha blocos de threads associados com os blocos da matriz.

Capítulo 4

Solução das equações de Maxwell usando GPUs

4.1 Introdução

As ondas eletromagnéticas e suas interações com o entorno possuem diversas aplicações, como por exemplo nas tecnologias de comunicação sem fio e em alguns tipos de tratamentos e diagnósticos usados na área médica [13, 36, 43]. O desenvolvimento e aprimoramento dessas aplicações requerem um estudo detalhado da interação do campo eletromagnético e a propagação de ondas eletromagnéticas na região de interesse. Esse estudo toma como base as equações de Maxwell, um sistema de equações em derivadas parciais que descreve a evolução das ondas eletromagnéticas.

Resolver essas equações de forma analítica torna-se extremamente complexo e às vezes é simplesmente impossível, por isso métodos numéricos são normalmente usados para se encontrar soluções aproximadas [16, 36].

Apesar da existência de métodos numéricos simples para a resolução dessas equações, como por exemplo o esquema de Yee [36, 42], as restrições na discretização temporal tornam a implementação do método computacionalmente desafiadora quando precisamos modelar períodos longos de tempo.

O presente capítulo é dedicado ao estudo de diferentes estratégias de paralelização do esquema de Yee para execução em GPUs, considerando o caso bidimensional das equações de Maxwell.

4.2 Método de diferenças finitas para as equações de Maxwell

As equações de Maxwell descrevem a evolução no tempo do campo eletromagnético em uma região do espaço. Na sua forma diferencial, elas são dadas pelo sistema de equações diferenciais parciais [36]:

$$\begin{aligned} \frac{\partial \vec{H}}{\partial t} &= -\frac{1}{\mu}\nabla \times \vec{E} - \frac{1}{\mu}(\vec{M}_{fonte} + \sigma^* \vec{H}) \\ \frac{\partial \vec{E}}{\partial t} &= \frac{1}{\epsilon}\nabla \times \vec{H} - \frac{1}{\epsilon}(\vec{J}_{fonte} + \sigma \vec{E}) \end{aligned} \tag{4.2.1}$$

onde $\vec{H}$ e $\vec{E}$ representam os campos magnético e elétrico, respectivamente; $\vec{M}_{fonte}$ e $\vec{J}_{fonte}$ representam as fontes magnéticas e elétricas, respectivamente; μ é a permeabilidade magnética, ϵ a permissividade elétrica, σ a condutividade elétrica e σ^* a perda magnética equivalente; $\nabla\times$ representa o operador rotacional.

Para obter as soluções desse sistema é preciso adicionar as condições iniciais e de contorno. Em regiões limitadas, a condição de contorno especifica a componente tangencial de um dos campos na fronteira [16, 36]. O caso mais comum é o *condutor perfeito*, em que exigimos que a componente tangencial do campo elétrico seja nula ao longo da fronteira. Quando a região não é limitada são usadas as condições de radiação de Sommerfeld [16, 36].

Quando consideramos que em relação a um sistema de coordenadas cartesianas fixado, o campo eletromagnético não depende da coordenada z, o sistema (4.2.1) pode ser reescrito em uma forma mais simplificada. O modo transversal elétrico TE_z (*transverse-electric mode*) é dado pelo seguinte sistema de equações diferenciais parciais para as componentes E_x, E_y e H_z do campo eletromagnético

$$\begin{aligned}
\frac{\partial H_z}{\partial t} &= \frac{1}{\mu}\left(\frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - (M_z + \sigma^* H_z)\right),\\
\frac{\partial E_x}{\partial t} &= \frac{1}{\varepsilon}\left(\frac{\partial H_z}{\partial y} - (J_x + \sigma E_x)\right),\\
\frac{\partial E_y}{\partial t} &= \frac{1}{\varepsilon}\left(-\frac{\partial H_z}{\partial x} - (J_y + \sigma E_y)\right).
\end{aligned} \tag{4.2.2}$$

As demais componentes do campo eletromagnético satisfazem a um sistema semelhante ao (4.2.2) que é conhecido como modo transversal magnético TM_z (*transverse-magnetic mode*).

Esses sistemas de equações diferenciais parciais são desacoplados. Eles descrevem a evolução de ondas eletromagnéticas que se propagam paralelamente ao eixo z, por isso devem ser resolvidos considerando uma região do plano cartesiano com coordenadas (x, y), introduzindo as condições iniciais e de contorno correspondentes.

4.2.1 Esquema de Yee

O método de diferenças finitas no espaço e tempo (FDTD) é muito popular para a resolução das equações de Maxwell. Este método foi introduzido por Kane Yee em 1966 [42] e ficou conhecido como esquema de Yee [36].

A ideia principal do esquema de Yee consiste no uso de duas malhas intercaladas para a aproximação das diferentes componentes: E_x, E_y e H_z. Dessa forma é possível aproximar as derivadas espaciais envolvidas usando diferenças finitas centradas. Isso permite introduzir uma discretização no tempo semelhante ao esquema *leapfrog*, e finalmente obter um esquema de segunda ordem de acurácia no tempo e no espaço [36].

Na discretização das equações, introduzimos a notação

$$u(n\Delta t, i\Delta x, j\Delta y) = u|_{i;j}^{n}, \qquad n, i, j \in \tfrac{1}{2}\mathbb{Z}, \tag{4.2.3}$$

onde u representa as distintas componentes E_x, E_y ou H_z; Δx, Δy e Δt representam o espaçamento na discretização espacial e o tamanho do passo na discretização do tempo, respectivamente. Observe que n, i e j podem assumir valores inteiros ou fracionários, nesse último caso eles estão associados com pontos numa malha intercalada.

Assim, vamos ter as seguintes aproximações de segunda ordem para as derivadas

$$u_t|_{i;j}^n \approx \frac{u|_{i;j}^{n+1/2} - u|_{i;j}^{n-1/2}}{\Delta t}, \tag{4.2.4}$$

$$u_x|_{i;j}^n \approx \frac{u|_{i+1/2;j}^n - u_t|_{i-1/2;j}^n}{\Delta x}, \tag{4.2.5}$$

$$u_y|_{i;j}^n \approx \frac{u|_{i;j+1/2}^n - u_t|_{i;j-1/2}^n}{\Delta y}. \tag{4.2.6}$$

Na Figura 4.1, apresentamos as malhas usadas na discretização das distintas componentes. Observamos que cada componente elétrica é rodeada pelas componentes magnéticas. Além disso, as componentes magnéticas são aproximadas nos tempos de índice inteiro e as componentes elétricas nos tempos de índices intermediários. Após fazermos essas substituições em (4.2.2) chegamos em um sistema explícito de fácil resolução (4.2.7).

