JOSÉ AURIMAR SEPKA JÚNIOR

EXTENSÃO DO FRAMEWORK WALBERLA PARA USO DE GPU EM SIMULAÇÕES DO MÉTODO DE LATTICE BOLTZMANN

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2015

JOSÉ AURIMAR SEPKA JÚNIOR

EXTENSÃO DO FRAMEWORK WALBERLA PARA USO DE GPU EM SIMULAÇÕES DO MÉTODO DE LATTICE BOLTZMANN

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2015

RESUMO

Este trabalho apresenta uma proposta de paralelização do Método de Lattice Boltzmann em três dimensões para o *framework* waLBerla utilizando a plataforma de computação paralela *Compute Unified Device Architecture* (CUDA). Com esse propósito, foi desenvolvido um novo módulo para o *framework* waLBerla que permite que as simulações de fluidos já realizados pelo *framework* usando CPUs, sejam realizadas também em GPU.

Esse módulo foi denominado de módulo CUDA e é composto por um conjunto de novas classes e métodos. O módulo foi desenvolvido com base nos conceitos já definidos pelo *framework* waLBerla, dessa maneira a opção por realizar a simulação em CPU ou GPU é acessível para o usuário.

Para validação do módulo foi utilizado o caso de testes conhecido como *lid-driven cavity* em conjunto com o operador de colisão *Single Relaxation Time* (SRT) e com o *stencil* D3Q19. Além do desenvolvimento do módulo CUDA, esse trabalho também levou em conta o desempenho das simulações utilizando memória alocada de maneira linear e alinhada e também analisou diferentes tamanhos de domínio, com a finalidade de definir um critério para alocação eficiente de *grids* de blocos de *threads* para a GPU. Também foram realizadas comparações entre diferentes arquiteturas de GPUs NVIDIA como, Fermi, Kepler e Maxwell.

Os resultados obtidos pelo módulo CUDA estão de acordo com os valores encontrados na literatura. A GPU que obteve o maior desempenho foi a NVIDIA Tesla K40m, alcançando até 612 MLUPS usando precisão dupla para um tamanho de domínio de $256 \times 256 \times 256$ células, atingindo resultados bem expressivos se comparado as demais GPUs.

Palavras-chave: Método de Lattice Boltzmann, D3Q19, waLBerla, CUDA, GPU, Lid-driven cavity, SRT.

ABSTRACT

A parallelization approach of Lattice Boltzmann method in three dimensions for the waLBerla framework using the Compute Unified Device Architecture (CUDA) is present in this work. A new module for the waLBerla framework is developed, allowing the fluid simulations already carried out by the framework using CPUs, are also held in GPU.

This module is called CUDA module and consists of a set of new classes and methods. The CUDA module is based on the concepts already defined for waLBerla framework. This way, the option to perform the simulation in the CPU or in the GPU is straightforward to the user.

To validate the module, we used the lid-driven cavity problem with the SRT collision operator and D3Q19 stencil. Additionally, we compared the simulations' performance using linear memory and pitched memory. Moreover, different sizes of domain were analysed. In this work, we realized comparations between different GPUs architectures like Fermi, Kepler and Maxwell.

The achieved results for the CUDA module are according related works. The GPU had better performance was a NVIDIA Tesla K40m reaches up to 612 MLUPS in double precision for a domian size of $256 \times 256 \times 256$ cells, achieving expressive values if compared of the others GPUs.

Keywords: Lattice Boltzmann Method, D3Q19, waLBerla, CUDA, GPU, Lid-driven cavity, SRT.

LISTA DE FIGURAS

1.1 Etapas para a solução de problemas em DFC. 2
1.2 Comparação entre o desempenho de operações de ponto flutuante entre processadores Intel e as placas gráficas da NVIDIA. 3

2.1 Modelo HPP - As setas representam as posições ocupadas pelas células. . . 7
2.2 Regras de colisão para o modelo HPP. 7
2.3 Modelo FHP. 8
2.4 Regras de colisão para o modelo (a) Determinístico e (b) Não-Determinístico. 8
2.5 Discretização de um fluido em duas dimensões. 11
2.6 Modelo de *lattice* em duas dimensões. 13
2.7 *Lattices* (a) D1Q3 e (b) D1Q5. 14
2.8 *lattice* D2Q9. 15
2.9 *lattice* D3Q19. 16
2.10 Condição de contorno *bounce back*. 17
2.11 Condição de contorno periódica. 17
2.12 Atualização da célula central do *lattice* com os valores das células vizinhas através da propagação das funções de distribuição de partículas. 18
2.13 Sequência dos passos dos algoritmos (a) *stream-collide* e (b) *collide-stream*. 23
2.14 Arranjo do caso de testes *lid-driven cavity*. 24
2.15 Corte vertical sobre o ponto médio do eixo y. Imagem obtida após 10 mil iterações do algoritmo *lid-driven cavity* em GPU para um domínio de 128×128×128 células com número de Reynolds Re = 100 e com velocidade da borda superior $U_w = 0.1$. 24
2.16 *Grid* de blocos de *threads*. 26
2.17 Arquitetura Fermi GF100. 28
2.18 Arquitetura Kepler GF100. 29
2.19 Arquitetura Maxwell GM107. 30
2.20 Hierarquia de memória. 32
2.21 Mapeamento das *threads* de uma região bidimensional da memória. 33
2.22 Memória alinhada. 34
2.23 Um bloco com 9x9 células mais uma camada adicional onde as células tracejadas representam a camada *ghost layer*. 35
2.24 Layout dos dados em memória. 38
2.25 Diagrama de classe com o relacionamento entre as classes *Field*, *GhostLayerField* e *PdfField*. 40

