UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE BRITO VASCONCELLOS

PROGRAMANDO COM GPUs: PARALELIZANDO O MÉTODO LATTICE-BOLTZMANN COM CUDA

Trabalho de Graduação.

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, junho de 2009.

AGRADECIMENTOS

Gostaria de agradecer a todas as pessoas que me apoiaram durante todo o caminho trilhado nessa longa busca pelo conhecimento. Passei alguns anos de minha vida frequentando o Instituto de Informática. Sou plenamente grato a todos pelo apoio e pela dedicação recebidos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS....8
LISTA DE FIGURAS....9
LISTA DE CÓDIGOS....10
RESUMO....11
ABSTRACT....12
1 INTRODUÇÃO....13
2 CONTEXTO CIENTÍFICO: PROGRAMAÇÃO CONCORRENTE....16
2.1 A Noção de Thread....17
2.2 A Norma POSIX Threads....17
2.3 OpenMP....19
2.4 Message Passing Interface (MPI)....21
3 A PROPOSTA DA NVIDIA PARA GPUS: CUDA....24
3.1 A Arquitetura de GPUs Nvidia....24
3.2 A API CUDA....26
3.2.1 Bloco e grid....26
3.2.2 Extensões CUDA à Linguagem C....27
3.2.2.1 Qualificadores de tipo de função....27
3.2.2.2 Qualificadores de tipo de variável....28
3.2.2.3 Nova sintaxe de chamada de função....28
3.2.2.4 Variáveis auxiliares....28
3.2.3 Restrições da API....29
3.2.4 Síntese: Ilustração do Fluxo de Processamento....29
3.3 Gerenciamento de Memória....30
3.4 Notação de Ponto Flutuante....31
3.5 Aspectos de Desempenho....32
3.6 Exemplos de Programas....33
4 O PROGRAMA LATTICE-BOLTZMANN....36
4.1 A Implementação CUDA....38
4.2 Um Passo a Mais: Paralelização em Cluster de GPUs....42
4.2.1 A Implementação CUDA e MPI....42
4.2.2 Compilando o Código....46
5 AVALIAÇÃO DE DESEMPENHO....47
5.1 Verificando a Corretude da Impementação CUDA....47
5.2 Ganho de Desempenho Obtido com a Tecnologia CUDA....47
5.3 Avaliando a utilização da tecnologia CUDA....51
6 CONCLUSÃO....53
7 BIBLIOGRAFIA....55

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
BD	Banco de Dados
CFD	Dinâmica de Fluidos Computacional
CUDA	Compute Unified Device Architecture
GFLOPS	Giga Operações de Ponto Flutuante por Segundo
GPU	Unidade de Processamento Gráfico
LBM	Método Lattice-Boltzmann
MPI	Message Passing Interface
RAM	Memória de Acesso Randômico
UFRGS	Universidade Federal do Rio Grande do Sul

LISTA DE FIGURAS

Figura 1.1: Evolução dos GFLOPS das GPUs e CPUs....14
Figura 1.2: Evolução da BANDWIDTH das GPUs e CPUs....15
Figura 2.1: Uma árvore de *threads*....17
Figura 2.2: Sincronização de *threads*....18
Figura 2.3: O modelo fork-join....20
Figura 3.1: Destinação dos transistores em GPUs e CPUs....24
Figura 3.2: A arquitetura da GPU Tesla....25
Figura 3.3: A pilha de software da plataforma CUDA....26
Figura 3.4: Exemplo de um grid com seus blocos....27
Figura 3.5: Fluxo de execução de um programa CUDA....29
Figura 3.6: Caminho de execução de uma aplicação CUDA....30
Figura 4.1: O laço principal do método LBM....36
Figura 4.2: Modelos de lattice....36
Figura 4.3: Modelos de reticulado bidimensionais....37
Figura 4.4: Modelos de reticulado tridimensionais....37
Figura 4.5: Propagação das partículas....37
Figura 4.6: Condição de contorno....37
Figura 4.7: Modelando o reticulado D2Q9....39
Figura 4.8: Comunicação entre processos....43
Figura 5.1: Desempenho para bloco 1 x 512....46
Figura 5.2: Distribuição de warps para bloco 1 x 512....46
Figura 5.3: Desempenho para bloco 256 x 1....47
Figura 5.4: Distribuição de warps para bloco 256 x 1....47
Figura 5.5: Desempenho para bloco 512 x 1....48
Figura 5.6: Distribuição de warps para bloco 512 x 1....48
Figura 5.7: Distribuição dos dados na memória....49

LISTA DE CÓDIGOS

Código 1.1: Multiplicação de *X* por *Y*.....13
Código 2.1: Função onde há dependência de dados.....16
Código 2.2: Função onde não há dependência de dados.....16
Código 2.3: Programa multithread.....17
Código 2.4: Sincronizando múltiplas *threads*.....18
Código 2.5: Cálculo do produto escalar com múltiplas *threads*.....19
Código 2.6: Produto escalar com OpenMP.....20
Código 2.7: Variáveis privadas em OpenMP.....20
Código 2.8: Utilizando as diretivas do OpenMP.....21
Código 2.9: Inicializando e encerrando um programa MPI.....22
Código 2.10: Identificando um processo.....22
Código 2.11: Troca de mensagens.....23
Código 3.1: Alocando um *array* bidimensional.....30
Código 3.2: Utilizando um CUDA *array*.....31
Código 3.3: Alocação estática de memória.....31
Código 3.4: Paralelizando o algoritmo SAXPY.....33
Código 3.5: Redução paralela.....33
Código 3.6: Produto escalar em CUDA.....34
Código 4.1: Inicialização da estrutura de parâmetros.....38
Código 4.2: Definição da estrutura do *lattice*.....38
Código 4.3: Alocação de memória para o *lattice*.....39
Código 4.4: O laço principal da simulação.....39
Código 4.5: Algoritmo sequencial para a função *redistribute*.....40
Código 4.6: Paralelizando a a função *redistribute*.....40
Código 4.7: Execução sequencial das funções *propagate*, *bounceback* e *relaxation*.....41
Código 4.8: Paralelizando as funções propagate, bounceback e relaxation.....41
Código 4.9: Estrutura de dados para abrigar o lattice.....43
Código 4.10: Estrutura de comunicadores.....43
Código 4.11: Inicializando os comunicadores.....43
Código 4.12: A paralelização com CUDA e MPI.....44
Código 4.13: Cópia de dados entre GPU e memória RAM.....45
Código 4.14: Sincronização MPI.....45
Código 4.15: Enviando dados da borda.....45
Código 4.16: Colocando os dados na rede.....46
Código 4.17: Arquivo *makefile* para CUDA e MPI.....46
Código 5.1: Somando dois vetores com Posix Threads.....51
Código 5.2: Somando dois vetores com OpenMP.....52
Código 5.3: Somando dois vetores com CUDA.....52

RESUMO

O presente trabalho consiste na exploração do potencial computacional das GPUs da Nvidia, através do uso da tecnologia CUDA, com a finalidade de otimizar o tempo de execução da simulação do Método Lattice-Bolzmann (LBM). Para tanto, um breve estudo sobre algumas técnicas para a programação paralela, passando por Posix *Threads*, OpenMP e *Message Passing Interface* (MPI) é apresentado. Adicionalmente, apresenta-se a arquitetura das GPUs Nvidia e a tecnologia CUDA. Duas implementações paralelas do LBM são propostas: uma utilizando a tecnologia CUDA e outra utilizando CUDA em conjunto com MPI. Por fim, apresenta-se os resultados obtidos para esta implementações.

Palavras-Chave: paralelismo, LBM, *thread*, MPI, OpenMP, CUDA, GPU.

Programming with GPUs: Parallelizing the LBM

ABSTRACT

The present work consists in the exploration of the computational potencial of the Nvidia's GPUs, through the use of the CUDA technology, aiming to optimize the execution time of the Lattice-Boltzmann Method (LBM). To do that, a short study about some techniques for parallel programming, showing Posix Threads, OpenMP and Message Passing Interface (MPI) is presented. Moreover, it is presented the architecture of the Nvidia's GPUs and the CUDA technology. Two parallel implementations of the LBM are proposed: one using the CUDA technology and another using CUDA and MPI. Finally, the results obtained with this two implementations are presented.

Keywords: parallelism, LBM, thread, OpenMP, MPI, CUDA, GPU.

1. INTRODUÇÃO

A Ciência da Computação é, por excelência, uma ciência que se ocupa com o problema do processamento de dados. Seu objetivo maior é transformar dados de entrada em dados de saída através de ciclos de processamento, ou seja, sucessivas computações. Pode-se dizer que a multiplicação de dois números naturais x e y feita com papel e caneta através do algoritmo "some-se x y vezes" é uma computação. Entretanto, com o advento dos computadores, a computação passou a ser realizada por máquinas, as quais são programáveis. Um programa de computador é uma sequência de instruções, ou comandos, que a máquina é capaz de interpretar e executar. O seguinte pseudocódigo representa o programa que implementa o algoritmo da multiplicação de dois números naturais descrito acima.

Código 1.1: Multiplicação de *X* por *Y*

```
1. resultado = 0
2. PARA i = 0 ENQUANTO i < y FAÇA
3.   resultado = resultado + x
4. PRÓXIMO i
```

O principal ganho que se obteve, inicialmente, com o advento dos computadores foi a rapidez na execução das tarefas. Um ser humano, com papel e caneta, consome alguns minutos para multiplicar dois números naturais de tamanho razoável, enquanto que uma máquina o faz em apenas alguns milissegundos. A partir desse contexto, o objetivo maior do progresso obtido com o avanço da tecnologia é o de reduzir o tempo da computação.

Basicamente, essa redução foi obtida através do aumento do poder de processamento das CPUs ao longo do tempo. Esse crescimento tornou-se concreto através da crescente miniaturização dos componentes de hardware das CPUs, a qual possibilitou o contínuo incremento do seu *clock*. Entretanto, em 8 de maio de 2004 (FLYNN, 2004), a Intel anunciou o cancelamento de dois projetos de processadores: Tejas e Jayhawk. O motivo desse cancelamento foi o nível de consumo de energia alcançado e o consequente aquecimento do hardware, o que tornou inviável a continuidade do incremento do *clock* dos processadores. A partir desse momento, a Intel concentrou-se em projetos de processadores de dois núcleos. Em outubro de 2005 a Intel apresentou ao mercado seu novo processador: Dual Core Xeon de 2.8 Ghz.

Tradicionalmente, um programa de computador é um conjunto de instruções executadas sequencialmente por um processador. Sendo assim, o tempo levado por uma computação é o tempo necessário para que todas as instruções sejam executadas pelo processador. Como elas são executadas em sequência, o tempo total da computação é a soma dos tempos de cada instrução. Naturalmente, com a evolução do hardware, o mesmo programa era executado em um tempo cada vez menor, bastando apenas a

compra de um novo processador. Com o fim do incremento do *clock* das CPUs, esse fácil ganho de performance acabou.

Considerando-se que o tempo de execução de uma instrução de máquina parou de diminuir, a forma de reduzir o tempo de uma computação é através da paralelização da execução das instruções, ou seja, busca-se executar mais de uma instrução ao mesmo tempo. Para que isso seja possível, é preciso executar o código em um computador paralelo. Um computador paralelo é um conjunto de processadores capazes de trabalhar cooperativamente para resolver um problema computacional (FOSTER, 1995).

Retornando ao exemplo do algoritmo de multiplicação de dois números naturais visto no início desse capítulo, tem-se que aquele programa pode ser reescrito de forma a ter seu tempo de execução reduzido. A multiplicação de *X* por *Y*, baseado no método descrito em 1.1, é o equivalente à realização do somatório de *X*, *Y* vezes, em uma variável *resultado*. Ou seja,

$$resultado = x + x + ... + x,\ Y \text{ vezes.}$$

Esse somatório pode ser dividido em dois fluxos de execução, onde cada fluxo computa metade do somatório em duas variáveis $resultado_1$ e $resultado_2$, ou seja,

$$resultado_1 = x + x + ... + x,\ Y/2 \text{ vezes e}$$

$$resultado_2 = x + x + ... + x,\ Y/2 \text{ vezes.}$$

Considerando que os dois fluxos são executados em paralelo, apenas metade do tempo utilizado pelo algoritmo sequencial é necessário para realizar a mesma computação, com o acréscimo de uma instrução final que soma os resultados parciais de cada fluxo para se chegar ao resultado final: $resultado = resultado_1 + resultado_2$.

Impulsionado pela crescente complexidade do processamento gráfico, especialmente em aplicações de jogos 3D, as placas de vídeo passaram por um tremendo progresso tecnológico. Atualmente, as placas de vídeo são pesadamente paralelas. A figura 1.1 exibe uma comparação entre a capacidade de processamento das GPUs da Nvidia e dos processadores da Intel, no que diz respeito às operações de ponto flutuante. Pode-se observar que, a partir de novembro de 2006, as GPUs da Nvidia estabeleceram uma larga vantagem no poder computacional em relação às CPUs da Intel.

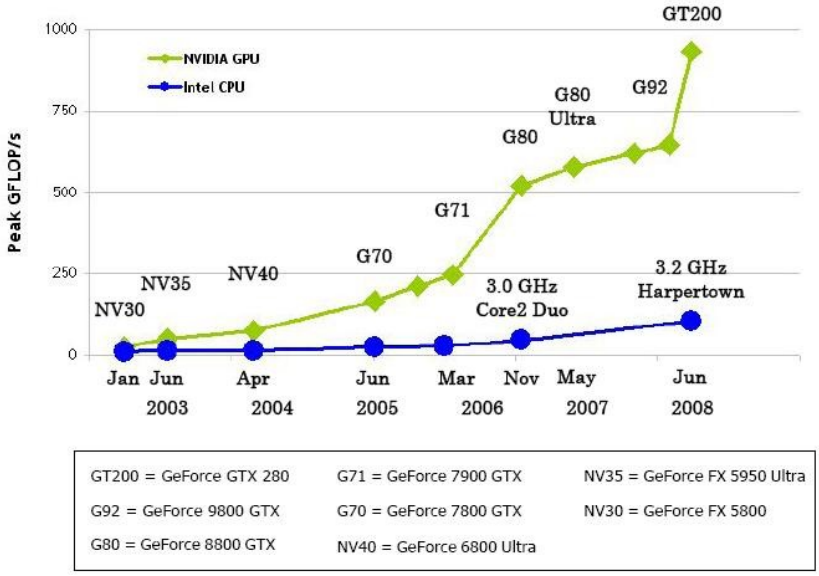

Figura 1.1: Evolução dos GFLOPS das GPUs e CPUs (NVIDIA CUDA, 2009).

Em outro aspecto importante na análise do poder de processamento de um hardware, a largura de banda da memória (*memory bandwidth*), as GPUs da Nvidia também ultrapassaram as CPUs da Intel, conforme pode ser visto na figura 1.2.

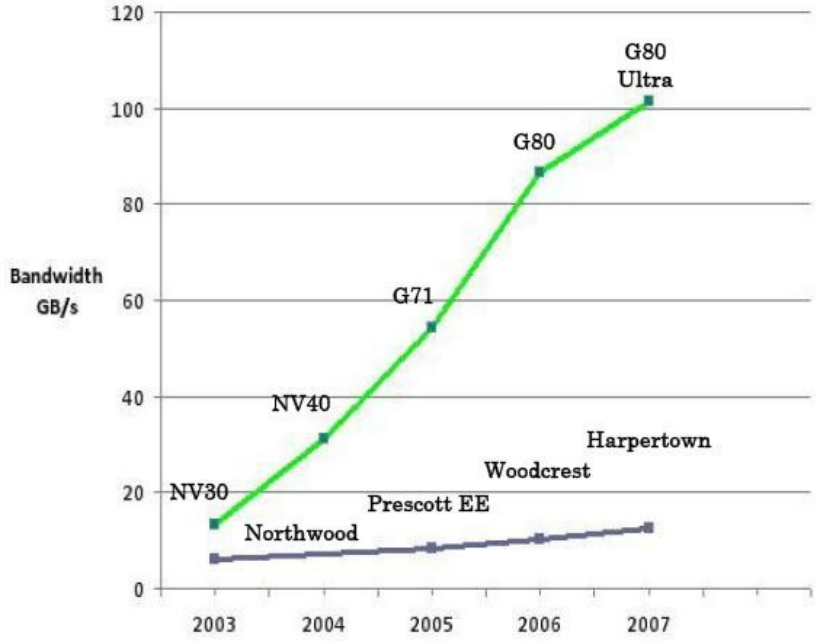

Figura 1.2: Evolução da BANDWIDTH das GPUs e CPUs (NVIDIA CUDA, 2009).