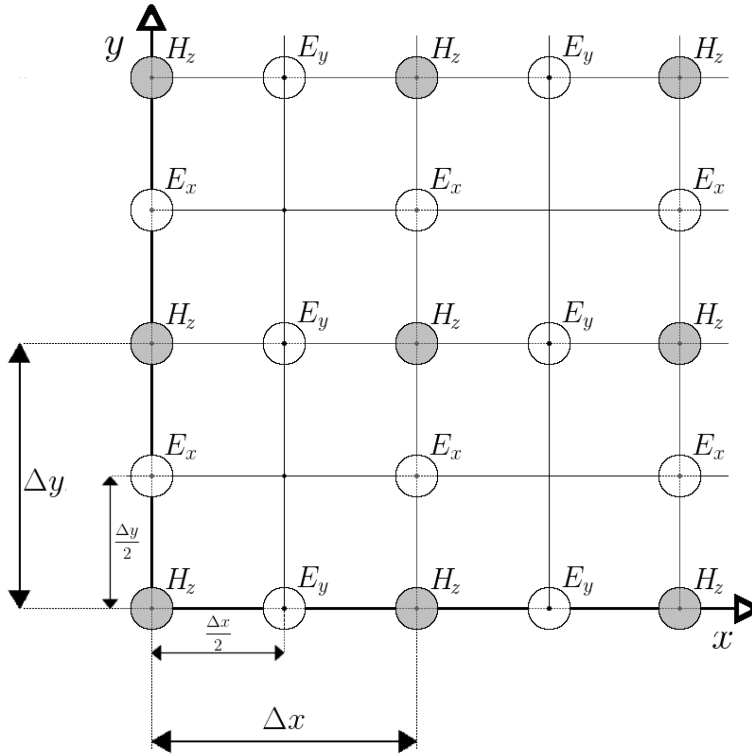


Figura 4.1: Malha da discretização espacial associada com o esquema de Yee.

O sistema de equações discretizadas tem a forma

$$H_z|_{i;j}^{n+1} = aH_z|_{i;j}^{n} + b\left\{\frac{E_x|_{i;j+\frac{1}{2}}^{n+\frac{1}{2}} - E_x|_{i;j-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta y} - \frac{E_y|_{i+\frac{1}{2};j}^{n+\frac{1}{2}} - E_y|_{i-\frac{1}{2};j}^{n+\frac{1}{2}}}{\Delta x} - M_z|_{i;j}^{n+\frac{1}{2}}\right\} \tag{4.2.7}$$

$$E_x|_{i;j+\frac{1}{2}}^{n+\frac{3}{2}} = cE_x|_{i;j+\frac{1}{2}}^{n+\frac{1}{2}} + d\left\{\frac{H_z|_{i;j+1}^{n+1} - H_z|_{i;j}^{n+1}}{\Delta y} - J_x|_{i;j+\frac{1}{2}}^{n+1}\right\} \tag{4.2.8}$$

$$E_y|_{i+\frac{1}{2};j}^{n+\frac{3}{2}} = cE_y|_{i+\frac{1}{2};j}^{n+\frac{1}{2}} + d\left\{-\frac{H_z|_{i+1;j}^{n+1} - H_z|_{i;j}^{n+1}}{\Delta x} - J_y|_{i+\frac{1}{2};j}^{n+1}\right\} \tag{4.2.9}$$

onde

$$a = \frac{1 - \frac{\sigma^*\Delta t}{2\mu}}{1 + \frac{\sigma^*\Delta t}{2\mu}}, \qquad b = \frac{\frac{\Delta t}{\mu}}{1 + \frac{\sigma^*\Delta t}{2\mu}}, \qquad c = \frac{1 - \frac{\sigma\Delta t}{2\epsilon}}{1 + \frac{\sigma\Delta t}{2\epsilon}}, \qquad d = \frac{\frac{\Delta t}{\epsilon}}{1 + \frac{\sigma\Delta t}{2\epsilon}} \tag{4.2.10}$$

são constantes associadas com as propriedades do meio.

Uma representação gráfica do processo de evolução no tempo para o esquema de Yee é mostrada na Figura 4.2. Nessa figura também são representadas as dependências entre os diferentes dados.

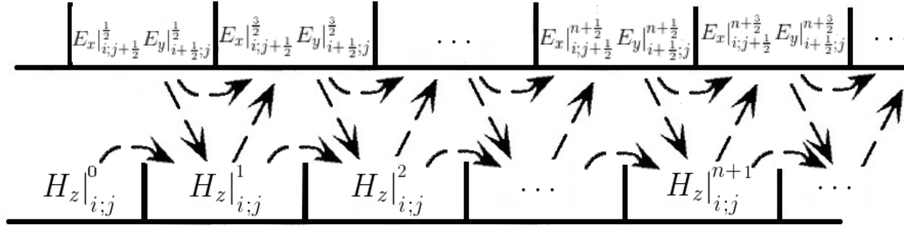


Figura 4.2: Representação gráfica do processo de cálculo da evolução no tempo para o esquema de Yee, mostrando a dependência entre os diferentes dados.

Para garantir a estabilidade numérica das soluções, o tamanho do passo do tempo deve satisfazer a condição de estabilidade de Courant-Friedrichs-Lewy (CFL) [36] dada por

$$\Delta t \leq \frac{1}{\upsilon\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}}},$$

onde υ representa a velocidade de propagação das ondas eletromagnéticas no meio dada por $\upsilon = 1/\sqrt{\epsilon\mu}$.

As condições de contorno usadas na modelagem de um meio limitado são as seguintes: na parte da fronteira em que $y = j\Delta y = const$, o valor de H_z é especificado nos pontos $(i\Delta x, j\Delta y)$; na parte em que $y = (j + \frac{1}{2})\Delta y = const$, o valor de E_x é especificado nos pontos $(i\Delta x, (j+\frac{1}{2})\Delta y)$; na parte em que $x = i\Delta x = const$, o valor de H_z é especificado nos pontos $(i\Delta x, j\Delta y)$; e na parte em que $x = (i+\frac{1}{2})\Delta x = const$,

o valor de E_y é especificado nos pontos $((i+\frac{1}{2})\Delta x, j\Delta y)$. No restante desse texto, chamaremos essas condições de *condições de contorno tipo Dirichlet.*

Na modelagem de um meio não limitado temos que usar um domínio computacional finito e impor condições artificiais na fronteira. A forma mais comum de fazer isso é considerando condições de contorno absorventes nessa fronteira artificial. Uma das formas mais bem sucedidas de introduzir condições de contorno absorventes no método FDTD foi feita por Bérenger em 1994 [4], sua proposta é chamada de PML (*Perfectly Matched Layer*).

Nessa abordagem, são introduzidas camadas extras de células (chamadas de PML) rodeando o domínio de interesse. Em cada célula da PML, o campo magnético é dividido em duas partes (H_{zx}, H_{zy}). A condutividade elétrica é determinada de forma que as ondas eletromagnéticas penetrem sem reflexão na interface entre o meio de estudo e a PML, para qualquer frequência de onda e ângulo de incidência [4, 36].

4.2.2 Algoritmo do esquema de Yee

O fluxograma apresentado na Figura 4.3 descreve o algoritmo correspondente ao esquema de Yee.

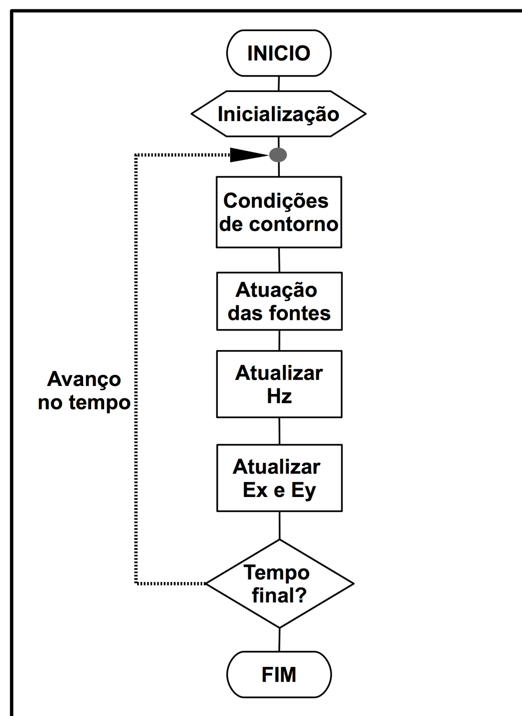


Figura 4.3: Fluxograma correspondente ao esquema de Yee.

A partir dele podem ser desenvolvidos diferentes códigos de acordo com as condições de contorno sendo utilizadas. Na inicialização são definidas as propriedades do meio e a discretização do domínio em malhas. No caso em que usamos PML também é necessário definir a camada extra de células (PML) e fazer o cálculo de suas propriedades. Logo, a cada passo do tempo, o algoritmo determina as condições de contorno e os valores das fontes e em seguida atualiza os valores

do campo H_z e os campos E_x e E_y (nessa ordem). No caso em que temos PML, o cálculo do campo H_z é dividido em duas etapas.

4.3 Implementação paralela do esquema de Yee em GPU

Como discutido no início, apesar de simples, o método de Yee é muito dispendioso computacionalmente devido às restrições na discretização temporal necessárias para garantir a estabilidade numérica. Nesta seção, apresentamos uma proposta de paralelização do método de Yee (em sua forma bidimensional) para execução em GPU. Tomamos como base o algoritmo proposto em [7, 37].

4.3.1 Aspectos gerais da implementação para GPU

A ideia básica da implementação paralela para GPU é realizar o controle das iterações no tempo na CPU e lançar em cada iteração vários kernels para realizar as diferentes tarefas: calcular as condições de contorno, determinar a influência das fontes, atualizar o campo H_z e, por último, atualizar os campos E_x e E_z.

Dependendo do problema, pode ser mais eficiente fazer os cálculos das condições de contorno e da influência das fontes na própria CPU. A atualização dos campos será sempre uma tarefa computacionalmente intensa em que o paralelismo da GPU deve ser aproveitado.

Internamente, a implementação dos kernels faz com que cada thread seja responsável pelo cálculo em uma posição da malha espacial. Os blocos da grade são escolhidos para cobrir essa malha. A dimensão dos blocos de threads é um parâmetro que pode ser usado para aumentar a eficiência da implementação.

4.3.2 Implementação paralela em CUDA

Os códigos que apresentamos a seguir correspondem à solução das equações de Maxwell para a propagação de um pulso magnético gerado em um único ponto do domínio a partir de um estado inicial de repouso. Consideramos que a fronteira é um condutor perfeito. Para esse problema é preciso implementar somente os kernels que atualizam os vetores dos campos.

Em nossa implementação, os valores de H_z, E_x e E_y correspondentes aos diferentes pontos da malha são armazenados como vetores de tipo `double`. Usamos precisão dupla para evitar que a propagação dos erros de arredondamento produzam resultados de baixa acurácia quando são modelados períodos longos de tempo. Destacamos que, em geral, o uso de precisão dupla provoca uma queda no desempenho da GPU pois as operações aritméticas em precisão simples são efetuadas com maior eficiência.

Explicamos a seguir, como é realizado o armazenamento dos vetores. Observe que nas equações discretizadas (4.2.7), as componentes do campo estão associadas com índices espaciais da forma (i, j) $(i+1/2, j)$ e $(i, j+1/2)$ em que i, j são inteiros. Assim sendo, para efeito de ordenação dos dados, associamos também os índices $(i + 1/2, j)$ e $(i, j + 1/2)$ com o índice (i, j). Como nosso domínio computacional é finito, então os índices (i, j) da malha podem ser restringidos ao conjunto $0 \leq i < N_x$ e $0 \leq j < N_y$ onde N_x, N_y são dois inteiros positivos que definem as dimensões da malha. Portanto, os $u_{i,j}$ associados com os pontos (i, j) da malha serão ordenados

de acordo com o novo índice $I = i \cdot N_y + j$. Em outras palavras, no armazenamento dos vetores a ordem usada é de baixo para cima e da esquerda para a direita

Importante observar que para o problema que está sendo implementado, a malha para as componentes E_x e E_y tem uma coluna e uma linha a menos que a malha para H_z, respectivamente.

Kernels para atualização dos campos magnético e elétrico Vamos começar apresentando o código do kernel `atualiza_H()` para a atualização dos vetores do campo H_z. Essa função recebe como entradas os vetores de dados dos campos `d_Ex`, `d_Ey` e `d_Hz`; os coeficientes constantes `c_a`, `c_b` (veja (4.2.10)); as dimensões da malha e sua discretização `nx`, `ny` e `dx`, `dy`; os índices da posição da fonte pontual na malha `i_f`, `j_f` e sua intensidade `val_f`.