Tendo em vista esse enorme potencial computacional oferecido pelas GPUs da Nvidia, essa empresa, em 15 de fevereiro de 2007, tornou público uma nova plataforma de software: o CUDA. Com o CUDA é possível explorar o potencial computacional das GPUs, até então restrito à computação gráfica, para computação em geral.

Dado esse contexto, o presente trabalho procura explorar o potencial dessa tecnologia para a paralelização do Método Lattice-Boltzmann (LBM). O método LBM é uma técnica númerica iterativa de modelagem e simulação de fluidos dinâmicos. Para tanto, partiu-se de uma implementação sequencial do método. Além disso, um passo a mais começou a ser trilhado: a paralelização em rede utilizando as tecnologias CUDA e *Message Passing Interface* (MPI). A tecnologia MPI permite distribuir uma determinada carga de processamento entre diversos nodos de um cluster. Um cluster é um conjunto de computadores conectados através de uma interface de rede dedicados a processar determinada tarefa.

Sendo assim, o capítulo 2 apresenta uma contextualização da programação concorrente, onde é feita, inicialmente, uma análise do conceito de *thread*, para logo após apresentar duas propostas concretas que implementam *threads*: a norma POSIX Threads e o OpenMP. No capítulo 3 apresenta-se a plataforma CUDA através de uma breve análise da arquitetura atual das GPUs da Nvidia, com a apresentação da API CUDA, seguido de alguns exemplos de programas escritos em CUDA. O capítulo 4 apresenta duas implementações paralelas do Método Lattice-Boltzmann (LBM): uma utilizando a tecnologia CUDA e outra utilizando CUDA com *Message Passing Interfac*e (MPI). O capítulo 5 apresenta os resultados experimentais obtidos em laboratório demonstrando os ganhos obtidos com a exploração dessa nova tecnologia, seguidos de uma breve análise desses resultados. Por fim, o capítulo 6 apresenta a conclusão deste trabalho.

2. CONTEXTO CIENTÍFICO: PROGRAMAÇÃO CONCORRENTE

Programação concorrente é aquela onde diversos processos cooperam entre si para executar simultaneamente um determinado programa (OLIVEIRA; CARÍSSIMI; TOSCANI, 2001). Para que um programa possa ser executado concorrentemente é preciso que não hajam dependências entre os dados a serem processados. Uma instrução B depende da instrução A quando ela necessita do resultado da computação de A para a sua execução.

O trecho de código abaixo ilustra uma função onde há dependência de dados.

Código 2.1: Função onde há dependência de dados

```
5. function Dep(a, b) {
6.    c := a * b;
7.    d := 2 * c;
8. }
```

Conforme pode ser observado no trecho de código acima, a instrução da linha 3 depende do conteúdo da variável C, o qual só estará disponível após a computação da instrução da linha 2. Portanto, a instrução da linha 3 não pode ser executada enquanto a instrução da linha 2 não for concluída.

O próximo trecho de código ilustra uma função onde não há dependência de dados.

Código 2.2: Função onde não há dependência de dados

```
1. function NoDep(a, b) {
2.    c := a * b;
3.    d := 2 * b;
4.    e := a + b;
5. }
```

Neste código as instruções das linhas 2, 3 e 4 são totalmente independentes, pois elas utilizam apenas os valores das variáveis A e B, os quais estão disponíveis por terem sido passados como argumentos na chamada da função *NoDep*. Desta forma, é possível executar as instruções das linhas 2, 3 e 4 paralelamente.

Para a implementação do paralelismo é necessário suporte do sistema operacional e da linguagem de programação utilizada. Na seção 2.1 será visto o conceito de *thread*. Na seção 2.2 a implementação pelo sistema operacional de *threads* e sua implementação utilizando a linguagem de programação C: a Norma POSIX Threads. Na seção 2.3 uma implementação alternativa de paralelismo: a linguagem OpenMP. Por fim, a seção 2.4 apresenta uma implementação em rede de paralelismo: *Message Passing Interface* (MPI).

2.1 A Noção de Thread

Uma *thread* é um fluxo de execução independente que possui uma memória compartilhada com o processo pai e que pode ser escalonada pelo sistema operacional (BARNEY, 2009). Normalmente, a utilização de *threads* ocorre em conjunto com a repartição dos dados a serem processados. Dessa forma, cada *thread* pode processar um subconjunto dos dados paralelamente.

2.2 A Norma POSIX Threads

Uma das alternativas de implementação de *threads* é a norma POSIX Threads. A API da norma POSIX Threads define funções, variáveis e constantes para criar, destruir e gerenciar *threads*. Existem implementações da norma para diversos sistemas operacionais, tais como: Linux, Solaris e Windows. A biblioteca que implementa essa norma é chamada de Pthreads. Para implementar um programa que utilize a biblioteca Pthreads é necessário incluir a diretiva **#include <pthread.h>** no cabeçalho do código. Na compilação é preciso passar o argumento **-lpthread**, considerando o uso do compilador *gcc*.

Basicamente, basta criar *threads* para que elas possam executar alguma computação. Para criar uma *thread*, é preciso definir uma variável do tipo **pthread_t** e executar uma chamada à função **pthread_create()** passando alguns argumentos. Segue trecho de código para criar uma *thread* que executa a função **thread_func**.

Código 2.3: Programa multithread

```
1.  #include <pthread.h>
2.  #include <stdio.h>
3.
4.  void *thread_func() {
5.    printf("Oi. Eu sou uma thread.\n");
6.  }
7.
8.  int main() {
9.    pthread_t thread;
10.   pthread_create(&thread, NULL, thread_func, NULL);
11.   pthread_exit(NULL);
12. }
```

Entretanto, para obter paralelismo é preciso criar ao menos duas *threads*. Na implementação NPTL, cuja primeira implementação aconteceu no sistema operacional Linux 2.6, é possível criar bilhões de *threads* (DREPPER, 2005), caso o hardware tenha recursos disponíveis. Adicionalmente, as *threads* podem criar *threads*, gerando uma árvore de *threads*. A figura 2.1 ilustra uma árvore de *threads*.

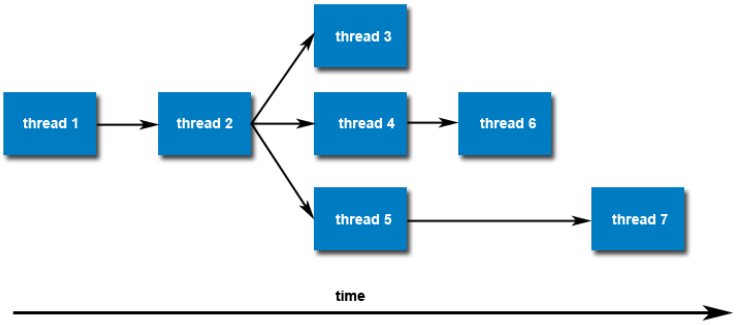

Figura 2.1: Uma árvore de *threads* (BARNEY, 2009).

Como a execução das *threads* depende do escalonador do sistema operacional, sua ordem de execução é indeterminada. Ou seja, não é possível saber qual *thread* vai encerrar a sua execução primeiro. Isso pode introduzir inconsistências na computação. Esse problema é solucionado com o uso de sincronização das *threads*. Dessa forma é possível determinar explicitamente que uma *thread* deve esperar a finalização de outra *thread* antes de continuar. A figura 2.2 exibe a problemática da sincronização de *threads*.

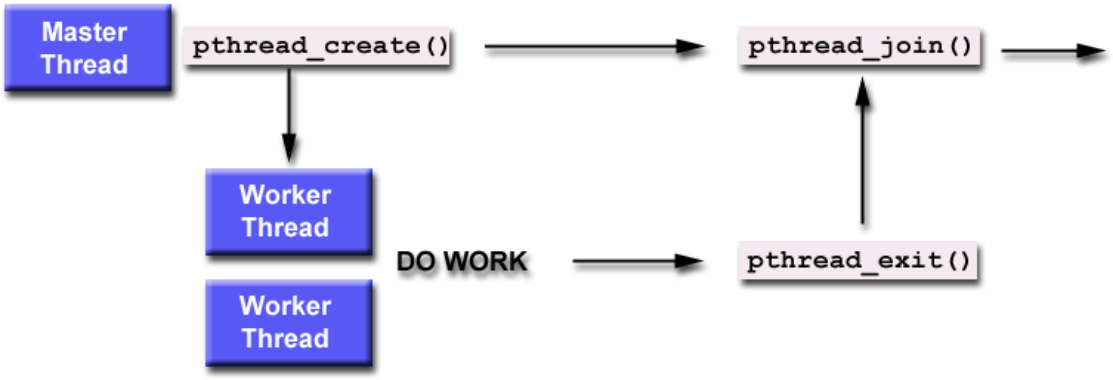

Figura 2.2: Sincronização de *threads* (BARNEY, 2009).

O trecho de código abaixo ilustra a utilização de sincronização de *threads*. São criadas quatro *threads*. O uso da diretiva **pthread_join** garante que o programa principal não vai continuar antes do fim da execução das quatro *threads*. Note que a *thread* precisa ser definida como *joinable* para que essa diretiva tenha significado. Isso é feito através da definição de atributos. A criação, inicialização e configuração dos atributos é feita através das diretivas **pthread_attr_**, conforme código abaixo.

Código 2.4: Sincronizando múltiplas threads

```
1.  #include <pthread.h>
2.  #include <stdio.h>
3.  #define THREADS 4
4.
5.  void *thread_func(void *arg) {
6.    int i = (int*)arg;
7.    printf("Eu sou a thread %d.\n", i);
8.  }
9.
10. int main() {
11.   pthread_t thread[THREADS];
12.   pthread_attr_t attr;
13.   pthread_attr_init(&attr);
14.   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
15.
16.   int i;
17.   for (i=0; i<THREADS; i++) {
18.     pthread_create(&thread[i], &attr, thread_func, (void *)i);
19.   }
20.
21.   pthread_attr_destroy(&attr);
22.   for(i=0; i<THREADS; i++) {
23.     pthread_join(thread[i], NULL);
24.   }
25.   pthread_exit(NULL);
26. }
```

Outro problema que pode ocorrer com o uso de *threads* é o da escrita concorrente. Caso duas *threads* realizem uma operação de escrita em uma mesma variável compartilhada, é possível que uma das escritas não seja computada. Esse problema é solucionado com o uso de *mutexes*. Um *mutex* define um trecho de código que deve ser executado atomicamente, ou seja, sem interrupções. A criação e utilização de um *mutex* é feita através das diretivas **pthread_mutex_**. O código abaixo ilustra a implementação do produto escalar entre dois vetores A e B de tamanho N utilizando a biblioteca Pthreads. Cada *thread* processa um subconjunto do somatório necessário ao cálculo do produto escalar.

Código 2.5: Cálculo do produto escalar com múltiplas threads

```
1.  void * prodEscalar(void *arg) {
2.    int thread_task = 0, i;
3.    int begin = ((int)arg) * (VECTOR_LENGTH/NUMTHREADS);
4.    int end = begin +  VECTOR_LENGTH/NUMTHREADS;
5.
6.    for(i=begin; i<end ; i++) {
7.      thread_task += a[i] * b[i];
8.    }
9.    produto_escalar = 0;
10.   pthread_mutex_lock(&mutexsum);
11.   produto_escalar += thread_task;
12.   pthread_mutex_unlock(&mutexsum);
13.
14.   pthread_exit((void*) 0);
15. }
16.
17. void main() {
18.   for(i=0; i<NUMTHREADS; i++) {
19.     pthread_create( &threads[i], &attr, prodEscalar, (void *)i);
20.   }
21.
22.   pthread_attr_destroy(&attr);
23.   for(i=0; i<NUMTHREADS; i++) {
24.     pthread_join(threads[i], &status);
25.   }
26.
27.   printf ("Produto escalar = %d\n", produto_escalar);
28.   pthread_mutex_destroy(&mutexsum);
29.   pthread_exit(NULL);
30.
31. }
```

2.3 OpenMP

O OpenMP é uma API para implementar paralelização de código (BARNEY, 2009). Esta API foi definida por um consórcio de fabricantes de hardware e software, além de ter a participação de membros do governo americano e de universidades (GALLINA, 2006). Este modelo de programação está baseado no uso de máquinas *multi-threaded* com memória distribuída. O objetivo do OpenMP é tornar explícito o paralelismo existente nos programas. Ele adiciona uma camada de abstração além no nível das *threads*, o que implica em que o programador não precisa se preocupar com a tarefa de criar, gerenciar e destruir *threads*.

O padrão OpenMP está definido nas linguagens Fortran, C e C++, tendo implementações disponíveis para *Windows* e *Unix*. Ele utiliza o modelo *fork-join*, ilustrado na figura 2.3, para paralelizar a execução de código.

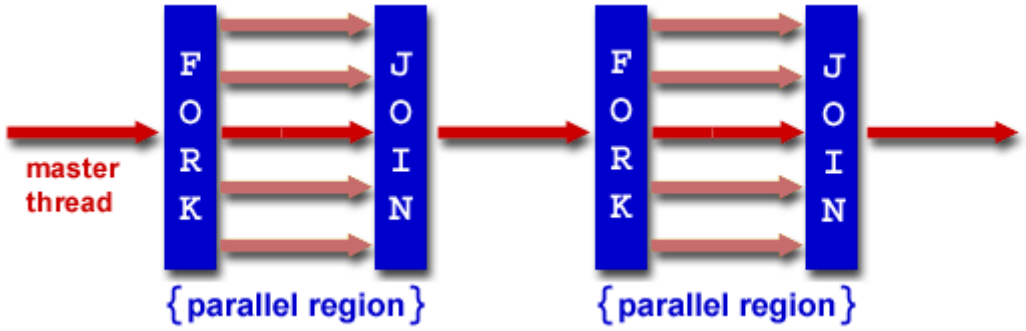

Figura 2.3: O modelo fork-join (BARNEY, 2009).

O programador deve explicitar as regiões do código que deverão ser paralelizadas através do uso da diretiva *#pragma*. O compilador converte estas diretivas em chamadas para *threads*. Em tempo de execução, serão criadas tantas *threads* quantos forem os *cores* do computador alvo, a menos que o programador explicite este número. Segue exemplo de como paralelizar o cálculo do produto escalar com OpenMP.

Código 2.6: Produto escalar com OpenMP

```
1.  #include <omp.h>
2.  #define N 1000
3.
4.  double x[N], y[N];
5.  double res = 0;
6.
7.  #pragma omp parallel for reduction(res:+)
8.  for (i=0;i<N;i++) {
9.    res += x[i] * y[i];
10. }
11.
12. printf("A norma eh: %lf\n", res);
```

Conforme pode ser observado no trecho de código acima, a linha 7 apresenta as definições em OpenMP para a paralelização do laço compreendido pelas linhas 8, 9 e 10. Ainda, ela define uma restrição com relação à variável *res*, determinando que cada *thread* terá uma cópia da variável e que, ao final da execução, ela sofrerá o processo de redução para a operação de soma. No programa acima, será criada uma *thread* para cada núcleo existente na arquitetura alvo.

Caso o programador deseje definir um número diferente de *threads* ele deve utilizar a diretia *omp_set_num_threads* passando como argumento o número de *threads* a serem criadas. É possível definir variáveis privadas para cada *thread*, isto é realizado através do uso da diretiva *private*. O código 2.7 ilustra o uso destas diretivas.

Código 2.7: Variáveis privadas em OpenMP

```
1.  omp_set_num_threads(4);
2.  int tid;
3.  #pragma omp parallel for private(tid)
4.  for (i=0; i<8; i++) {
5.    tid = omp_get_thread_num();
6.    printf("Eu sou a thread %d.\n", tid);
7.  }
```

Conforme pode ser obvervado na linha 1, no momento da execução serão criadas 4 *threads* para executar o laço definido nas linhas 4 a 7. Ainda, a variável *tid* é definida como *private*, o que significa dizer que cada *thread* terá uma cópia desta variável em seu espaço privado de memória. Em contrapartida, a diretiva *shared* define que uma variável é compartilhada por todas as *threads*.