Nas linhas 2 e 3, associamos o identificador da thread com o índice da malha. Temos que tomar cuidado para desativar as threads que estão associadas com pontos "fantasmas" da malha (linha 7). A partir daí, fazemos o cálculo de (4.2.7) levando em conta que nas bordas do domínio a componente tangencial do campo elétrico é nula. Quando o ponto não está na borda devemos considerar a contribuição do ponto vizinho correspondente no cálculo das diferenças finitas (linhas 9–21). Na linha 25 é feita a atualização usando (4.2.7). Por último na linha 29 é feita a correção do valor de H_z no ponto onde se encontra a fonte.

```
1  __global__ void  atualiza_H(double *d_Ex, double *d_Ey, double
       *d_Hz, double c_a, double c_b, int nx, int ny, double  dx,
       double  dy, int i_f, int j_f, double val_f ) {
2     int i=blockIdx.x*blockDim.x+threadIdx.x;
3     int j=blockIdx.y*blockDim.y+threadIdx.y;

5     double DyEx = 0, DxEy = 0;

7     if (i < nx && j < ny) {
8        //não está na borda esquerda
9        if (i > 0) {
10          DxEy -= d_Ey[ (i-1)*ny + j ];
11       }
12       //não está na borda direita
13       if (i <  nx-1) {
14          DxEy += d_Ey[ i*ny + j ];
15       }
16       //não está na borda inferior
17       if (j > 0) {
18          DyEx -= d_Ex[ i*(ny-1) + j-1 ];
19       }
20       //não está na borda superior
21       if (j <  ny-1) {
22          DyEx += d_Ex[ i*(ny-1) + j ];
23       }

25       d_Hz[i*ny+j] = c_a*d_Hz[i*ny+j] + c_b*( DyEx/dy - DxEy/dx
              );

27       //influência da fonte
28       if (i == i_f && j == j_f) {
29          d_Hz[i*ny+j] = val_f;
30       }
```

```
31      }
32  }
```

Código 4.1: Kernel para a atualização da componente H_z do campo magnético.

A implementação do kernel `atualiza_E()` é apresentada no código 4.2. Nessa função são necessários menos dados de entrada que no kernel anterior já que não temos fontes associadas ao campo elétrico nesse problema. Porém, no lugar das constantes `c_a` e `c_b`, agora aparecem as constantes `c_c` e `c_d` (veja (4.2.10)). A implementação das equações (4.2.8) e (4.2.9) é bastante direta. O único aspecto diferente é que as dimensões das malhas de E_x e E_y são diferentes das dimensões da malha de E_z. Como sempre, devemos tomar cuidado com as threads associadas com pontos "fantasmas" da malha.

```
1   __global__ void  atualiza_E(double *d_Ex, double *d_Ey, double
      *d_Hz, double c_c, double c_d, int nx, int ny, double  dx,
      double  dy) {
2      int i=blockIdx.x*blockDim.x+threadIdx.x;
3      int j=blockIdx.y*blockDim.y+threadIdx.y;

5      double DyHz, DxHz;

7      if (i < nx && j < ny) {
8         DxHz= -d_Hz[i*ny+j];
9         DyHz = DxHz;

11        //não tem Ex acima da borda superior
12        if (j < ny-1)  {
13           DyHz += d_Hz[ i*ny + (j+1) ];
14           d_Ex[i*(ny-1)+j] = c_c*d_Ex[i*(ny-1)+j] + c_d*DyHz/dy;
15        }

17        //não tem Ey à direita da borda direita
18        if (i < nx-1)  {
19           DxHz += d_Hz[ (i+1)*ny + j ];
20           d_Ey[i*ny+j] = c_c*d_Ey[ i*ny+j ] - c_d*DxHz/dx;
21        }
22     }
23  }
```

Código 4.2: Kernel para a atualização das componentes E_x e E_y do campo elétrico.