Além disso, o OpenMP oferece as diretivas *barrier* e *master*. A diretiva *barrier* impõe uma barreira de sincronização no código, determinando que a execução só pode continuar quando todas as *threads* chegarem neste ponto. Por outro lado, a diretiva *master* define um trecho de código que é executado apenas pela *thread* principal, que é aquela cujo valor de retorno da função *omp_get_thread_num* é igual a 0 (zero). Adicionalmente, a diretiva *critical* define um região de sessão crítica no código, onde apenas uma *thread* executa o trecho de código por vez. Por fim, o programador pode explicitar como será a distribuição de carga de um laço entre as *threads* através do uso da diretiva *schedule*. O código 2.8 ilustra o uso das diversas diretivas do OpenMP.

Código 2.8: Utilizando as diretivas do OpenMP

```
1.  int x = 10;
2.  int y = 0;
3.  int z = 0;
4.  omp_set_num_threads(5);
5.  #pragma omp parallel for private(y) shared(x, z) schedule(static, 10)
6.  for (i=0; i<100; i++) {
7.    y += x;
8.    #pragma omp master
9.    {
10.     printf("Somente a thread principal... Y = %d\n", y);
11.   }
12.   #pragma omp critical
13.   {
14.     z += y;
15.   }
16.   y += x;
17.   #pragma omp barrier
18.   printf("Agora eu posso continuar...\n");
19. }
```

Conforme pode ser observado no código acima, a linha 4 define que serão criadas 5 *threads* no momento da execução deste código, independentemente do número de *cores* da arquitetura alvo. A linha 5 define a paralelização do laço das linhas 6 a 19, define que a variável *Y* é privada a cada *thread*, que as variáveis *X* e *Z* são compartilhadas por todas as *threads* e que cada *thread* executa uma carga de 10 iterações do laço por vez. O trecho de código compreendido pelas linhas 9 a 11 são executadas apenas pela *thread* principal. Adicionalmente, o trecho de código definido pelas linhas 13 a 15 é executado sequencialmente pelas *threads*, uma de cada vez. Por fim, a linha 17 define uma barreira de sincronização a partir do qual a execução só continua quando todas as *threads* chegarem neste ponto.

2.4 Message Passing Interface (MPI)

A tecnologia MPI é um padrão para a comunicação de dados em computação paralela (SNIR, 1996). Ela fornece as diretivas básicas para a troca de mensagens entre os diversos processos, os quais podem estar distribuídos em um cluster.

Basicamente, um programa MPI consiste lançar diversos processos pesados, cada um com seu espaço privado de endereçamento, os quais atuam sobre um determinado programa. A diferenciação de cada processo é realizada através dos identificadores de processo. É com o uso destes identificadores que o programador particiona o código entre os diversos processos. Além disso, eles permitem a comunicação entre os processos. O MPI provê uma API para a comunicação entre os processos. Dentre as principais funções disponíveis encontram-se: *MPI_Send*, *MPI_Recv*, *MPI_Bcast*, *MPI_Barrier*, *MPI_Comm_create*, *MPI_Reduce*, dentre outras. A inicialização e o encerramento de um programa MPI é feito através das diretivas *MPI_Init* e *MPI_Finalize*, conforme pode ser observado no código 2.9 abaixo.

Código 2.9: Inicializando e encerrando um programa MPI

```
1. #include <mpi.h>
2.
3. MPI_Init(&argv, &argc);
4. ...
5. MPI_Finalize();
```

Uma vez visto como inicializar e encerrar um programa MPI, o próximo passo é explorar as funções de identificação de processos, a saber: *MPI_Comm_rank* e *MPI_Comm_size*. Estas duas funções retornam na variável passada como segundo argumento o *rank* do processo, ou seja, o seu identificador no grupo informado no primeiro argumento, e o número total de processos ligados a este comunicador, respectivamente. O MPI tem por definição todos os processos agrupados em um comunicador padrão chamado *MPI_COMM_WORLD*. A listagem 2.10 ilustra o uso destas funções.

Código 2.10: Identificando um processo

```
1. #include <mpi.h>
2. #include <stdio.h>
3.
4. int meu_rank, total_de_processos;
5.
6. MPI_Init(&argv, &argc);
7.
8. MPI_Comm_size(MPI_COMM_WORLD, &total_de_processos);
9. MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
10.
11. printf("Existem %d processos em execução.\n", total_de_processos);
12.
13. if (meu_rank == 0)
14.   printf("Eu sou o processo pai.\n");
15. else
16.   printf("Eu sou o processo %d.\n", meu_rank);
17.
18. MPI_Finalize();
```

Após a execução deste código, a variável *total_de_processos* conterá o número correspondente ao total de processos MPI disparados e a variável *meu_rank* conterá o valor do identificador de cada processo dentro do comunicador *MPI_COMM_WORLD*. Note que cada processo MPI contém a sua cópia privada destas variáveis. A definição de quantos processos serão disparados é feita em tempo de execução através da chamada *mpirun* passando como argumento o número de processos a serem disparados.

Adicionalmente, o padrão MPI apresenta duas funções para troca de mensagens, a saber: *MPI_Send* e *MPI_Recv*. Estas funções definem uma mensagem e seu tamanho, o

tipo de dado da mensagem, um destinatário ou um receptor, uma *tag* para identificar a mensagem, o comunicador ao qual o *rank* referencia os processos e, para o caso do *MPI_Recv*, o *status* da mensagem recebida. Através do uso destas funções é possível enviar mensagens de um processo a outro. O código 2.11 ilustra o uso destas funções em um programa onde os processos filhos enviam mensagens para o processo raiz, o qual imprime estas mensagens na saída padrão.

Código 2.11: Troca de mensagens (MOR, 2007)

```
1.  #include <mpi.h>
2.  #include <stdio.h>
3.  #include <string.h>
4.
5.  int main(int argc, char *argv[]) {
6.    int meu_rank, total_de_processos, rank_origem, rank_destino, tag = 50;
7.    char mensagem[100];
8.    MPI_Status status;
9.
10.   MPI_Init(&argv, &argc);
11.   MPI_Comm_size(MPI_COMM_WORLD, &total_de_processos);
12.   MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
13.
14.   if (meu_rank != 0) {
15.     sprintf(mensagem, "Eu sou o processo %d.\n", meu_rank);
16.     rank_destino = 0;
17.     MPI_Send(mensagem, strlen(mensagem) + 1, MPI_CHAR, rank_destino, tag,
    MPI_COMM_WORLD);
18.   } else {
19.     for (rank_origem = 1; rank_origem < total_de_processos; rank_origem++) {
20.       MPI_Recv(mensagem, 100, MPI_CHAR, rank_origem, tag,
    MPI_COMM_WORLD, &status);
21.       printf("%s\n", mensagem);
22.     }
23.   }
24.
25.   MPI_Finalize();
26.   return 0;
27. }
```

De acordo com a listagem acima, pode-se observar que o trecho compreendido pelas linhas 19 a 22 será executado apenas pelo processo raiz, cujo *rank* é 0 (zero). Por outro lado, os demais processos em execução irão executar o trecho de código das linhas 15 a 17. Sendo assim, cada processo filho imprime uma mensagem na variável *mensagem*, incluindo o seu *rank* para identificar a origem da mensagem quando esta for impressa na saída padrão, e realiza uma chamada a função *MPI_Send* informando que o processo destino é o de *rank* igual a 0 (zero), ou seja, o processo raiz. Paralelamente, o processo raiz efetua uma chamada a função *MPI_Recv* para cada processo filho existente no comunicador global *MPI_COMM_WORLD*. Após o recebimento da mensagem, o processo raiz imprime na saída padrão o seu conteúdo.

Uma vez apresentado este breve estudo sobre diversas técnicas para a programação paralela, o próximo capítulo irá apresentar a arquitetura das GPUs da Nvidia, bem como a tecnologia CUDA.

3. A PROPOSTA DA NVIDIA PARA GPUS: CUDA

CUDA é uma plataforma de software para computação massivamente paralela de alto desempenho que utiliza o poder de processamento das GPUs da Nvidia. A arquitetura da GeForce 8 possui 128 processadores de *threads*, suportando um total de 12.288 *threads* concorrentes (HALFHILL, 2008). A plataforma foi formalmente introduzida em fevereiro de 2007 (CUDA, 2007). Atualmente, encontra-se em sua versão 2.2. Ela tem ganho usuários nos campos científico, biomédico, da computação, da análise de risco e da engenharia devido às características das aplicações nesses campos, as quais são altamente paralelizáveis. Essa tecnologia tem chamado a atenção da comunidade acadêmica devido ao seu grande potencial computacional. É o que será analisado neste capítulo, com a seguinte organização: primeiramente, será apresentada a arquitetura das GPUs Nvidia (seção 3.1), após, a API CUDA (seção 3.2), a seguir, o gerenciamento de memória em CUDA (seção 3.3) e, por fim, exemplos de programas em CUDA (seção 3.4).

3.1 A Arquitetura de GPUs Nvidia

Conforme foi visto na introdução (ver figura 2), a capacidade de processamento das GPUs da Nvidia ultrapassou a das CPUs da Intel. O principal motivo desse enorme ganho de performance é que as GPUs são dedicadas ao processamento de dados exclusivamente, não tendo a preocupação em guardar informações em memórias cache nem em tratar de um controle de fluxo altamente complexo. Isso deve-se ao fato de as aplicações gráficas serem extremamente paralelas, realizando a mesma operação em um grande volume de dados. Desta forma, a área destinada à memória cache e controle de fluxo nas CPUs são utilizadas para processamento de dados nas GPUs. A figura 3.1 exibe uma comparação entre a destinação dos transistores em GPUs e CPUs.

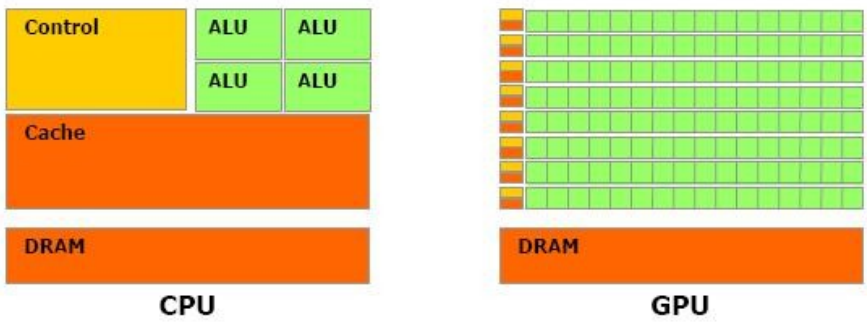

Figura 3.1: Destinação dos transistores em GPUs e CPUs (NVIDIA CUDA, 2009).

Em novembro de 2006 (NVIDIA CUDA, 2009), a Nvidia introduziu no mercado sua primeira GPU destinada à computação de propósitos gerais (GPGPU): a arquitetura Tesla. Essa nova arquitetura é exclusivamente dedicada ao processamento de dados, não

tendo como característica a utilização como placa de vídeo. Algumas de suas principais características são: 4 GPUs com 240 processadores de *threads* cada, 16 GB de memória principal com velocidade de acesso de 102 GB/s em cada GPU, totalizando 408 GB/s de pico e barramento da memória de 512 bits, uma interface para cada GPU, resultando num poder computacional de 4 TeraFlops (NVIDIA TESLA, 2008).

Através dessa nova arquitetura, a Nvidia introduziu um novo conceito na computação paralela: *single-instruction multiple-thread*, ou SIMT. Essa arquitetura cria, gerencia, agenda e executa *threads* automaticamente em grupos de 32 *threads* paralelas, o que a Nvidia chama de *warp*, com zero *overhead* de agendamento (NVIDIA CUDA, 2009). Caso um multiprocessador receba um bloco com mais de 32 *threads* para processar, ele quebra esse bloco em *warps*, agrupando as *threads* de acordo com o seu *thread ID*.

Uma GPU Tesla consiste em um *array* de multiprocessadores de *threads*, conforme pode ser observado na figura 3.2.

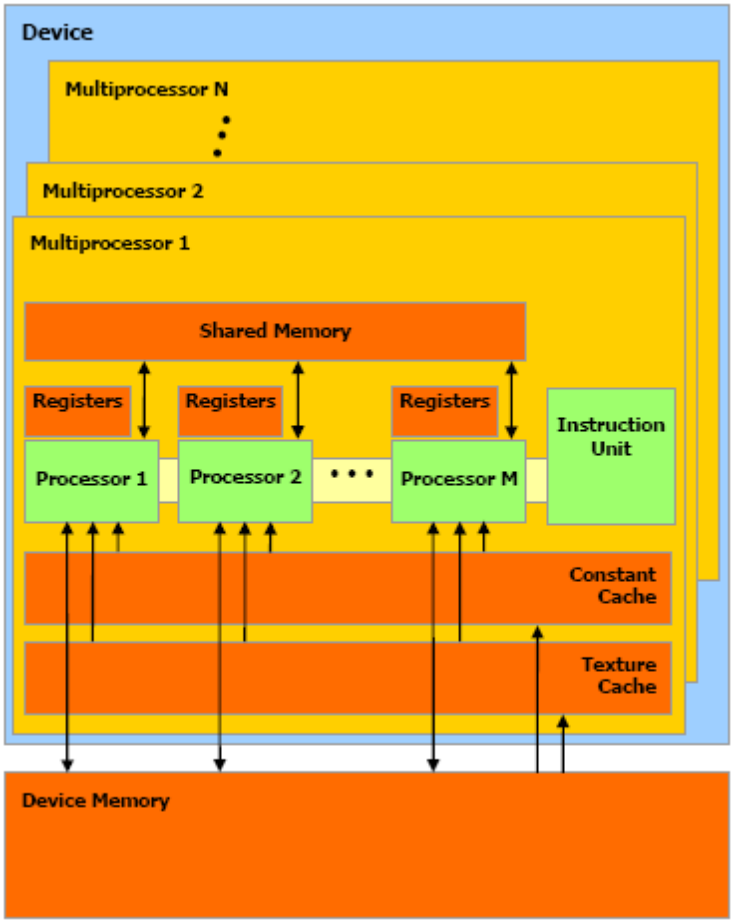

Figura 3.2: A arquitetura da GPU Tesla (NVIDIA CUDA, 2009).

Cada multiprocessador consiste de 8 processadores escalares com 16384 registradores de 32 bits de uso exclusivo (2048 registradores por processador, totalizando 8Kb). Cada multiprocessador possui uma memória compartilhada (chamada de *shared memory*, com 16Kb de tamanho e acesso ultra-rápido), acessível por todas as *threads* de um bloco, além de duas memórias somente-leitura de rápido acesso: a memória de textura (chamada de *texture cache*) e a memória constante (chamada de *constant cache*, de 64Kb). Adicionalmente, é possível acessar a memória principal da

GPU, chamada de *device memory*, onde reside a memória local de cada *thread* e a memória global da aplicação. É a memória com a maior latência de acesso.

É para os multiprocessadores que os blocos de *threads* são mapeados. Cada bloco é dividido em *warps*. Cada multiprocessador pode ter até 32 *warps* ativos, o que totaliza 1024 *threads* concorrentes por multiprocessador. Entretanto, esse número depende da quantidade de memória compartilhada que cada bloco demanda e do número de registradores demandados por cada *thread* (o número máximo de registradores disponíveis por multiprocessador é de 16384), o que pode reduzir o total de *threads* por multiprocessador. Contendo quatro GPUs com 30 multiprocessadores cada, totalizando 960 processadores escalares, a arquitetura Tesla S1070 pode processar milhares de *threads* concorrentemente e representa o estado da arte no que diz respeito às GPUs da Nvidia (NVIDIA TESLA, 2008).

3.2 A API CUDA

Executar um programa escrito em CUDA requer alguns componentes de software e hardware. No que tange ao hardware, uma extensa lista de GPUs habilitadas a executar código CUDA pode ser encontrada em (NVIDIA CUDA, 2009). Uma vez satisfeita essa condição, é necessária a instalação de algum software, qual seja: um *driver* específico e um *toolkit* contendo um compilador e algumas ferramentas adicionais. Ambos são fornecidos pela Nvidia. Eles podem ser obtidos no *website* da platafora (DOWNLOAD CUDA). A figura 3.3 ilustra a pilha de software da plataforma CUDA, composta pelo *driver* de acesso ao hardware, um componente de *runtime* e de duas bibliotecas matemáticas prontas para uso: CUBLAS (*Basic Linear Algebra Subroutines*) e CUFFT (*Fast Fourier Transform*). No topo dessa pilha encontra-se a API da plataforma CUDA.

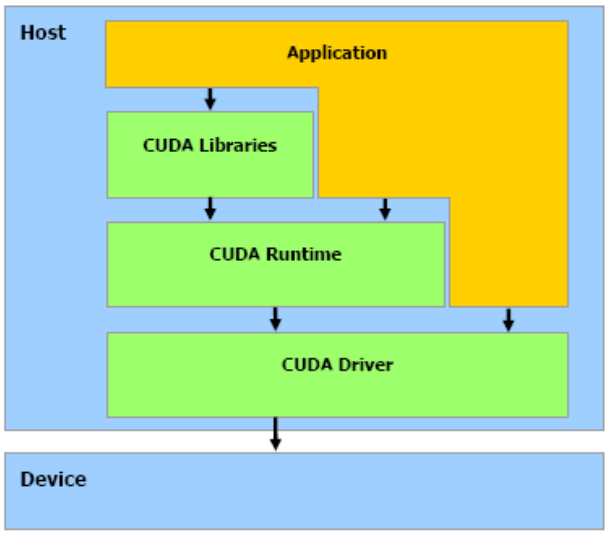

Figura 3.3: A pilha de software da plataforma CUDA (NVIDIA CUDA, 2009).

A plataforma CUDA introduz dois novos conceitos para o escalonamento das *threads*: bloco e grid. É com esses conceitos que se organiza a repartição dos dados entre as *threads*, bem como sua organização e distribuição ao hardware.

3.2.1 Bloco e Grid

Um bloco é a unidade básica de organização das *threads* e de mapeamento para o hardware. Um bloco de *threads* é alocado a um multiprocessador da GPU. Dessa forma,

o tamanho mínimo recomendado a um bloco é de 8 *threads*, caso contrário haverão processadores ociosos. Os blocos podem ter uma, duas ou três dimensões.

O grid é a unidade básica onde estão distribuídos os blocos. O grid é a estrutura completa de distribuição das *threads* que executam uma função. É nele que está definido o número total de blocos e de *threads* que serão criados e gerenciados pela GPU para uma dada função. Um grid pode ter uma ou duas dimensões.

A figura 3.4 ilustra um grid de dimensões 2x3 com blocos de tamanho 3x4.

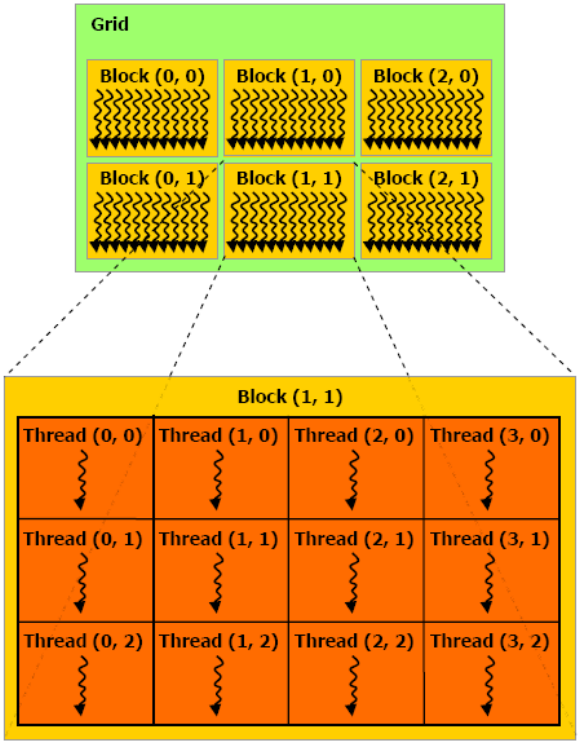

Figura 3.4: Exemplo de um grid com seus blocos (NVIDIA CUDA, 2009).

3.2.2 Extensões CUDA à Linguagem C

A API da plataforma CUDA introduz quatro extensões à linguagem C, quais sejam:

- qualificadores de tipo de função, para definir a unidade lógica de execução do código: CPU ou GPU (seção 3.2.2.1);
- qualificadores de tipo de variável, para definir onde elas serão armazenadas: na CPU ou na GPU (seção 3.2.2.2);
- uma nova sintaxe de chamada de função para configurar os blocos e o grid (seção 3.2.2.3);
- quatro variáveis internas para acessar os índices e dimensões dos blocos, do grid e das *threads* (seção 3.2.2.4).

3.2.2.1 Qualificadores de tipo de função

Existem três qualificadores de tipo de função: __device__, __global__ e __host__.

O qualificador __device__ define uma função que será executada na GPU. Também que ela só pode ser chamada a partir da GPU.

O qualificador __global__ define o que a plataforma CUDA chama de *kernel*, ou seja, uma função que é executada na GPU e é chamada a partir da CPU.

Por fim, o qualificador __host__ define uma função que será executada na CPU e que ela só pode ser chamada a partir da CPU.

3.2.2.2 Qualificadores de tipo de variável

São três os qualificadores de tipo de variáveis: __device__, __constant__ e __shared__.

O qualificador __device__ define uma variável que reside na memória global da GPU. Elas são acessíveis por todas as *threads* de um grid e também a partir da CPU através do uso da biblioteca de *runtime* do CUDA. Seu tempo de vida é o da aplicação.

O qualificador __constant__ difere apenas no fato de que a variável reside no espaço de memória constante da GPU.

Por outro lado, as variáveis __shared__ residem na memória compartilhada da GPU e são acessíveis apenas pelas *threads* de um mesmo bloco. Seu tempo de vida é o do bloco.

3.2.2.3 Nova sintaxe de chamada de função

Para realizar uma chamada de função na GPU é preciso informar no código as dimensões do grid e do bloco. Isso é feito através de uma nova sintaxe na chamada da função. Utiliza-se, entre o nome da função e os argumentos passados a ela, um *array* bidimensional onde constam as dimensões do grid e do bloco, respectivamente. Esse *array* é delimitado pelos caracteres <<< e >>>.

3.2.2.4 Variáveis auxiliares

A plataforma CUDA fornece como forma de particionar os dados o acesso aos índices e dimensões das *threads*, dos blocos e do grid. Para acessar o valor dos índices das *threads* utiliza-se a variável threadIdx, a qual é um vetor de até três dimensões. O acesso a cada dimensão é feito através das componentes x, y e z, como em threadIdx.x, threadIdx.y e threadIdx.z. Cada bloco dentro do grid fornece seu índice identificador, como em blockIdx.x e blockIdx.y. Além disso, os índices das dimensões dos blocos de *threads* são acessíveis através da variável blockDim, como em blockDim.x, blockDim.y e blockDim.z. Por fim, é possível acessar os valores das dimensões do grid através da variável gridDim, como em gridDim.x e gridDim.y.

Tomando-se como exemplo a *thread* (1,2) do bloco (1,1) da figura 3.4, tem-se que:

- threadIdx.x = 1;
- threadIdx.y = 2;
- blockIdx.x = 1;
- blockIdx.y = 1;
- blockDim.x = 4;
- blockDim.y = 3;
- gridDim.x = 3;
- gridDim.y = 2.

3.2.3 Restrições da API

As funções da plataforma CUDA possuem algumas restrições. Dentre elas, tem-se que funções __device__ e __global__ não suportam recursão, não podem declarar variáveis estáticas e não podem ter um número variável de argumentos. Além disso, as funções __device__ não fornecem seu endereço, entretanto, ponteiros para funções __global__ são suportados. As funções __global__ retornam *void*, obrigatoriamente.

Qualquer chamada a uma função __global__ deve especificar a dimensão do grid e dos blocos. Chamadas às funções __global__ são assíncronas, ou seja, a execução continua na CPU mesmo que não tenha terminado na GPU. Os parâmetros de uma função __global__ são passados através da memória compartilhada e estão limitados a 256 bytes.

3.2.4 Síntese: Ilustração do Fluxo de Processamento

Por fim, a figura 8 ilustra o fluxo de processamento de um programa escrito em CUDA. Basicamente, o programa inicia copiando os dados a serem processados da memória RAM para a memória principal da GPU. Após, configura-se um grid e é realizada uma chamada *kernel* para ser executada na GPU. Por fim, os dados resultantes são copiados para a memória RAM para posterior processamento.

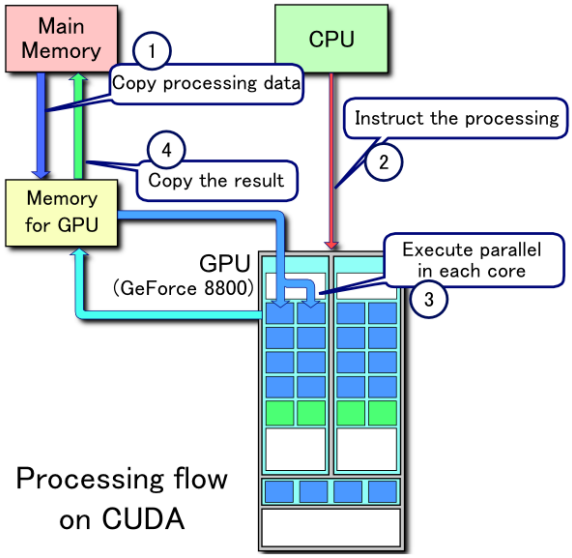

Figura 3.5: Fluxo de execução de um programa CUDA (NVIDIA CUDA, 2009).

Dessa forma, o ciclo de execução de uma aplicação utilizando a tecnologia CUDA alterna entre execuções na CPU e na GPU. É o que ilustra a figura 9.

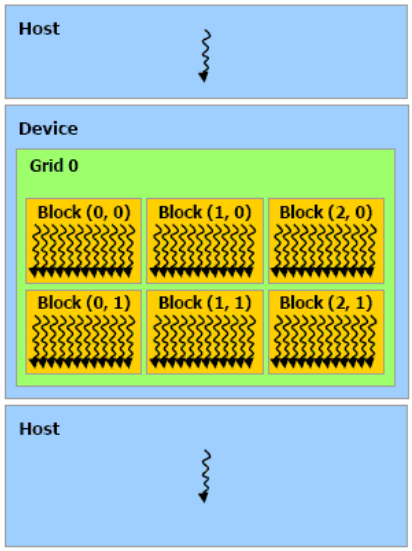

Figura 3.6: Caminho de execução de uma aplicação CUDA (NVIDIA CUDA, 2009).

3.3 Gerenciamento de memória

A tecnologia CUDA não permite a utilização de ponteiros para ponteiros devido ao fato de ser uma operação ilegal derreferenciar um ponteiro do *device* em código *host*, o que resulta em falha de segmentação. Sendo assim, existem duas formas de alocar memória dinamicamente: como memória linear ou CUDA *arrays*. Para a alocação de memória linear utiliza-se uma das seguintes funções: *cudaMalloc*, *cudaMallocPitch*, *cudaMalloc2D* ou *cudaMalloc3D*. Já para a alocação de um CUDA *array* utiliza-se a função *cudaMallocArray*.

Com execeção da função *cudaMalloc*, no que tange à alocação de memória linear, as demais funções efetuam um ajuste dos dados na memória para que esta fique alocada de forma a otimizar o acesso aos dados. Este ajuste significa aumentar o tamanho das linhas da estrutura, preenchendo com zeros os elementos excedentes. Além disso, essas funções retornam um valor chamado *pitch*, o qual deve ser utilizado para acessar o *array* alocado. Cabe ressaltar aqui que o *array* alocado é linearizado na memória, independentemente do número de dimensões que este possua. O código 3.1 abaixo ilusta a utilização desse modelo de alocação de memória.

Código 3.1: Alocando um array bidimensional (NVIDIA CUDA, 2009)

```
1.  float* devPtr;
2.  int pitch;
3.  cudaMallocPitch((void**)&devPtr, &pitch, width * sizeof(float), height);
4.  myKernel<<<100, 512>>>(devPtr, pitch);
5.
6.  __global__ void myKernel(float* devPtr, int pitch) {
7.    for (int r = 0; r < height; ++r) {
8.      float* row = (float*)((char*)devPtr + r * pitch);
9.      for (int c = 0; c < width; ++c) {
10.       float element = row[c];
11.     }
12.   }
13. }
```

Conforme pode ser observado na linha 3, a função *cudaMallocPitch* aloca espaço suficiente para um *array* bidimensional de tamanho *width* * *height* do tipo *float*.

Entretanto, como pode ser observado nas linhas 7 a 10, o acesso aos dados deste *array* é linearizado.

Por outro lado, um CUDA *array* pode ter uma, duas ou três dimensões e seus elementos podem ser inteiros de 8, 16 ou 32 bits ou *floats* de 32 bits. São estruturas otimizadas para leitura a partir da memória de textura. Para utilizá-los é preciso gravar os dados na memória de textura e acessá-las através de uma função cujo retorno é o elemento associado. O código 3.2 ilustra um exemplo de utilização dessa estrutura.

Código 3.2: Utilizando um CUDA array (NVIDIA CUDA, 2009)

```
1. texture<float, 2, cudaReadModeElementType> texRef;
2. cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,
3.                                                 cudaChannelFormatKindFloat);
4. cudaArray* cuArray;
5. cudaMallocArray(&cuArray, &channelDesc, width, height);
6. cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);
7. cudaBindTextureToArray(texRef, cuArray, channelDesc);
8. ...
9. float elemento = tex2D(texRef, x, y);
```

Conforme pode ser observado na listagem acima, para utilizar um CUDA *array* é preciso definir uma textura, linha 1, definir um CUDA *array*, linhas 2 e 4, alocar o CUDA *array*, linha 5, copiar os dados para o CUDA *array*, linha 6, e, por fim, mapear o CUDA *array* para a textura, linha 7. Por fim, o acesso aos dados em uma função *kernel* é realizado através da função *tex2D*, para um *array* bidimensional. Cabe ressaltar o fato de que a memória de textura é apenas-leitura.

Alternativamente, é possível alocar memória estática. Dessa forma, é possível acessar um *array* de várias dimensões utilizando a sintaxe usual de colchetes. O código 3.3 ilustra um exemplo.

Código 3.3: Alocação estática de memória

```
1. typedef struct {
2.   float meuarray[largura][altura];
3. }  blocobidimensional;
4. ...
5. blocobidimensional *bloco;
6. cudaMalloc((void**)&bloco, sizeof(blocobidimensional));
7. ...
8. float elemento = bloco->meuarray[x][y];
```

De acordo com a listagem acima, as linhas 1 a 3 definem a estrutura de dados para conter o *array* bidimensional, as linhas 5 e 6 tratam de definir e alocar espaço para esta estrutura e a linha 8 ilustra como acessar dados dessa estrutura dentro de uma função *kernel*.

3.4 Notação de Ponto Flutuante

Cabe ressaltar que as placas da NVIDIA possuem divergências com relação à norma IEEE-754, a qual padroniza a notação de ponto flutuante. Dentre estas, interessante citar que operações de multiplicação seguidas de adição frequentemente são combinadas em uma única instrução de hardware chamada FMAD. Esta instrução trunca os valores intermediários das multiplicações, o que pode resultar em perda de precisão. Entretanto, a tecnologia CUDA fornece implementações via software, o que resulta em perda de desempenho, para realizar estas operações de forma a seguir o padrão IEEE-754.

3.5 Aspectos de Desempenho

De acordo com (NVIDIA CUDA, 2009), para obter o melhor desempenho da GPU é preciso que o *grid* aloque ao menos um bloco para cada multiprocessador. Adicionalmente, cada bloco deve conter pelo menos 64 *threads*. Entretanto, o acesso aos registradores, que normalmente não demanda nenhum ciclo extra, pode implicar em atrasos devido a dependêcias de *read-after-write* e conflitos de endereçamento. Sendo assim, estes atrasos podem ser ignorados se houverem pelo menos 192 *threads* ativas por multiprocessador. Dessa forma, quando ocorre algum atraso na execução de *warp*, imediatamente outro *warp* é colocado em execução, mantendo a GPU ocupada. É possível manter até 32 *warps* ativos por multiprocessador.

Outro aspecto importante é que o acesso à memória global pode ser otimizado de acordo com determinadas condições. O que ocorre é que o acesso à memória por um *half-warp* (um *half-warp* pode ser as 16 *threads* superiores ou inferiores de um *warp*) pode ser realizado com até uma única instrução de leitura. Para palavras de 32 bits, um *float* por exemplo, o segmento de memória é de 128 bytes, ou 32 *floats*. Sendo assim, quanto mais *threads* de um *half-warp* ativo no multiprocessador acessarem endereços de um mesmo segmento, mais rápida será a sua execução. A escolha do segmento a ser lido segue o seguinte algoritmo:

- Buscar o segmento de memória que contém o endereço requisitado pela *thread* ativa de menor *thread ID*;
- Encontrar todas as outras *threads* ativas que requisitaram dados do mesmo segmento;
- Reduzir o tamanho do segmento, se possível, de 128 bytes para 64 bytes caso todas *threads* acessem apenas a metade superior ou inferior do segmento;
- Reduzir o tamanho do segmento, se possível, de 64 bytes para 32 bytes caso todas *threads* acessem apenas a metade superior ou inferior do segmento;
- Executar a instrução de leitura do segmento e marcar as *threads* servidas como inativas;
- Executar até que todas as *threads* do *half-warp* sejam servidas.