Função principal Apresentamos a seguir um modelo para a função principal de um programa para resolver as equações de Maxwell, usando os kernels apresentados acima. Para o programa funcionar corretamente, é preciso adicionar as linhas de código com a descrição das propriedades do meio (linhas 9–11), a geometria do problema, o tempo de simulação e os parâmetros da discretização (linhas 17–21). Adicionalmente, podem ser introduzidas linhas de códigos para considerar outras condições iniciais (linhas 55 e 56) e salvar os resultados no disco ou na memória para o pós-processamento (linha 84).

```
1   int main() {
2      //parâmetros dos blocos de threads
3      int TAM_B_x, TAM_B_y;
4      TAM_B_x = 32;
```

```
5     TAM_B_y = 32;

7     double eps=1, mu=1, sigma1=0, sigma2=0;

9     //adicionar  código para inicializar as variáveis
10    //para as propriedades do meio
11    //eps, mu, sigma1, sigma2

13    double x_ini,x_final,y_ini,y_final,t_ini,t_final;
14    double x_fonte, y_fonte;
15    int nx, ny, n_passos;

17    //adicionar  código para inicializar as variáveis
18    //para a geometria do problema, a posição da fonte e
19    //o tempo de simulação, e os parâmetros da discretização:
20    //x_ini,x_final,y_ini,y_final,t_ini,t_final,
21    //x_fonte, y_fonte, nx, ny, n_passos

23    double dx, dy, dt;
24    double c_a, c_b, c_c, c_d;

26    dx = (x_final-x_ini)/nx;
27    dy = (y_final-y_ini)/ny;
28    dt = (t_final-t_ini)/n_passos;

30    //cálculo dos coeficientes constantes
31    c_a = (1-0.5*sigma2*dt/mu)/(1+0.5*sigma2*dt/mu);
32    c_b = (dt/mu)/(1+0.5*sigma2*dt/mu);
33    c_c = (1-0.5*sigma1*dt/eps)/(1+0.5*sigma1*dt/eps);
34    c_d = (dt/eps)/(1+0.5*sigma1*dt/eps);

36    //índice da posição da fonte na malha
37    int i_f, j_f;
38    i_f  = (int)((x_fonte-x_ini)/dx);
39    j_f  = (int)((y_fonte-y_ini)/dy);

41    //vetores para inicializar e salvar os resultados
42    double *Hz, *Ex, *Ey;
43    size_t dim_Hz = nx*ny, dim_Ex=nx*(ny-1), dim_Ey=(nx-1)*ny;

45    //vetores inicializados com zeros
46    Hz = (double *) calloc(dim_Hz,sizeof(double));
47    Ex = (double *) calloc(dim_Ex,sizeof(double));
48    Ey = (double *) calloc(dim_Ey,sizeof(double));

50    if (Hz == NULL || Ex == NULL || Ey == NULL) {
51       fprintf(stderr, "Memoria insuficiente\n");
52       exit(EXIT_FAILURE);
53    }

55    //adicionar  código para preencher os vetores
56    //Hz, Ex, Ey com as condições iniciais

58    //alocar memória  na GPU para os vetores dos campos
59    double* d_Hz, *d_Ex, *d_Ey;
```

```
60    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_Hz, dim_Hz*sizeof(
         double) ));
61    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_Ex, dim_Ex*sizeof(
         double) ));
62    CUDA_SAFE_CALL( cudaMalloc( (void**) &d_Ey, dim_Ey*sizeof(
         double) ));

64    //transferir os dados para a GPU
65    CUDA_SAFE_CALL( cudaMemcpy(d_Hz, Hz, dim_Hz*sizeof(double),
         cudaMemcpyHostToDevice));
66    CUDA_SAFE_CALL( cudaMemcpy(d_Ex, Ex, dim_Ex*sizeof(double),
         cudaMemcpyHostToDevice));
67    CUDA_SAFE_CALL( cudaMemcpy(d_Ey, Ey, dim_Ey*sizeof(double),
         cudaMemcpyHostToDevice));

69    //definir a grade de blocos
70    int g_x= (nx + TAM_B_x-1)/TAM_B_x;
71    int g_y= (ny + TAM_B_y-1)/TAM_B_y;
72    dim3 dim_bloco(TAM_B_x,TAM_B_y);
73    dim3 dim_grade(g_x,g_y);

75    //laço para iteração no tempo
76    for (i_t=0; i_t<n_passos; i_t++ ) {
77       //adicionar o código que calcula a intensidade da fonte
78       //no ponto
79       //deve ser uma função de (i_t+1)*dt
80       double val_f = ... ;

82       atualiza_H<<< dim_grade,dim_bloco>>>(d_Ex,d_Ey,d_Hz,c_a,
            c_b,nx,ny,dx,dy,i_f,j_f,val_f );
83       atualiza_E<<< dim_grade,dim_bloco>>>(d_Ex,d_Ey,d_Hz,c_c,
            c_d,nx,ny,dx,dy);

85       //adicionar código para usar os resultados
86    }

88    //liberar a memória da GPU e da CPU
89    CUDA_SAFE_CALL(cudaFree(d_Hz));
90    CUDA_SAFE_CALL(cudaFree(d_Ex));
91    CUDA_SAFE_CALL(cudaFree(d_Ey));
92    free(Hz);
93    free(Ex);
94    free(Ey);

96    CUDA_SAFE_CALL(cudaDeviceReset());
97    exit(EXIT_SUCCESS);
98 }
```

Código 4.3: Modelo de programa para resolver as equações de Maxwell usando a GPU.

4.3.3 Exemplo de aplicação do programa

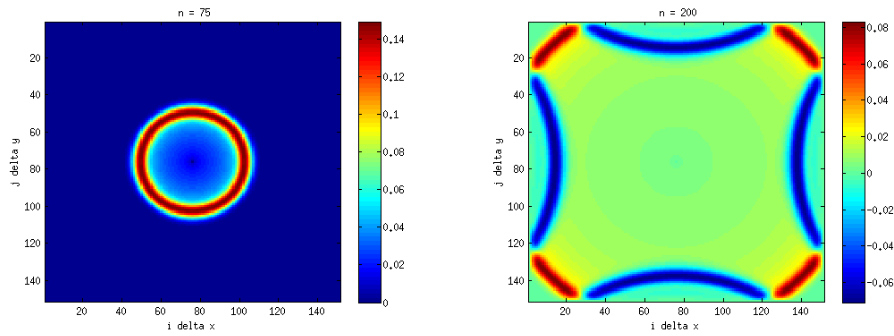
Nesta seção, mostramos alguns resultados de simulações obtidos usando o programa descrito anteriormente. Simulamos numericamente a propagação de uma onda ge-

rada por um pulso pontual. Consideramos que a propagação acontece no vácuo, ou seja $\epsilon = 8.854 \times 10^{-12}$ e $\mu = 4\pi \times 10^{-7}$ e portanto $\upsilon = (\epsilon\mu)^{-1/2} \approx 3 \times 10^8$.

O domínio foi discretizado com uma malha formada por 150×150 células quadradas de dimensão $\Delta x = \Delta y = \Delta = 0.01$ e para o tamanho do passo de tempo escolhemos $\Delta t = \frac{\Delta}{2\upsilon} \approx 1.7 \times 10^{-11}$. No ponto médio do domínio atua uma fonte pontual com intensidade:

$$H_z|^n = \begin{cases} \frac{1}{32}\left\{10 - 15\cos(n\pi/20) + 6\cos(2n\pi/20) - \cos(3n\pi/20)\right\}, & n \leq 40, \\ 0, & n > 40. \end{cases} \tag{4.3.11}$$

Os gráficos da Figura 4.4 mostram a propagação do pulso para algumas variações de passos no tempo. Assim, é possível observar o seu comportamento quando a energia do campo alcança as fronteiras.



(a) Após 75 passos de tempo.

(b) Após 200 passos de tempo.

Figura 4.4: Propagação no vácuo do pulso pontual dado por (4.3.11).

Podemos verificar que a propagação da onda inicialmente ocorre de forma radial. No entanto, quando ela atinge a fronteira, ocorre uma reflexão total da onda devido ao uso das condições de contorno de Dirichlet (Figura 4.4).

Devido a esta reflexão, quando desejamos simular um meio não limitado devemos realizar um aumento da malha computacional, de forma que a onda não atinja a fronteira do domínio de interesse dentro do tempo de análise. Nessa situação, o mais apropriado é usar condições de contorno absorventes, como por exemplo PML.

Mais detalhes sobre este e outros exemplos podem ser encontrados em [37].

4.4 Avaliação de desempenho

Para avaliar os algoritmos propostos, usamos um computador Intel i7-960, com o sistema operacional Ubuntu 14.04, uma placa NVIDIA Tesla C2070 e os compiladores: nvcc (versão 5.5) e gcc (versão 4.8.2).

Usamos uma implementação do algoritmo preparada para considerar meios heterogêneos e fazer uso da memória compartilhada [38]. Variamos o tamanho da malha da discretização espacial de 300×300 até 6900×6900 (tamanho máximo limitado pelo espaço de memória global da GPU usada, considerando precisão dupla). Em todos os experimentos consideramos 1000 iterações no tempo.

Usamos as métricas de **tempo de execução** e **aceleração** *(speedup)* para avaliar esses experimentos. Acrescentamos ainda uma métrica mais específica para simulações de malhas que contabiliza o número de células da malha processadas por unidade de tempo [7]. Ela é denominada ***velocidade [MCells/s]*** *(speed)* e está definida pela equação

$$velocidade[MCells/s] = \frac{N_x \, N_y \, N_t}{10^6 T_e} \tag{4.4.12}$$

onde N_x e N_y são as dimensões da malha de nós utilizada para a discretização; N_t é o número de passos no tempo; e T_e é o tempo de execução da implementação medido em segundos.

4.4.1 Avaliação do algoritmo sequencial

O código sequencial foi implementado na linguagem C e compilado com as opções de otimização do *gcc* ($O0$, $O1$, $O2$ e $O3$), gerando quatro arquivos executáveis que foram experimentados separadamente.

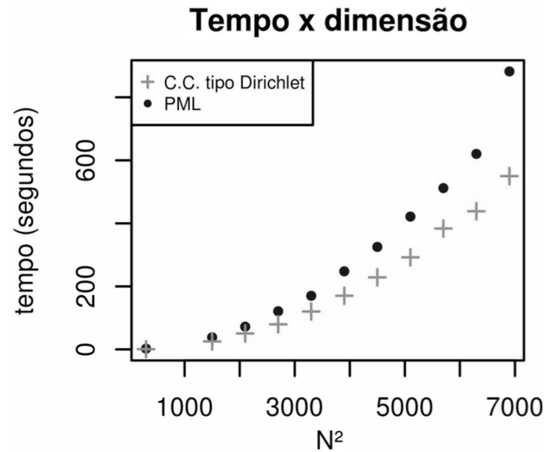


Figura 4.5: Desempenho dos algoritmos sequenciais. Tempo de processamento vs. dimensão da malha.

O gráfico mostrado na Figura 4.5 apresenta o desempenho das duas versões do algoritmo sequencial (com as condições de contorno tipo Dirichlet e a *PML*), ambas usando a otimização $O3$ do gcc que foi a implementação com o melhor desempenho. Observe que o algoritmo com PML teve um aumento significativo no tempo de processamento. Isso se deve à divisão do campo Hz em duas componentes e da necessidade de incluir camadas extras de células, que neste caso têm tamanho 25 em cada direção. O código com PML demora em média 43% a mais do que a versão com Dirichlet, valor similar ao aumento da memória dado pelo uso da PML.

Utilizaremos o menor tempo obtido com o algoritmo sequencial (otimização $O3$) como *benchmark* para o cálculo das demais métricas.

4.4.2 Avaliação do algoritmo paralelo

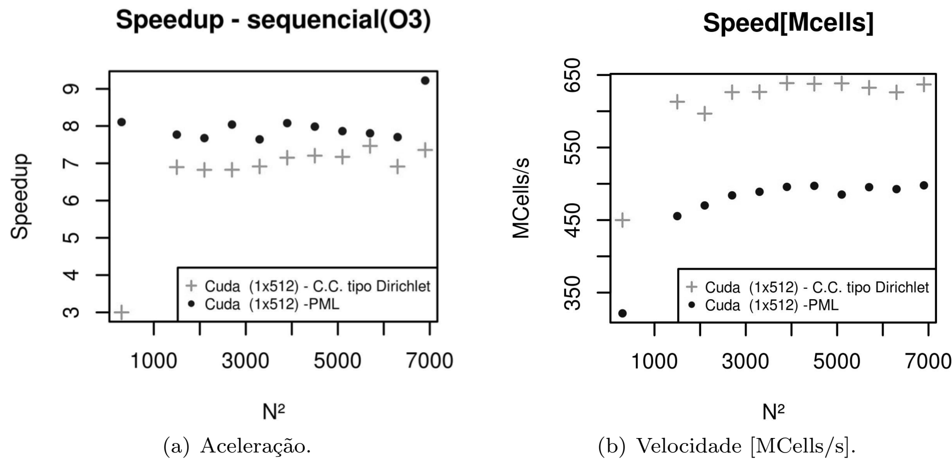
Na avaliação do algoritmo paralelo em CUDA, usamos como parâmetro de entrada a dimensão dos blocos de threads utilizados, uma vez que o desempenho do algoritmo

é afetado por esse valor.

	C.C. tipo Dirichlet				PML			
	1x128	1x256	1x512	1x1024	1x128	1x256	1x512	1x1024
300	0.23	**0.18**	0.20	0.28	0.41	**0.27**	0.28	0.38
1500	4.3	**3.7**	**3.7**	4.0	4.9	**4.8**	4.9	5.1
2100	7.1	**7.0**	7.4	8.2	9.3	**9.0**	9.4	10.3
2700	12.2	11.8	**11.6**	12.2	16.7	15.3	**15.1**	15.7
3300	18.5	**17.4**	**17.4**	18.5	23.6	22.6	**22.3**	23.7
3900	24.7	**23.8**	**23.8**	24.7	31.5	**30.4**	30.7	31.9
4500	33.2	31.8	**31.7**	33.7	43.0	41.0	**40.7**	43.5
5100	42.5	41.4	**40.7**	42.0	55.2	53.7	**53.6**	56.5
5700	53.3	51.8	**51.4**	53.1	69.2	66.7	**65.6**	68.4
6300	65.8	63.8	**63.4**	66.3	83.4	80.8	**80.6**	85.2
6900	78.2	75.8	**74.7**	77.3	100.2	97.4	**95.6**	99.7

Tabela 4.1: Tempo de processamento em segundos do algoritmo paralelo para os dois tipos de condições de contorno implementadas

Na Tabela 4.1, mostramos o desempenho das quatro dimensões de blocos que apresentaram os melhores resultados para as duas versões do algoritmo (Dirichlet e PML). Note que os melhores resultados foram para blocos linhas, em particular o bloco com dimensão 1×512 foi o que obteve, em geral, o melhor resultado para ambos os algoritmos.



(a) Aceleração.

(b) Velocidade [MCells/s].