De acordo com o exposto na seção 3.3, a alocação de memória com CUDA é linearizada. Sendo assim, quando todas as *threads* de um *half-warp* requisitarem elementos de uma mesma coluna em sequência, este acesso será otimizado para uma única instrução de leitura da memória, devido ao fato de que os dados requisitados pertencem ao mesmo segmento de memória. Esta é a técnica adequada para se obter altos índices de transferência de dados da memória.

O compilador realiza um esforço a fim de tornar o acesso a memória o mais otimizado possível, entretanto, para se obter um resultado melhor é preciso que o programador organize o código e os dados de forma a facilitar este processo. Para que seja possível organizar o código de forma a obter estes ganhos de desempenho é preciso entender a regra de formação dos *warps*. A regra de formação de um *warp* é a seguinte: dado um bloco com mais de 32 *threads*, estas serão agrupadas em grupos de 32 de acordo com o seu *thread ID*, iniciando com a *thread* de menor *thread ID*, prosseguindo de forma incremental até a *thread* com o maior *thread ID* no bloco. O cálculo do *thread ID* é direto: para blocos unidimensionais, o *thread ID* é o próprio índice da *thread*. Para

um bloco bidimensional de tamanho (*Dx*, *Dy*), o *thread ID* da *thread* de índice (*x*, *y*) é (*x + y * Dx)*.

Sendo assim, para realizar a soma de dois vetores compostos por elementos de 32 bits de tamanho 512 da forma mais eficiente possível, basta alocar um bloco unidimensional de tamanho 512 onde a função kernel que realiza a soma é como segue: *c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x]*. O bloco de 512 *threads* será particionado em 16 *warps* de modo que o primeiro *warp* conterá as *threads* de índice [0, 1, 2, …, 31], o segundo *warp* conterá as *threads* de índice [32, 33, 34, …, 63], e assim por diante. No momento de execução, cada *thread* irá substituir no código o valor da variável *threadIdx.x* pelo seu respectivo índice. Dessa forma, no momento da execução do primeiro *warp* cada *thread* irá executar o seguinte código:

c[0] = a[0] + b[0];

c[1] = a[1] + b[1];

c[2] = a[2] + b[2];

e assim por diante. Quando a *thread* de índice 0 solicita o dado *a[0]*, o algoritmo de acesso à memória é executado. Como as *threads* requisitam os dados *a[0]*, *a[1]*, *a[2]*, ..., *a[31*] e estes encontram-se em um mesmo segmento de memória de 128 bytes, apenas uma instrução de leitura da memória será realizada.

3.6 Exemplos de programas

Nesta seção são apresentados exemplos de programas escritos em CUDA. Como primeiro exemplo, o seguinte código, obtido em (NICKOLLS, 2008), implementa o algoritmo saxpy, que calcula o resultado de $Y = A * X + Y$, dados dois vetores X e Y de tamanho N e um escalar A. Primeiramente, tem-se o código serial desse algoritmo. Após, a sua implementação utilizando CUDA.

Código 3.4: Paralelizando o algoritmo SAXPY

```
1.  void saxpy_serial(int n, float alpha, float *x, float *y) {
2.          for(int i = 0; i<n; ++i)  y[i] = alpha * x[i] + y[i];
3.  }
4.  // Invoke serial SAXPY kernel
5.  saxpy_serial(n, 2.0, x, y);
6.
7.  __global__ void saxpy_parallel(int n, float alpha, float *x, float *y) {
8.          int i = blockIdx.x * blockDim.x + threadIdx.x;
9.          if( i<n ) y[i] = alpha*x[i] + y[i];
10. }
11. // Invoke parallel SAXPY kernel (256 threads per block)
12. int nblocks = (n + 255) / 256;
13. saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Outro exemplo de utilização do CUDA ocorre em algoritmos que utilizam a redução paralela (*parallel reduction*). Esse algoritmo, também obtido em (NICKOLLS, 2008), consiste em, dado um vetor de tamanho N, aplicar em seus elementos uma dada operação, por exemplo, soma ou multiplicação, até reduzi-los a um valor único. Abaixo, segue código que implementa a redução para a operação de soma.

Código 3.5: Redução paralela

```
1.  __global__ void plus_reduce(int *input, unsigned int N, int *total) {
2.    unsigned int tid = threadIdx.x;
```

```
3.    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
4.    __shared__ int x[blocksize];
5.
6.    x[tid] = ((i<N) ? input[i] : 0);
7.    __syncthreads();
8.
9.  for(int s=blockDim.x/2; s>0; s=s/2) {
10.   if(tid < s) x[tid] += x[tid + s];
11.     __syncthreads();
12. }
13.
14.   if (tid == 0) atomicAdd(total, x[tid]);
15. }
```

Uma aplicação interessante para esse algoritmo é o do produto escalar de dois vetores *X* e *Y* de tamanho *N*. O produto escalar é definido como o somatório dos produtos dos elementos de mesmo índice, ou seja, *produto_escalar = X[0] * Y[0] + X[1] * Y[1] + ... + X[n] * Y[n]*. Utilizando CUDA, é possível paralelizar o somatório dos produtos. O código abaixo refere-se à implementação desse algoritmo de produto escalar.

Código 3.6: Produto escalar em CUDA

```
1.  int main(){
2.
3.    cudaMalloc((void**) &d_A, mem_size);
4.    cudaMalloc((void**) &dev, sizeof(unsigned int));
5.
6.    cudaMemcpy(d_A, h_A, mem_size, cudaMemcpyHostToDevice);
7.    cudaMemcpy(d_B, h_B, mem_size, cudaMemcpyHostToDevice);
8.
9.    mult<<<blocos, threads>>>(d_C, d_A, d_B);
10.   soma<<<blocos, threads>>>(d_C, d_Result);
11.
12.   cudaMemcpy(host, dev, sizeof(unsigned int), cudaMemcpyDeviceToHost);
13.   cudaFree(d_A);
14. }
15.
16. __global__ void mult(unsigned int* C, unsigned int* A, unsigned int* B) {
17.   int i = blockDim.x * blockIdx.x + threadIdx.x;
18.   C[i] = A[i] * B[i];
19. }
20.
21. __global__ void soma(unsigned int* C, unsigned int* total) {
22.   unsigned int tid = threadIdx.x;
23.   unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
24.
25.   __shared__ unsigned int x[VECTOR_LENGTH/THREAD_BLOCKS];
26.   x[tid] = C[i];
27.   __syncthreads();
28.
29.   for(int s=blockDim.x/2; s>0; s=s/2) {
30.     if(tid < s) x[tid] += x[tid + s];
31.     __syncthreads();
32.   }
33.
34.   if (tid == 0) atomicAdd(total, x[0]);
35. }
```

Analisando-se esse código é possível perceber o fluxo de execução de um programa CUDA apresentado na figura 3.5. Nas linhas 3 e 4 tem-se a alocação de memória na GPU. Após, nas linhas 6 e 7, tem-se a cópia dos dados para a memória da GPU. Com os dados na GPU parte-se para a execução do código: linhas 9 e 10 . Por fim, copiam-se os dados para a CPU e libera-se a memória da GPU: linhas 12 e 13.

O presente capítulo apresentou a arquitetura das GPUs da Nvidia e a tecnologia CUDA. O próximo capítulo irá apresentar o Método Lattice-Boltzmann (LBM) e sua implementação utilizando a tecnologia CUDA.

4. O PROGRAMA LATTICE-BOLTZMANN

A Dinâmica de Fluidos Computacionais (CFD) é uma importante área de pesquisa tecnológica. Através do estudo das propriedades físicas de líquidos e gases é possível determinar diferentes fenômenos físicos (SCHEPKE). O método Lattice-Boltzmann (LBM) é uma técnica numérica iterativa de modelagem e simulação de fluidos dinâmicos onde espaço, tempo e velocidade são discretos. A figura 4.1 ilustra o laço principal do algoritmo que implementa o método LBM.

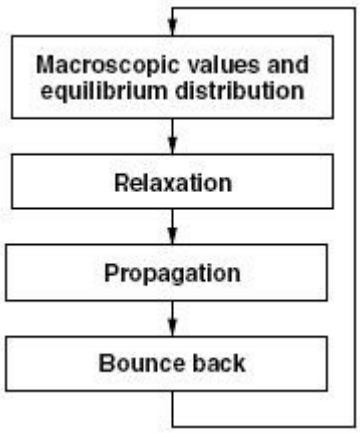

Figura 4.1: O laço principal do método LBM (SCHEPKE).

Esse algoritmo é aplicado a uma estrutura bi ou tridimensional de tamanho 512 x 512, chamada de *lattice*, onde a cada passo da iteração diversos cálculos são efetuados para cada ponto da estrutura. Como os cálculos são realizados tomando-se por base apenas informações do ponto e de seus vizinhos o método é altamente paralelizável. A figura 4.2 ilustra a modelagem de um *lattice* bi e tridimensionais.

Figura 4.2: Modelos de *lattice* (SCHEPKE).

Além disso, é preciso modelar a dinâmica de propagação das partículas, o que é feito através da utilização de um reticuladoA figura 4.3 ilustra modelos de reticulado utilizados em *lattices* bidimensionais.

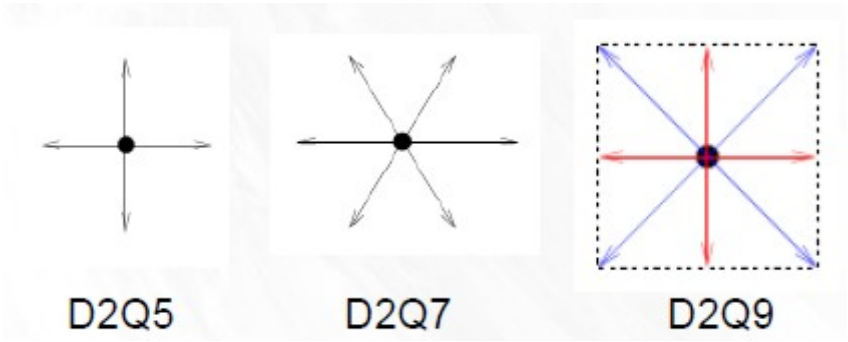

Figura 4.3: Modelos de reticulado bidimensionais (SCHEPKE).

Conforme pode ser observado na figura 4.3, há diversos modelos de reticulado possíves de se utilizar com o LBM. O modelo D2Q9, utilizado na implementação do LBM deste trabalho, modelo oito direções de propagação das partículas mais o ponto estático. Adicionalmente, a figura 4.4 ilustra modelos de reticulado tridimensionais.

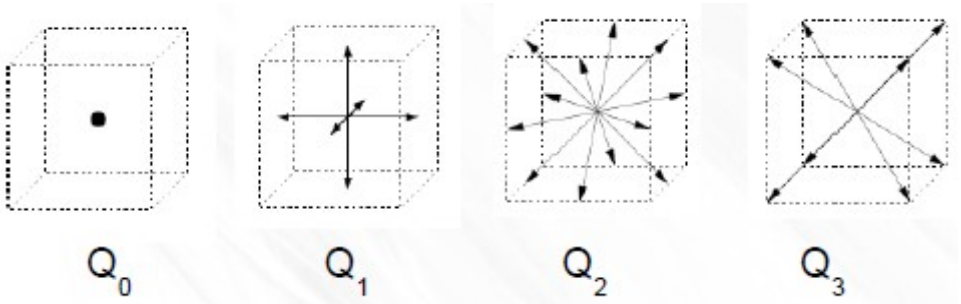

Figura 4.4: Modelos de reticulado tridimensionais (SCHEPKE).

Estes modelos de reticulados são agrupados para dar origem aos modelos D3Q15, por exemplo, o qual é a união dos reticulados q_0, q_1 e q_3. Além desta modelagem das direções de propagação das partículas, é preciso modelar a propagação das partículas propriamente dito, a qual é realizada pela função *propagate*. É o que pode ser observado na figura 4.5.

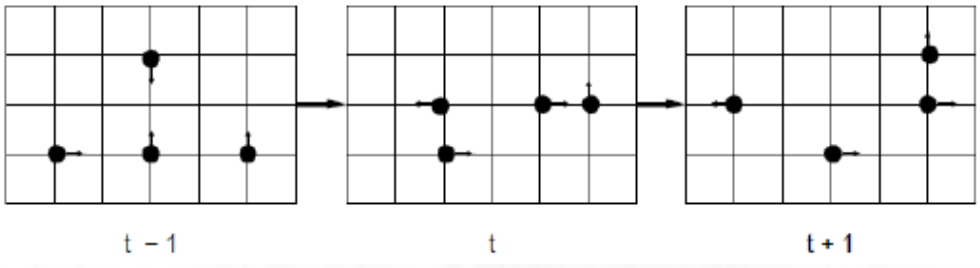

Figura 4.5: Propagação das partículas (SCHEPKE).

Por fim, é preciso modelar o comportamente das partículas pertante os obstáculos, o que é feito pela função *bounceback* do algoritmo LBM. A figura 4.6 ilustra esse processo.

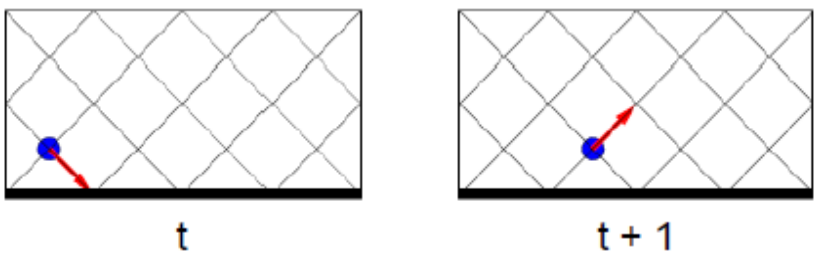

Figura 4.6: Condição de contorno (SCHEPKE).

Este trabalho propõe duas implementações paralelas do método Lattice-Boltzmann, o qual é utilizado na simulação da CFD: uma utilizando a tecnologia CUDA e outra utilizando a tecnologia CUDA associada à tecnologia *Message Passing Interface* (MPI). O uso do MPI permite distribuir a carga de processamento entre diversos nodos de um cluster. O uso das duas tecnologias é apropriado em um cluster de GPUs.

É o que será analisado neste capítulo com a seguinte organização: a seção 4.1 apresenta a implementação LBM utilizando CUDA com uma análise das dificuldades e das facilidades encontradas e a seção 4.2 apresenta uma breve introdução à tecnologia MPI, a implementação do LBM utilizando MPI com CUDA e, por fim, uma análise das dificuldades desta implementação.

4.1 A Implementação CUDA

Para a implementação CUDA do método LBM descrito acima partiu-se de um código sequencial na linguagem C. Este código define uma estrutura de dados especial para guardar as informações referentes ao número máximo de iterações, à densidade do fluido, à aceleração, ao parâmetro de relaxamento e à dimensão linear para o cálculo do número de Reynold. Os valores desses parâmetros são lidos de um arquivo de entrada. Para o código CUDA esta estrutura é alocada na memória da GPU. Inicialmente, os dados são lidos para a memória RAM e após copiados para a GPU. O código 4.1 ilustra o uso das funções CUDA para alocar memória e copiar dados.

Código 4.1: Inicialização da estrutura de parâmetros

```
1. s_properties *GPUproperties;
2. cudaMalloc((void**)&GPUproperties, sizeof(s_properties));
3. cudaMemcpy(GPUproperties, properties, sizeof(s_properties),
   cudaMemcpyHostToDevice);
```

Conforme pode ser observado na linha 1, a alocação de memória em CUDA é realizada através da função *cudaMalloc*. Na linha 2 pode-se observar o uso da função *cudaMemcpy*, que realiza a cópia de dados entre memória RAM e memória da GPU, com o cuidado de utilizar corretamente o quarto parâmetro – *cudaMemcpyHostToDevice* ou *cudaMemcpyDeviceToHost* –, o qual define a direção do fluxo de dados, no caso acima da memória RAM para a memória da GPU. O próximo passo é inicializar a estrutura do *lattice*. O código 4.2 abaixo exibe esta estrutura.

Código 4.2: Definição da estrutura do *lattice*