Figura 4.6: Desempenho dos algoritmos paralelos.

A Figura 4.6 é composta por dois gráficos relativos aos melhores resultados dos códigos com condições de contorno de Dirichlet e PML. O primeiro apresenta os resultados de aceleração e o segundo a velocidade (número de milhões de células calculadas por unidade de tempo), variando-se as dimensões da malha de discretização. Note que a partir da dimensão 1500×1500 os resultados se mostraram estáveis. O valor da aceleração para a malha de 6900×6900 no caso de PML, se diferencia bastante dos outros porém os valores da velocidade permanecem próximos. Isso indica que a implementação sequencial apresenta uma queda do desempenho quando a dimensão da malha atinge 6900×6900.

Na Tabela 4.2 temos os valores médios dessas mesmas métricas, além de uma medida de dispersão, para a qual foi adotada a amplitude dos resultados observados (diferença entre o maior valor obtido reduzido do menor). O algoritmo descrito conseguiu um desempenho de 7 a 8 vezes superior ao algoritmo sequencial otimizado.

	Sequencial (O3)		Cuda (1x512)	
	Dirichlet	PML	Dirichlet	PML
Velocidade	89,04 (7.03)	61,08 (10.02)	663,13 (42.05)	486,55(42.39)
Aceleração	-	-	7,08 (0,64)	7.98 (1,58)

Tabela 4.2: Algoritmo sequencial versus algoritmo paralelo. São mostrados os valores das médias e a dispersão.

4.5 Exercícios

1. Obtenha as equações (4.2.2) a partir das equações (4.2.1).
2. Obtenha o esquema de Yee (equações (4.2.7)–(4.2.10)) a partir das equações (4.2.2) e (4.2.4)–(4.2.6).
3. Desenvolva um programa para resolver as equações de Maxwell em um meio heterogêneo, considerando que a fronteira é um condutor perfeito.
   1) Modifique o tipo de dados usado para representar as propriedades do meio. Neste caso, nas equações (4.2.7)–(4.2.10) ϵ, σ, σ^* e os parâmetros a, b, c e d não são constantes mas dependem da posição na malha. Os mesmos devem ser representados por vetores de tipo `double`.
   2) Introduza essa mudança nos kernels 4.1 e 4.2.
4. Modifique os kernels 4.1 e 4.2 para desenvolver um programa para resolver as equações de Maxwell considerando que existem várias fontes distribuídas atuando no domínio de interesse.
   1) Neste caso, a cada ponto da malha deve ser associado o valor da intensidade da fonte (em geral depende do passo de tempo). Devemos considerar vetores de tipo `double` para representar esses dados.
   2) Para uma fonte magnética devemos introduzir mudanças no kernel 4.1 incluindo os vetores de intensidade nos dados de entrada e considerando o efeito da fonte na hora de atualizar o campo magnético de acordo com a equação (4.2.7).
   3) Para uma fonte elétrica devemos introduzir mudanças semelhantes no kernel 4.2 levando em conta as equações (4.2.8), (4.2.9).

Bibliografia

[1] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the Spring Joint Computer Conference*. New York, NY, USA: ACM, 1967. (AFIPS '67 (Spring)), p. 483–485.

[2] AMODIO, P.; MAZZIA, F. Backward error analysis of cyclic reduction for the solution of tridiagonal systems. *Math. Comp.*, AMS, v. 62, n. 206, p. 601–617, 1994.

[3] BELL, N.; DALTON, S.; OLSON, L. N. Exposing fine-grained parallelism in algebraic multigrid methods. *Journal on Scientific Computing*, SIAM, v. 34, n. 4, p. C123–C152, 2012.

[4] BERENGER, J. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, Elsevier, v. 114, n. 2, p. 185–200, 1994.

[5] BLAS. *Basic Linear Algebra Subprograms*. Acessado em julho de 2016. Disponível em: <http://www.netlib.org/blas>.

[6] BURDEN, R.; FAIRES, J. *Análise Numérica*. Sao Paulo, Brasil: Cengage Learning, 2008. (Tradução da 8ª edição norte-americana.).

[7] DONNO, D. D. et al. GPU-based acceleration of computational electromagnetics codes. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, Wiley Online Library, v. 26, n. 4, p. 309–323, 2013.

[8] FOSTER, I. *Designing and building parallel programs : concepts and tools for parallel software engineering*. 1. ed. Boston, MA, USA: Addison-Wesley, 1995.

[9] GANDER, W.; GOLUB, G. H. Cyclic reduction–history and applications. In: *Proceedings of the 6th Workshop on Scientific Computing*. Singapore: Springer Verlag, 1997. p. 73–85.

[10] GASTER, B. et al. *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Waltham, USA: Elsevier, 2012.

[11] GOLUB, G. H.; Van Loan, C. F. *Matrix Computations*. 4. ed. Baltimore: Johns Hopkins University Press, 2013.

[12] GUSTAFSON, J. L. Reevaluating Amdahl's law. *Communications of the ACM*, v. 31, p. 532–533, 1988.

[13] HAND, J. W. Modelling the interaction of electromagnetic fields (10 MHz-10 GHz) with the human body: methods and applications. *Physics in Medicine and Biology*, IOP Publishing, v. 53, n. 16, p. R243, 2008.

[14] HOCKNEY, R. W.; JESSHOPE, C. R. *Parallel Computers: Architecture, Programming and Algorithms*. Bristol: Adam Hilger Ltd, 1987.

[15] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Standard 754-1985*, 1985.

[16] ISHIMARU, A. *Electromagnetic wave propagation, radiation and scattering*. New Jersey: Prentice-Hall, 1991.

[17] Khronos Group. *The open standard for parallel programming of heterogeneous systems*. Acessado em julho de 2016. Disponível em: <https://www.khronos.org/opencl/>.

[18] KINDRATENKO, V. (Ed.). *Numerical Computations with GPUs*. Urbana, USA: Springer, 2014.

[19] MKL-INTEL. *Intel Math Kernel Library*. Acessado em julho de 2016. Disponível em: <https://software.intel.com/en-us/intel-mkl>.

[20] NAVARRO, C. A.; HITSCHFELD-KAHLER, N.; MATEU, L. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, v. 15, p. 285–329, 2014.

[21] NVIDIA Corporation. *CUDA C Programming Guide*. Acessado em setembro de 2015. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide>.

[22] NVIDIA Corporation. *CUDA quick start guide*. Acessado em setembro de 2015. Disponível em: <http://docs.nvidia.com/cuda/cuda-quick-start-guide>.