```
1. typedef struct {
2.   int lx;
3.   int ly;
4.   int n;
5.   bool obst[512][512];
6.   float node[512][512][9];
7.   float temp[512][512][9];
8.   float n_sum;
9. } s_lattice;
```

De acordo com a definição acima pode-se observar que ela contém o número de elementos no eixo *X*, o número de elementos no eixo *Y*, o número de dimensões em cada elemento, o *array* bidimensional de obstáculos, o *array* tridimensional para armazenar as densidades do fluido em cada ponto da *lattice*, um *array* auxiliar para evitar dependência de dados e uma variável *float* para realizar verificações de controle. Esta estrutura é utilizada para representar o modelo de reticulado D2Q9, onde o terceito

elemento do *array* de pontos representa as nove direções de propagação das partículas. Isto pode ser observado na figura 4.7.

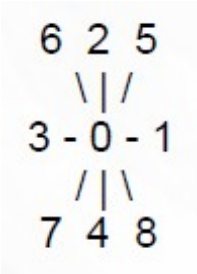

Figura 4.7: Modelando o reticulado D2Q9.

A alocação em memória dessa estrutura varia de acordo com o tamanho do *lattice* utilizado. Alocação de memória para esta estrutura pode ser observada na listagem de código 4.3 abaixo.

Código 4.3: Alocação de memória para o *lattice*

```
1. s_lattice *GPUlattice;
2. cudaMalloc((void**)&GPUlattice, sizeof(s_lattice));
3. cudaMemcpy(GPUlattice, lattice, sizeof(s_lattice), cudaMemcpyHostToDevice);
```

Analisando-se o código acima, observa-se que a linha 1 define o ponteiro para a estrutura do *lattice*, a linha 2 realiza a alocação de memória na GPU e a linha 3 efetua a cópia dos dados da memória RAM para a memória da GPU. Cabe ressaltar aqui que se a estrutura do *lattice* for grande o bastante para não entrar na memória da GPU ele terá que ser particionado em *N* partes. Isso irá impactar na performance da simulação, tendo em vista que um determinado trabalho de entrada e saída terá que ser realizado, transferindo dados da memória da GPU para a memória RAM e vice-versa a cada passo da simulação. Finalizado esse processo inicial de alocação de memória para as estruturas de parâmetros e do *lattice*, o programa está pronto para iniciar a simulação.

O primeiro passo para iniciar a simulação é inicializar os valores iniciais para os nodos do *lattice*. Isso é feito através de uma função de inicialização, a qual percorre todo o *lattice* inicializando os valores e é executada apenas uma vez. Com os valores da estrutura inicializados a simulação pode ser iniciada. Conforme visto no início deste capítulo a simulação do LBM efetua uma série de operações sobre os nodos do *lattice* de forma cíclica. A listagem 4.4 abaixo exibe o laço principal.

Código 4.4: O laço principal da simulação

```
1.  int main() {
2.    ... inicializar estrutura de propriedades do fluido ...
3.    ... inicializar estrutura do lattice ...
4.    dim3 grid(1, 1);
5.    dim3 block(512, 512);
6.    for (iter = 0; iter < max_iter; iter++) {
7.      GPUredistribute<<<grid,block>>>(lattice, properties->accel, properties->density);
8.      cudaThreadSynchronize();
9.      GPUpropagate<<<grid,block>>>(lattice);
10.     cudaThreadSynchronize();
11.     GPUbounceback<<<grid,block>>>(lattice);
12.     cudaThreadSynchronize();
13.     GPUrelaxation<<<grid,block>>>(lattice, properties->density, properties->omega);
14.     cudaThreadSynchronize();
15.   }
16. }
```

Neste trecho de código a função mais importante é a *cudaThreadSynchornize* (linhas 8, 10, 12 e 14), pois é ela que garante a corretude do programa, tendo em vista que as chamadas *kernel* do CUDA são assíncronas, ou seja, o programa principal segue a execução mesmo que a função CUDA não tenha encerrado sua execução. Isso traria problemas de acesso aos dados, pois a cada passo da simulação (linhas 1, 3, 5 e 7) é preciso garantir que os dados tenham sido escritos antes de partir para o passo seguinte. Sendo assim, a chamada *cudaThreadSynchronize* atua como uma barreira da qual o código só continua quando todas as chamadas *kernel* tiverem chego àquele ponto.

Este trecho da simulação não permite a paralelização, tendo em vista que cada passo precisa completar a sua execução antes de avançar. É dentro de cada passo que se encontra a porção paralelizável do algoritmo. Cada função da simulação (linhas 1, 3, 5 e 7) é paralelizada. Os maiores ganhos encontram-se nas função *propagate*, *bounceback* e *relaxation* (a função de *redistribute* atua apenas na primeira coluna do *lattice*, o que demanda no máximo 3072 operações simples de soma ou subtração considerando um *lattice* de 512 x 512 com 9 direções de propagação para cada ponto), as quais atuam sobre todos os pontos do *lattice*. O trecho de código 4.5 ilustra como é feito o processamento no caso do algoritmo sequencial.

Código 4.5: Algoritmo sequencial para a função *redistribute*

```
1.  double t_1 = density * accel / 9.0;
2.  double t_2 = density * accel / 36.0;
3.  for (y=0; y<ly; y++) {
4.    if (l->obst[0][y] == false && ...) {
5.      l->node[0][y][1] += t_1;
6.      l->node[0][y][3] -= t_1;
7.      l->node[0][y][5] += t_2;
8.      l->node[0][y][6] -= t_2;
9.      l->node[0][y][7] -= t_2;
10.     l->node[0][y][8] += t_2;
11.   }
12. }
```

Como pode ser observado na linha 3, a função *redistribute* executa um laço que percorre todas as linhas do *lattice*, atuando em alguns pontos da primeira coluna. Esse laço tem uma complexidade de 512 iterações para o *lattice* 512 x 512. Para paralelizar esta função com CUDA, considerando-se a utilização de um *grid* com apenas um bloco e sendo este bloco formado por 512 *threads*, basta substituir o laço por instruções de cálculo do índice do elemento sobre o qual cada *thread* irá atuar. A listagem 4.6 ilustra essa alteração.

Código 4.6: Paralelizando para a função *redistribute*

```
1.  double t_1 = density * accel / 9.0;
2.  double t_2 = density * accel / 36.0;
3.  int y = threadIdx.y;
4.  if (l->obst[0][y] == false && ...) {
5.    l->node[0][y][1] += t_1;
6.    l->node[0][y][3] -= t_1;
7.    l->node[0][y][5] += t_2;
8.    l->node[0][y][6] -= t_2;
9.    l->node[0][y][7] -= t_2;
10.   l->node[0][y][8] += t_2;
11. }
```

De acordo com o código acima, a linha 3 apresenta a alteração necessária para paralelizar a função *redistribute* com CUDA. Essa alteração reduz a complexidade desta função de 512 para 1.

Além disso, as funções *propagate*, *bounceback* e *relaxation* também são paralelizadas. Suas versões sequenciais são apresentadas na listagem 4.7 abaixo.

Código 4.7: Execução sequencial das funções *propagate*, *bounceback* e *relaxation*

```
1.  ... inicialização de dados ...
2.  for (x = 0; x < l->lx; x++) {
3.    for (y = 0; y < l->ly; y++) {
4.      ... computações diversas ... l->node[x][y] ...
5.    }
6.  }
```

Como pode ser observado o algoritmo sequencial é composto por dois laços que percorrem todos os pontos do *lattice*. Sua complexidade é *lx* * *ly*, onde *lx* representa o número de elementos no eixo *X* e *ly* o número de elementos no eixo *Y*. Para o caso do *lattice* 512 x 512, isso representa 262.144 iterações para cada função do algoritmo. Com a tecnologia CUDA estes laços são quebrados entre as diversas *threads* postas em execução na GPU. No extremo oposto do algoritmo sequencial situa-se o grau máximo de paralelização possível: uma *thread* para cada ponto do *lattice*. A forma como isso é feito em CUDA encontra-se no código 4.8 abaixo.

Código 4.8: Paralelizando as funções *propagate*, *bounceback* e *relaxation*

```
1.  ... inicialização de dados ...
2.  int x = threadIdx.x;
3.  int y = threadIdx.y;
4.  {
5.    ... computações diversas ... l->node[x][y] ...
6.  }
```

Conforme pode ser observado, os dois laços das linhas 2 e 3 da listagem 4.7 foram substituídos por instruções de cálculo do índice do ponto sobre o qual cada *thread* irá atuar. Com esta paralelização, a complexidade de cada função diminui de 262.144 para 1 iteração.

Nesse ponto da utilização da tecnologia CUDA encontra-se um ponto de dificuldade, pois o cálculo dos índices sobre os quais cada bloco e cada *thread* vai atuar precisa ser cuidadosamente realizado para evitar que diferentes *threads* efetuem operações sobre os mesmos pontos do *lattice* ou deixem de efetuar operações sobre determinados pontos. Há também um processo decisório de como alocar os recursos da GPU para a realização dos cálculos, ou seja, é preciso analisar diferentes configurações de *grids* e blocos para melhor utilizar os recursos do hardware disponível. De acordo com (NVIDIA CUDA, 2009) é preciso que haja pelo menos um bloco de *threads* para cada multiprocessador da GPU a fim de melhor utilizar o recurso de hardware disponível. Entretanto, para processamentos em que hajam barreiras de sincronização de *threads* ou leituras de dados da memória da GPU, esse número deve ser de dois ou mais blocos por multiprocessador. Além disso, cada bloco deve possuir um número de *threads* que seja múltiplo de 64, a fim de obter uma melhor utilização do hardware.

Como conclusão desta seção cabe salientar a grande facilidade que a tecnologia CUDA ofereceu para a paralelização do código sequencial do LBM. As únicas mudanças necessárias nas quatro funções essenciais do algoritmo (*redistribute*, *propagate*, *relaxation* e *bounceback*) foram a substituição das duas linhas de código que implementam os dois laços para percorrer todos os pontos do *lattice* (listagem de código

4.6 acima) por duas linhas de código CUDA para o cálculo dos índices do ponto do *lattice* sobre o qual cada *thread* opera. Cabe salientar que esse é o resultado obtido no extremo oposto da operação sequencial, onde há uma *thread* para cada ponto do *lattice*. Entretanto, estima-se que, devido ao grande números de pontos do *lattice* de tamanho 512 x 512 (262.144 pontos), essa não deva ser a configuração ótima no processo de paralelização do LBM devido ao excessivo número de *threads* para gerenciar. Dessa forma, provavelmente o *lattice* será quebrado em um determinado número de *sublattices*, sendo cada um destes alocados a um bloco do *grid*, o qual conterá um determinado número de *threads*, as quais irão operar sobre mais de um ponto do *lattice*.

O capítulo 5 trata da avaliação do desempenho obtido para a simulação LBM utilizando esta tecnologia.

4.2 Um Passo a Mais: Paralelização em Cluster de GPUs

Após o término da implementação CUDA do LBM chegou-se a conclusão de que o trabalho poderia avançar ainda mais na direção da paralelização do código. A implementação CUDA descrita na seção anterior é baseada num hardware de apenas uma máquina com uma única GPU. O passo seguinte da paralelização pode ser obtido de duas formas: duplicando o hardware, ou seja, utilizando-se uma máquina com múltiplas GPUs ou distribuindo a carga de processamento entre diversas máquinas em uma rede, ou seja, utilizando-se um cluster onde cada nodo do cluster está equipado com uma GPU. Para ir ainda mais além, é possível utilizar-se um cluster onde cada nodo deste encontra-se equipado com múltiplas GPUs.

Para que seja possível paralelizar o processamento do LBM em um cluster é preciso utilizar alguma tecnologia que forneça o suporte necessário. Neste trabalho foi utilizada a tecnologia denominada *Message Passing Interface* (MPI). Esta tecnologia é uma extensão à linguagem C, assim como o CUDA. A seção 4.2.1 explica a implementação do LBM utilizando CUDA e MPI e a seção 4.2.2 apresenta dados técnicos referentes à compilação do código utilizando estas duas tecnologias.

4.2.1 A Implementação CUDA e MPI

A implementação do LBM utilizando as tecnologias CUDA e MPI em conjunto deve-se ao fato de elas serem ortogonais entre si. Desta forma, é possível realizar chamadas CUDA dentro de um programa MPI.

Para a implementação CUDA e MPI partiu-se da implementação CUDA descrita na seção anterior e de uma implementação MPI. Basicamente, bastou substituir as chamadas das funções do laço principal por chamadas CUDA. O sistema de paralelização MPI é transparente para o CUDA, o que não exigiu nenhuma adaptação ao código MPI no que diz respeito à distribuição da carga entre os diversos nodos do cluster. Cabe ressaltar aqui mais uma vez a facilidade que a utilização da tecnologia CUDA proporciona ao programador, não exigindo nenhum código de controle e gerenciamento da paralelização no que diz respeito ao processo de criação, destruição e gerenciamento de *threads*. Apenas a definição do *grid* e do bloco CUDA é exigida do programador, além da correta utilização das variáveis auxiliares descritas na subseção 3.2.2.4 para o acesso aos dados.

A diferença fundamental existente na implementação MPI é a necessidade de comunicação entre os diversos processos distribuídos no cluster. Devido ao fato de que a implementação MPI particiona o *lattice* em *N* partes e os distribuí entre os nodos do

cluster, é preciso ter o cuidado de sincronizar as informações para obter o resultado correto da computação, em especial nas regiões de fronteira, pois a função *propagate* utiliza informações dos vizinhos de cada nodo. Desta forma, é preciso definir o *sublattice* de cada nodo de forma a conter as informações necessárias ao processamento. A figura 4.8 ilustra os fluxos de comunicação entre os diversos processos (cada processador na figura representa um nodo distinto do cluster).

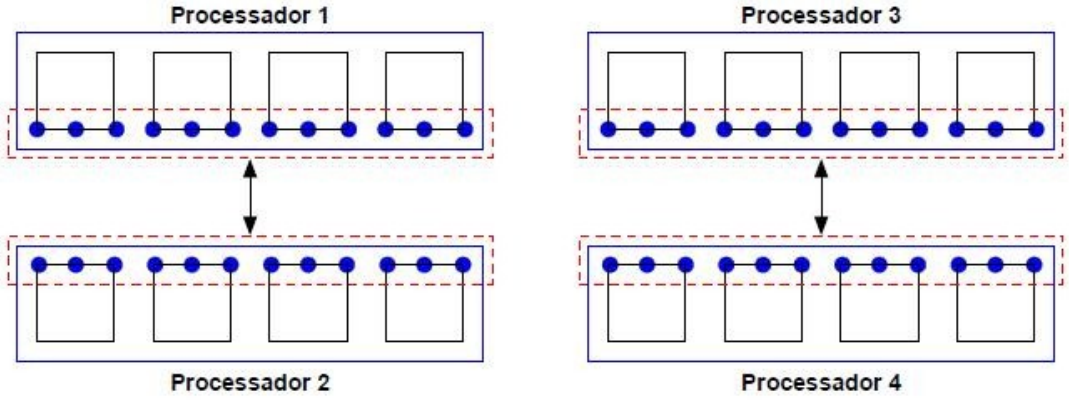

Figura 4.8: Comunicação entre processos (SCHEPKE).

Essa comunicação é realizada após o passo de propagação através do uso de uma função de sincronização. Dado que a estrutura do *lattice* é repartida entre os diversos processos MPI, é preciso redefinir a estrutura de dados correspondente. A listagem 4.9 ilustra a nova estrutura.

Código 4.9: Estrutura de dados para abrigar o *lattice*

```
1. typedef struct {
2.    int lx_first, lx_last, ly_first, ly_last, lx, ly, lx_total, ly_total;
3.    bool obst[lx_total / lx][ly_total / ly];
4.    float node[lx_total / lx][ly_total / ly][9];
5. } sublattice;
```

Além disso, é preciso criar uma estrutura para conter os comunicadores, conforme pode ser observado na listagem 4.10 abaixo.

Código 4.10: Estrutura de comunicadores