[23] NVIDIA Corporation. *Nsight Eclipse edition getting started guide*. Acessado em setembro de 2015. Disponível em: <http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide>.

[24] NVIDIA Corporation. *NVIDIA CUDA compiler driver NVCC*. Acessado em setembro de 2015. Disponível em: <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc>.

[25] NVIDIA Corporation. *Parallel Thread Execution ISA Version 4.3*. Acessado em setembro de 2015. Disponível em: <http://docs.nvidia.com/cuda/pdf/ptx_isa_4.3.pdf>.

[26] NVIDIA Corporation. *What is CUDA?* Acessado em setembro de 2015. Disponível em: <http://www.nvidia.com/object/cuda_home_new.html>.

[27] NVIDIA Corporation. *cuBLAS*. Acessado em julho de 2016. Disponível em: <https://developer.nvidia.com/cublas>.

[28] NVIDIA Corporation. *CUDA Math Library*. Acessado em julho de 2016. Disponível em: <https://developer.nvidia.com/cuda-math-library>.

[29] NVIDIA Corporation. *CUDA Zone: Existing University Courses*. Acessado em julho de 2016. Disponível em: <https://developer.nvidia.com/educators/existing-courses>.

[30] NVIDIA Corporation. *CUDA Zone: Training Material and Code Samples*. Acessado em julho de 2016. Disponível em: <https://developer.nvidia.com/cuda-education>.

[31] NVIDIA Corporation. *CUSP*. Acessado em julho de 2016. Disponível em: <https://developer.nvidia.com/cusp>.

[32] NVIDIA Corporation. *GPU-Accelerated Libraries*. Acessado em julho de 2016. Disponível em: <https://developer.nvidia.com/gpu-accelerated-libraries>.

[33] OWENS, J. et al. GPU computing. *Proceedings of the IEEE*, v. 96, n. 5, p. 879–899, May 2008.

[34] PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. 4. ed. Waltham, USA: Elsevier, 2012.

[35] PRESS, W. H. et al. *Métodos numéricos aplicados: rotinas em C++*. 3. ed. Porto Alegre, Brasil: Bookman, 2011.

[36] TAFLOVE, A.; HAGNESS, S. *Computational electrodynamics: the finite-difference time-domain method*. 2. ed. Boston: Artech House, 2000.

[37] VELOSO, L. J. Dissertação de Mestrado, *Implementação do método FDTD para as equações de Maxwell em ambientes de programação paralela*. Rio de Janeiro, Brasil: PPGI-UFRJ, 2015. Programa de Pós Graduação em Informática, Universidade Federal do Rio de Janeiro.

[38] VELOSO, L. J. P.; VIGO, D. G. A.; ROSSETTO, S. Melhorando o desempenho computacional de um esquema de diferenças finitas para as equações de maxwell. *TEMA – Trends in Applied and Computational Mathematics (São Carlos)*, SciELO Brasil, v. 17, n. 1, p. 93–112, 2016.

[39] WHITEHEAD, N.; FIT-FLOREA, A. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *rn (A+ B)*, v. 21, 2011.

[40] WILKINSON, J. H. Error analysis of direct methods of matrix inversion. *J. Assoc. Comput. Mach.*, v. 8, n. 4, p. 281–330, 1961.

[41] WILT, N. *The CUDA handbook: a comprehensive guide to GPU programming*. Indiana, USA: Pearson Education, 2013.

[42] YEE, K. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Trans Antennas Propagation*, IEEE, v. 62, n. 206, p. 302–307, 1966.

[43] ZEIN, G. E.; KHALEGHI, A. Emerging wireless communication technologies. In: *In New Technologies, Mobility and Security*. New York: Springer Netherlands, 2007. p. 271–279.

Índice

aceleração, 13, 29
acelerador, 17
algoritmo de Thomas, 54
algoritmo sequencial, algoritmo paralelo, 1, 2
alinhamento de dados, __align__, 49

blockDim, 21, 37
blockDim.x, 37
blockDim.y, 37
blockIdx, 21
blockIdx.x, 37
blockIdx.y, 37
blocos de threads, 21, 36

capacidade de computação GPU, 47
clock_gettime, 12
coalescência, 49
compilação e execução CUDA, 28
CUDA, 18, 19
cuda_safe_call, 25
cudaDeviceReset, 24
cudaError_t, 25
cudaEvent_t, 27
cudaEventCreate, 27
cudaEventElapsedTime, 27
cudaEventRecord, 27
cudaEventSynchronize, 27
cudaFree, 24
cudaGetErrorString, 25
cudaGetLastError, 26
cudaMalloc, 24
cudaMemcpy, 24
cudaMemcpyDeviceToHost, 24
cudaMemcpyHostToDevice, 24
cudaSuccess, 25

diferenças finitas para as equações de Maxwell, 83
dim3, 39
divergência de instruções, 50

equações de Maxwell, 83
equações lineares, 53
esquema de Yee, 85, 88
eventos em CUDA, 27

fatoração LU, 70
funções assíncronas, 26
funções síncronas, 27

grade, 21, 36
Graphics Processing Units (GPU), 17
grid, 21

hardware GPU, 18
hierarquia de memória, 42

kernel, 20, 21

Lei de Amdahl, 15
Lei de Gustafson, 15

método de fatoração LU, 71
método de redução cíclica, 55, 57, 59
método de redução cíclica paralela, 64, 66, 68
métricas de desempenho, 12
memória compartilhada, extern __shared__, 46
memória compartilhada, __shared__, 43
memória GPU, 22
memória local, 50
modelo de execução CUDA, 40
modelo de programação CUDA, 36
multiplicação de matrizes, 6, 38, 43

Nsight, 28
nvcc, 28

OpenCL, 18, 50
operações de ponto flutuante, 50

paralelismo de dados, 17
programação paralela, 10

saxpy, 2, 20, 30
SIMT, 18, 40
sincronização, 7, 34
sincronização por barreira, 10, 30, 34
sincronização por condição, 10
sincronização por exclusão mútua, 9, 42
sincronização, __syncthreads, 30
sistemas de equações tridiagonais, 54
soma de vetores, 2, 20
speedup, 13

tempo de execução, 12, 26, 29
threadIdx, 20, 21
threadIdx.x, 37
threadIdx.y, 37
threads CUDA, 20

variáveis atômicas, 42
vazão computacional, 13, 29
verificação de erros, 25

warp, 40, 47