```
1. typedef struct {
2.    MPI_Comm grid_comm;
3.    int myid;
4.    int myid_x, myid_y;
5.    int proc_x, proc_y;
6. } comm;
```

Cada processo MPI conterá uma cópia privada desta estrutura, sendo que o comunicador *grid_comm* é igual para todos os processos. Além disso, é preciso inicializar esta estrutura. O código 4.11 ilustra esse processo, considerando que no momento da execução do programa MPI são passados dois argumentos: em quantas partes será quebrado o *lattice*, tanto vertical quanto horizontalmente. Estes argumentos são passados para a função de inicialização dos comunicadores através das variáveis *diim1* e *dim2*.

Código 4.11: Inicializando os comunicadores

```
1. initialize_comm(dim1, dim2, comm) {
2.    int dimensions[2];
3.    dimensions[0] = atoi(dim1);
4.    dimensions[1] = atoi(dim2);
```

```
5.    ...
6.    MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, wrap_around, reorder,
7.  &comm->grid_comm);
8.    comm->proc_x = dimensions[0];
9.    comm->proc_y = dimensions[1];
10.   ...
11.   MPI_Cart_coords(comm->grid_comm, comm->myid, 2, coordinates);
12.   comm->myid_x = coordinates[0];
13.   comm->myid_y = coordinates[1];
14. }
```

Conforme pode ser observado, a estrutura de comunicadores é inicializada através da função *MPI_Cart_create*, a qual cria uma estrutura cartesiana, semelhando ao *grid* da implementação CUDA, para conter os processos MPI. Além disso, esses processos são inseridos no comunicador global: MPI_COMM_WORLD. Por fim, é realizada uma chamada à função *MPI_Cart_coords* para guardar o *rank* de cada processo MPI na variável *comm->myid*. Isso será importante para a comunicação entre os processos no decorrer da execução da simulação.

Uma vez inicializadas as estruturas básicas, o algoritmo está pronto para execução. O código 4.12 descreve o laço principal da implementação CUDA e MPI com enfoque na função de sincronização.

Código 4.12: A paralelização com CUDA e MPI

```
1.  int main() {
2.    … inicializar estrutura de comunicadores ...
3.    … inicializar estrutura de propriedades do fluido ...
4.    … inicializar estrutura do lattice …
5.    dim3 grid(1,1);
6.    dim3 block(lattice->lx_total / comm->proc_x, lattice->ly_total / comm->proc_y);
7.    for (iter = 0; iter < max_iter; max_iter++) {
8.      if (comm->myid_x == 0) {
9.        GPUredistribute<<<grid,block>>>(GPUlattice, prop->accel, prop->density);
10.     }
11.     GPUpropagate<<<grid,block>>>(GPUlattice);
12.     sincronization(lattice, comm);
13.     GPUbounceback<<<grid,block>>>(GPUlattice);
14.     GPUrelaxation<<<grid,block>>>(GPUlattice, prop->density, prop->omega);
15.   }
16. }
```

Neste ponto da implementação encontra-se a primeira dificuldade para o uso do CUDA com MPI: a função de sincronização. Na implementação MPI pura, o bloco de dados de 512 x 512 é dividido em *N* partes verticais e *M* partes horizontais. Supondo uma divisão em 2 partes verticais e duas horizontais, temos 4 subblocos de 256 x 256 cada. Cada subbloco representa um pedaço do *lattice* e é distribuído para um nodo do cluster. Após a função de propagação, os nodos precisam comunicar uns aos outros o resultado de seus processamentos. Esta é a tarefa que a função de sincronização realiza.

A diferença chave aqui é a localização dos dados na memória. A aplicação MPI tem os dados do *lattice* alocados na memória RAM, o que torna imediato o acesso a estes para efetuar a sincronização. Entretanto, na aplicação CUDA este dados encontram-se na memória da GPU. Como os dados na memória da GPU são acessíveis apenas ao processador da GPU é necessário copiá-los para a memória RAM antes de efetuar a sincronização. Além disso, após o término da sincronização é necessário copiá-los de

volta à memória da GPU. O código 4.13 ilustra o trecho de código que realiza esta operação.

Código 4.13: Cópia de dados entre GPU e memória RAM

```
1.  for (iter = 0; iter < max_iter; max_iter++) {
2.    redistribute<<<grid, block>>>(lattice, prop->accel, prop->density);
3.    propagate<<<grid, block>>>(lattice);
4.
5.    cudaMemcpy(lattice, GPUlattice, cudaMemcpyDeviceToHost);
6.    sincronization(lattice, comm);
7.    cudaMemcpy(GPUlattice, lattice, cudaMemcpyHostToDevice);
8.
9.    bounceback<<<grid, block>>>(lattice);
10.   relaxation<<<grid, block>>>(lattice, prop->density, prop->omega);
11. }
```

Conforme pode ser observado na listagem acima, as linhas 5 e 7 realizam a cópia do *sublattice* para a memória RAM e, após a sincronização, novamente para a GPU. Esta cópia é necessária pelo fato de que não é possível acessar os dados contidos na memória da GPU diretamente. O listagem 4.14 ilustra o processo de sincronização.

Código 4.14: Sincronização MPI

```
1.  sincronization() {
2.    ...
3.    if(comm->myid_x%2 == 0){
4.      send_neg_x(lattice, comm);
5.      recv_neg_x(lattice, comm);
6.      send_pos_x(lattice, comm);
7.      recv_pos_x(lattice, comm);
8.    } else {
9.      recv_neg_x(lattice, comm);
10.     send_neg_x(lattice, comm);
11.     recv_pos_x(lattice, comm);
12.     send_pos_x(lattice, comm);
13.   }
14.
15.   ... idem para eixo y ...
16. }
```

De acordo com a listagem acima, as linhas 4 e 5 realizam chamadas a outras funções, as quais irão realizar o envio ou o recebimento dos dados através da rede. A listagem 4.15 ilustra o conteúdo de uma função de envio de dados.

Código 4.15: Enviando dados da borda

```
1.  send_neg_x(lattice, comm) {
2.    ...
3.    buffer[k++] = lattice->node[0][y][3];
4.    buffer[k++] = lattice->node[0][y][6];
5.    buffer[k++] = lattice->node[0][y][7];
6.    ...
7.    send_border(buffer, buffer_size, coord, 2, comm->grid_comm);
8.    ...
9.  }
```

Conforme pode ser observado, a função de envio coleta os dados da borda do *sublattice* e os agrega em um *buffer*, o qual será enviado pela rede através de uma chamada à função *MPI_Send*. O *MPI_Send* é realizado através da função *send_border*, a qual é chamada na linha 7. O código 4.16 ilustra o conteúdo dessa função.

Código 4.16: Colocando os dados na rede

```
1. send_border(buffer, coord) {
2.   …
3.   MPI_Cart_rank(grid_comm, coord, &neighbour);
4.   MPI_Send(buffer, buffer_size, MPI_FLOAT, neighbour, label, grid_comm, &request);
5.   ...
6. }
```

Conforme pode ser observado, a linha 3 realiza uma chamada à função *MPI_Cart_rank* a fim de obter o *rank* do processo vizinho, para o qual deverão ser enviados os dados. Por fim, a função *MPI_Send* envia os dados correspondentes através da rede.

É este processo de transferência de dados entre GPU e memória RAM o gargalo da implementação CUDA e MPI. O desempenho obtido com o uso destas duas tecnologias em conjunto será analisado no capítulo 5.

4.2.2 Compilando o Código

Para a compilação do código CUDA e MPI foi utilizado o compilador *nvcc*, o qual é fornecido pela Nvidia no pacote CUDA, juntamente com a biblioteca MPICH, disponível na internet. Todo este conjunto de software foi instalado em um sistema Linux, utilizando a distribuição OpenSuSE. O trecho de código abaixo descreve o conteúdo do arquivo *Makefile* utilizado na compilação do código.

Código 4.17: Arquivo *makefile* para CUDA e MPI

```
1. CC=nvcc
2. FLAGS=-lmpich -lm -I/opt/mpich/include -L/opt/mpich/ch-p4/lib
3. INPUT=lb.cu comm.cu main.cu
4. lb: lb.cu comm.cu main.cu
5. $(CC) -o lb $(INPUT) $(FLAGS)
```

Uma vez apresentado o LBM e sua implementação CUDA, o próximo capítulo irá apresentar os resultados práticos obtidos.

5. AVALIAÇÃO DE DESEMPENHO

Neste capítulo são analisados os diversos aspectos relevantes aos dados obtidos na simulação do LBM utilizando CUDA. O desempenho da implementação CUDA e MPI não foi medido devido à indisponibilidade de um cluster de GPUs.

Para a avaliação do desempenho do LBM foi utilizada uma máquina com processador Intel Core 2 Duo E8500 (3,16 GHz, FSB 1333 MHz, 6 MB Cache) para a obtenção da métrica sequencial. Já para a versão paralela foi utilizada uma GPU Nvidia XFX GeForce GTX 280 1024MB DDR3 XXX (GX-280N-ZDDU) (670 MHz, 1024MB DDR3@2500MHz), a qual possui 30 multiprocessadores de *threads*, cada qual com 8 processadores escalares, o que resulta em 240 processadores.

A seção 5.1 trata da análise da corretude da versão paralela e a seção 5.2 ilustra o ganho obtido com esta versão em relação à versão sequencial.

5.1 Verificando a Corretude da Impementação CUDA

A verificação da corretude da implementação CUDA deu-se através da comparação dos arquivos de saída gerados utilizando-se o programa *diff*. Este programa compara linha a linha os dois arquivos de entrada recebidos e gera como saída um arquivo identificando em quais linhas houve divergência. A implementação paralela foi considerada correta por apresentar apenas diferenças de precisão a partir da quinta casa decimal quando comparada com a saída da implementação sequencial. Cabe ressaltar que já eram esperadas divergências de precisão devido aos motivos elencados na seção 3.4 deste documento, tendo em vista que optou-se por utilizar a implementação de hardware da notação de ponto flutuante a fim de obter o máximo desempenho possível.

Não foram detectadas divergências em nenhuma saída das diversas configurações de *grids* e blocos da implementação paralela testadas.

5.2 Ganho de Desempenho Obtido com a Tecnologia CUDA

O método de avaliação do desempenho da implementação CUDA do LBM foi o tempo necessário ao completo processamento da simulação. A versão sequencial executada no hardware descrito no início deste capítulo necessitou de 27.520 segundos para concluir a sua execução.

Para a implementação CUDA, diversas configurações de *grid* foram testadas. Importante salientar as informações contidas na seção 3.5 deste documento, referentes a como obter o melhor desempenho da GPU. De posse dessas informações, foram avaliadas primeiramente configurações quadradas tanto de *grid* quanto de bloco. Após, configurações unidimensionais de *grid* e bloco também foram analisadas. A figura 5.1

ilustra o desempenho obtido com a configuração de bloco 1 x 512 utilizando-se diversos tamanhos de *grid*.

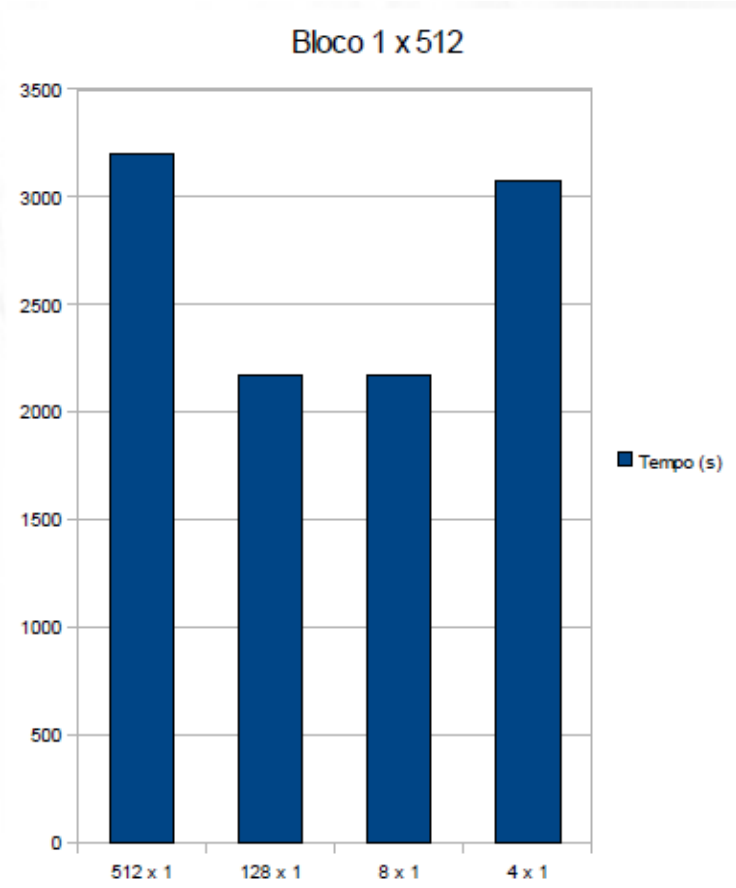

Figura 5.1: Desempenho para bloco 1 x 512.

A figura 5.2 ilustra a distribuição dos *warps* através dos multiprocessadores para cada configuração de *grid* utilizada.

	Warps por multiprocessador
512 x 1	273
128 x 1	68.2
8 x 1	4.2
4 x 1	2.1

Figura 5.2: Distribuição de *warps* para bloco 1 x 512.

A figura 5.3 ilustra os desempenhos obtidos com a utilização de um bloco de tamanho 256 x 1 em diversos *grids*.

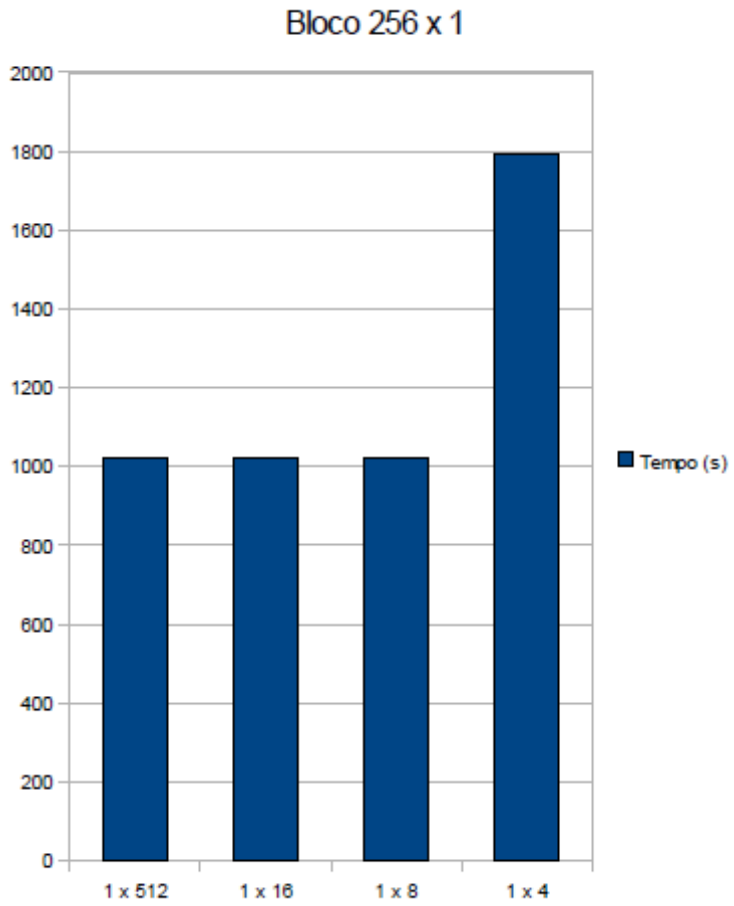

Figura 5.3: Desempenho para bloco 256 x 1.

A distribuição dos *warps* para essas configurações pode ser observado na figura 5.4.

	Warps por multiprocessador
1 x 512	136.5
1 x 16	4.2
1 x 8	2.1
1 x 4	1

Figura 5.4: Distribuição de *warps* para bloco 256 x 1.

Adicionalmente, foram realizados experimentos com configuração de bloco 512 x 1. A figura 5.5 ilustra os desempenhos obtidos.

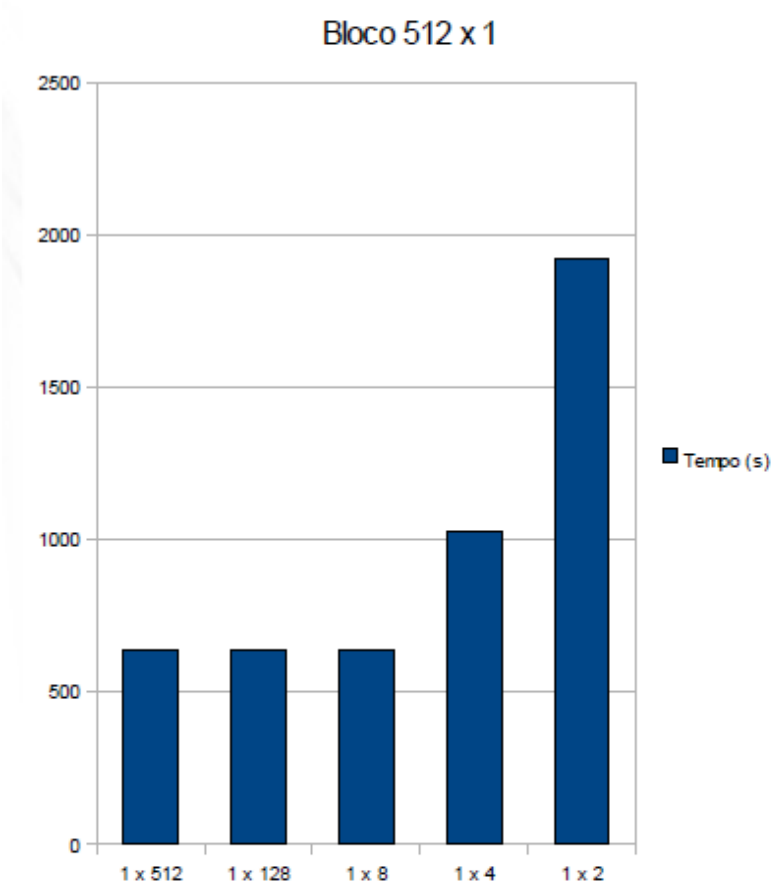

Figura 5.5: Desempenho para bloco 512 x 1.

A distribuição dos *warps* para essas configurações pode ser observado na figura 5.6.

	Warps por multiprocessador
1 x 512	273
1 x 128	68.2
1 x 8	4.2
1 x 4	2.1
1 x 2	1

Figura 5.6: Distribuição de *warps* para bloco 512 x 1.

De acordo com os dados das figuras 5.1, 5.3 e 5.5 acima, o melhor desempenho CUDA foi obtido com a configuração de bloco utilizando 512 *threads* dispostas ao longo do eixo *X*. Cabe ressaltar que, dentre estas configurações, o *grid* precisa conter no mínimo 8 blocos de *threads* dispostas no eixo *Y*. Outro dado importante obtido nos experimentos de laboratório foi o fato do desempenho para blocos com *threads* dispostas no eixo *Y* ser significativamente inferior. Adicionalmente, ao reduzir o

número de *threads* no eixo *Y*, houve perda de desempenho. Essa queda de desempenho decorre essencialmente da latência de acesso à memória global, conforme exposto na seção 3.5. A distribuição dos dados na memória principal obedece à sequência crescente em função do eixo z, y e x, respectivamente. A figura 5.7 ilustra essa distribuição.

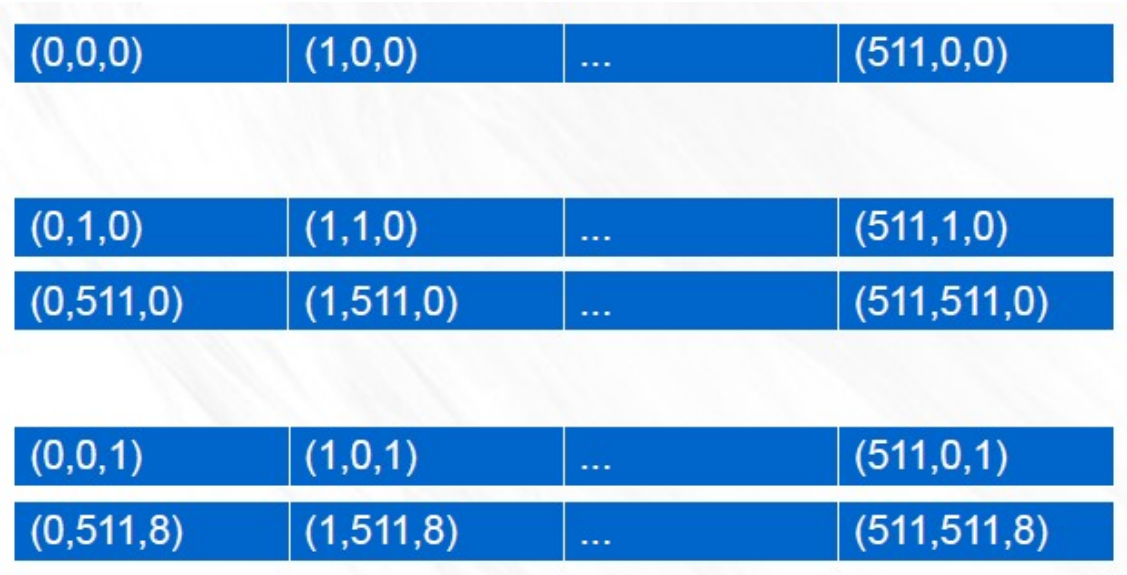

Figura 5.7: Distribuição dos dados na memória.

5.3 Avaliando a utilização da tecnologia CUDA

Esta seção trata de avaliar, comparativamente à norma Posix Threads e ao OpenMP, a facilidade da utilização da tecnologia CUDA. O código 5.1 abaixo ilustra o esforço de programação necessário para criar um programa que soma dois vetores *A* e *B* com 512 elementos cada, considerando a criação de 512 *threads*.

Código 5.1: Somando dois vetores com Posix Threads

```
1.  #include <pthread.h>
2.  int a[512], b[512], c[512];
3.
4.  void soma_vetor(void *pos) {
5.    int index = (int*)pos;
6.    c[index] = a[index] + b[index];
7.  }
8.
9.  int main() {
10.   ... inicializa vetores a e b ...
11.   pthread_t threads[512];
12.   for (i = 0; i < 512; i++) {
13.     pthread_create(&threads[i], null, soma_vetor, (void *)i);
14.   }
15.   for (i = 0; i < 512; i++) {
16.     pthread_join(threads[i], null);
17.   }
18.   pthread_exit();
19. }
```

Conforme pode ser observado nas linhas 12 a 14, é preciso realizar 512 chamadas à função *pthread_create* para criar 512 *threads*. Além disso, as linhas 15 a 17 ilustram a necessidade de sincronização das *threads*.

O mesmo programa escrito em OpenMP é demonstrado no código 5.2 abaixo.

Código 5.2: Somando dois vetores com OpenMP

```
1.  #include <omp.h>
2.  int a[512], b[512], c[512];
3.
4.  int main() {
5.    ... inicializa vetores a e b ...
6.    omp_set_num_threads(512);
7.    #pragma omp parallel for
8.    for (i = 0; i < 512; i++) {
9.      c[i] = a[i] + b[i];
10.   }
11. }
```

Conforme pode ser observado, a complexidade de progração é inferior em relação à norma Posix Threads, pois não há a necessidade de preocupar-se com a criação das *threads*, bastando definir a quantidade desejada, linha 6, e definindo a região paralela, linha 7 a 10. Paralelamente, em relação à tecnologia CUDA, esta complexidade também é inferior. É o que pode ser observado no código 5.3 abaixo.

Código 5.3: Somando dois vetores com CUDA

```
1.  __global__ my_kernel(int *a, *b, *c) {
2.    int index = threadIdx.x;
3.    c[i] = a[i] + b[i];
4.  }
5.
6.  int main() {
7.    int a[512], b[512], c[512];
8.    ... inicializa vetores a e b ...
9.    int *a_gpu, *b_gpu, *c_gpu;
10.   cudaMalloc((void **)&a_gpu, sizeof(int)*512);
11.   ... o mesmo para b_gpu e c_gpu ...
12.   cudaMemcpy(a_gpu, a, sizeof(int)*512, cudaMemcpyHostToDevice);
13.   ... o mesmo para b ...
14.   my_kernel<<<1, 512>>>(a_gpu, b_gpu, c_gpu);
15.   cudaMemcpy(c, c_gpu, sizeof(int)*512, cudaMemcpyDeviceToHost);
16. }
```

Conforme pode ser observado na código 5.3, a tecnologia CUDA exige do programador um trabalho adicional de alocação de memória, linha 10, e de cópia dos dados enter CPU ↔ GPU e vice-versa, linhas 12 e 15. Entretanto, a criação das *threads* é transparente para o programador, bastando definir a estrutura do *grid* e do bloco, linha 14. O ponto positivo a favor da tecnologia CUDA é o número de cores da GPU Tesla, a qual possui 240 cores.

Após ter analisado os resultados práticos obtidos para a execução do LBM utilizando a tecnologia CUDA, o próximo capítulo irá apresentar a conclusão deste trabalho.

6. CONCLUSÃO

De acordo com o conteúdo deste documento, é possível concluir que a percurso trilhado neste trabalho teve como primeiro passo um rápido estudo a respeito da programação concorrente, passando pela conceituação de *threads*, pelo estudo da norma POSIX Threads, de noções de OpenMP e do padrão MPI. Uma vez contextualizado o tema, partiu-se para o estudo da tecnologia CUDA. Sobre esta tecnologia foram estudados aspectos de hardware e software com vistas a sua melhor utilização. De posse desse conhecimento, o passo seguinte foi realizar uma implementação paralela de um algoritmo bem conhecido: o Método Lattice-Boltzmann (LBM). Por fim, após a realização de diversos experimentos, os resultados obtidos foram analisados e comentados.

Como resultado prático obtido neste trabalho tem-se a otimização do tempo de execução do LBM em relação à versão sequencial, a qual necessitou de 27.520 segundos para executar em um Intel Core 2 Duo E8500 (3,16 GHz, FSB 1333 MHz, 6 MB Cache). O melhor resultado obtido na versão CUDA do algoritmo foi de 640 seegundos. Importante ressaltar que este resultado foi obtido a um custo de perda de precisão devidos aos fatores limitantes em relação à notação de ponto flutuante implementada pelo hardware da Nvidia, conforme exposto na seção 3.4 deste documento.

No que diz respeito às limitações deste trabalho, é importante elencar o fato de que o objeto de estudo limitou-se ao *lattice* D2Q9, ou seja, modelo bidimensional com oito direções de propagação das partículas. Modelos de *lattice* tridimensionais, por exemplo o D3Q19, certamente aumentarão significativamente a complexidade do problema, especialmente pelo fato de que não será possível alocar espaço em memória suficiente para todo o *lattice*. Desta forma, será necessário realizar um esforço de transferência de dados entre GPU e CPU, o que certamente impactará no desempenho da simulação.

Considero importante salientar que os resultados obtidos neste trabalho não esgotam as possibilidades de exploração da tecnologia CUDA. Ele foi apenas um primeiro passo na incessante busca por desempenho computacional. A utilização de máquinas multiprocessadas – com múltiplas GPUs – representa um enorme potencial a ser explorado. Adicionalmente, pensando em ir mais além, a utilização do CUDA com a tecnologia MPI me parece bastante promissora. Certamente, este é um caminho a ser explorado por trabalhos futuros.

Como conclusão deste trabalho gostaria de dizer que estou extremamente satisfeito com os resultados obtidos, sejam em níveis teóricos, sejam em níveis práticos. Obtive um enorme aprendizado sobre como construir um documento completo a fim de apresentar um determinado trabalho prático realizado ao longo de um período considerável. Paralelamente, é muito gratificante chegar ao final do curso de Ciência da Computação com a certeza de ter adquirido a habilidade de expandir meus

conhecimentos teóricos e práticos de forma autônoma. O estudo de uma tecnologia completamente nova e não vista em nenhum momento durante o curso – o CUDA – mostrou-se perfeitamente palpável. Os conhecimentos adquiridos sobre hardware e software ao longo do curso mostraram-se extremamente necessários ao processo de aquisição de conhecimento. O estudo sobre estrutura de dados, organização de computadores, paralelismo de dados, dentre tantos outros, foi fundamental na minha capacitação. Sem esse conhecimento adquirido ao longo do curso, certamente encontraria muitas dificuldades na utilização da tecnologia CUDA.

7. BIBLIOGRAFIA

BARNEY, B. Posix Threads Programming. **Lawrence Livermore National Laboratory**, February 2009. Disponível em: <http://computing.llnl.gov/tutorials/pthreads>. Acesso em: fev. 2009.

BERILLO, A.. NVIDIA CUDA. Non-graphic computing with graphics processors. [S.l.]: **iXBT Labs**, October 2008. Disponível em: <http://www.digit-life.com/articles3/video/cuda-1-p1.html>. Acesso em: nov. 2008.

BOGGAN, S.; PRESSEL, D. M. GPUs: an emerging platform for general-purpose computation. [S.l.]: **Army Research Lab**, August 2007. Disponível em: <http://www.arl.army.mil/arlreports/2007/ARL-SR-154.pdf>. Acesso em: nov. 2008.

CUDA for GPU Computing. [S.l.]: **NVIDIA**, February 2007. Disponível em: <http://news.developer.nvidia.com/2007/02/cuda_for_gpu_co.html>. Acesso em: nov. 2008.

DOWNLOAD CUDA Code: complete and free toolkit for creating derivative works. [S.l.]: **NVIDIA**. Disponível em: <http://www.nvidia.com/object/cuda_get.html>. Acesso em: out. 2008.

DREPPER, U.; MOLNAR, I. The Native POSIX Thread Library for Linux. [S.l.: s.n.], February 2005. Disponível em: <http://people.redhat.com/drepper/nptl-design.pdf>. Acesso em: nov. 2008.

FLYNN, L. J. Intel Halts Development of 2 New Microprocessors. [S.l.]: **The New York Times**, May 2004. Disponível em: <http://www.nytimes.com/2004/05/08/business/08chip.html>. Acesso em: nov. 2008.

FOSTER, Ian T. **Designing and building parallel programs : concepts and tools for parallel software engineering**. Reading: Addison-Wesley, 1995. 379 p.

GALLINA, Leandro Z. **Avaliação de Desempenho do OpenMP em Arquiteturas Paralelas**. 2006. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

HALFHILL, T. R. Parallel Processing With CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading. [S.l.]: **Microprocessor Report**, January 2008. Disponível em: <http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf>. Acesso em: ago. 2008.

INTEL Microprocessor Quick Reference Guide: Product Family. [S.l.: s.n.]. Disponível em: <http://www.intel.com/pressroom/kits/quickrefyr.htm>. Acesso em: nov. 2008.

LOVE, R. Introducing the 2.6 Kernel. [S.l.]: **Linux Journal**, May 2003. Disponível em: <http://www.linuxjournal.com/article/6530>. Acesso em: nov. 2008.

MOR, Stéfano D. K. **Emprego da Técnica de *Workstealing*: Estudo de Caso com o Problema da Mochila e MPI**. 2007. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

MPICH. **A portable implementation of MPI**. [S. l.: s. n.]. Disponível em: <http://www.mcs.anl.gov/research/projects/mpi/mpich1/>. Acesso em: abr. 2009.

NICKOLLS, J. et al. Scalable Parallel Programming With CUDA. [S.l.]: **ACM Queue**. March/April 2008. Disponível em: <http://www.cs.stevens.edu/~spock/cs385/CUDA-Concepts.pdf>. Acesso em: nov. 2008.

NVIDIA CUDA Programming Guide. [S.l.]: **NVIDIA**, v2.2, February 2009. Disponível em: <http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf>. Acesso em: jun. 2009.

NVIDIA TESLA S1070 Datasheet. [S.l.]: **NVIDIA**, June 2008. Disponível em: <http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_S1070_US_Jun08_NV_LR_Final.pdf>. Acesso em: nov. 2008.

OLIVEIRA, Rômulo S. de; CARÍSSIMI, Alexandre da S.; TOSCANI, Simão S. **Sistemas Operacionais**. 2 ed. Porto Alegre: Sagra Luzzato, 2001.

SCHEPKE, C.; Maillard, N. **Performance Improvement of the Parallel Lattice Boltzmann Method Through Blocked Data Distributions**. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SNIR, M.; OTTO, S. W.; HUSS-LEDERMAN, S.; WALKER, D. W.; DONGARRA, J. J. **Mpi: the complete reference**. Cambridge: Mit Press, c1996. 336 p.

TANENBAUM, A. S. **Organização Estruturada de Computadores**. Rio de Janeiro: Livros Técnicos e Científicos Editora, 2001.

TREVISAN, J. **Programação Paralela Utilizando MPI: aspectos práticos**. 2005. GRUCAD, UFSC, Florianópolis. Disponível em: <http://www.grucad.ufsc.br/julio/doc/mpi.htm>. Acesso em: fev. 2009.