3.1 (a) Iteração par: leitura dos PDFs da célula atual. (b) Iteração par: armazena os valores pós-colisão na célula atual, mas com sentido oposto. (c) Iteração impar: leitura dos PDFs da células vizinhas. (d) Iteração impar: armazena os valores dos PDFs atualizados. 49
3.2 Exemplo do *Swap Algorithm* para o esquema de propagação *streaming-pull* usando o *stencil* D2Q9. (a) Célula atual, (b) Após a troca e (c) Colisão. . . 51

4.1 Principais módulos do *framework* waLBerla e a adição do módulo CUDA. . 56
4.2 Diagrama de classe com o relacionamento entre as classes *GPUField*, *GPUFieldLinearMem* e *GPUFieldMemPitch*. 57
4.3 Diagrama de classe das classes *FieldIndexingLinearMem* e *FieldIndexingMemPitch*. 58
4.4 Diagrama de classe das classes *FieldAccessorLinearMem* e *FieldAccessorMemPitch*. 59
4.5 Diagrama de classe da classe *Kernel*. 60
4.6 Diagrama de classes das classes *GPUNoSlip* e *GPUSimpleUBB*. 61

6.1 Comparação entre os resultados obtidos em o *framework* waLBerla e o módulo CUDA. (a) Representa as velocidades na direção y (V_y) sobre a direção x do domínio e (b) representa as velocidades na direção x (V_x) sobre a direção y do domínio. 74
6.2 Linhas de corrente vórtice principal ao redor do eixo y para o número de Reynolds Re = 100. 74
6.3 Desempenho das simulações usando **memória linear** e com **ECC desabilitado** para um domínio de tamanho N_x×128×128. 75
6.4 Desempenho das simulações usando **memória linear** e com **ECC desabilitado** para um domínio de tamanho N_x×256×256. 76
6.5 Desempenho das simulações usando **memória alinhada** e com **ECC desabilitado** para um domínio de tamanho N_x×128×128. 77
6.6 Desempenho das simulações usando **memória alinhada** e com **ECC desabilitado** para um domínio de tamanho N_x×256×256. 77
6.7 Desempenho das simulações usando **memória linear** e com **ECC habilitado** para um domínio de tamanho N_x×128×128. 78
6.8 Desempenho das simulações usando **memória linear** e com **ECC habilitado** para um domínio de tamanho N_x×256×256. 79
6.9 Desempenho das simulações usando **memória alinhada** e com **ECC habilitado** para um domínio de tamanho N_x×128×128. 79
6.10 Desempenho das simulações usando **memória alinhada** e com **ECC habilitado** para um domínio de tamanho N_x×256×256. 80

6.11 Taxa de Ocupação com relação a quantidade de *threads* para o *kernel* streamAndCollideD3Q19KernelOptimized. 81
6.12 (a) Comparação do desempenho entre **memória linear** e **memória alinhada** para um domínio de tamanho 64×128×128 e com **ECC desabilitado**, (b) Comparação do desempenho entre **memória linear** e **memória alinhada** para um domínio de tamanho 128×128×128 e com **ECC habilitado**. 81
6.13 Comparação com o trabalho de Habich et al. (2011a) usando **memória linear** para um tamanho de domínio de 200×200×200. 82
6.14 Comparação entre o tempo para realizar uma única iteração do LBM e o tempo necessário para copiar os dados da CPU para a GPU Tesla K40m com **ECC habilitado** e para um domínio de tamanho N_x×256×256. . . . 83

LISTA DE TABELAS

2.1 Peso w_i de cada direção para os *lattices* D1Q3 e D1Q5. 14
2.2 Peso w_i de cada direção para os *lattices* D2Q5 e D2Q9. 15
2.3 Peso w_i de cada direção para os *lattices* D3Q15 e D3Q19. 16
2.4 As principais propriedades de cada arquitetura. 31

3.1 Pontos relevantes de cada artigo. 54

5.1 Relação dos testes realizados 72

LISTA DE SIGLAS

AC	Autômato Celular
AoS	Array-of-Structure
API	Application Programming Interface
BGK	Bhatnagar, Gross e Krook
cc	compute capability
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
C3SL	Centro de Computação Científica e Software Livre
DFC	Dinâmica de Fluidos Computacional
ECC	Error Check and Correction
EDP	Equações Diferenciais Parciais
FAU	Friedrich-Alexander-Universität Erlangen-Nürnberg
GPU	Graphics Processing Unit
HPF	Frisch, Hasslacher e Pomeau
HPP	Hardy, de Pazzis e Pomeau
LBM	Lattice Boltzmann Method
LGA	Lattice Gas Automata
LSS	Laboratório de Simulação de Sistemas
MFLUPS	Million Fluid Lattice Updates per Second
MLUPS	Million Lattices Cells Updates per Second
MPI	Message Passing Interface
MRT	Multi Relaxation Time
NS	Navier-Stokes
OpenMP	Open Multi-Processing
PDF	Função de Distribuição de Partículas
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SM	Streaming Multiprocessor
SRT	Single Relaxation Time
SoA	Structure-of-Array
TRT	Two Relaxation Time
VRI	Visão, Robótica e Imagem
waLBerla	Widely Applicable LBM from Erlangen

LISTA DE SÍMBOLOS

a	Aceleração
b	Quantidade de velocidades do *lattice*
d	Dimensão do problema
c_s	Velocidade do som
e	Velocidade das partículas
f	função de distribuição de partículas
f^{eq}	Função de distribuição de equilíbrio
F	Força externa
j_x	Direção x da quantidade de movimento
j_y	Direção y da quantidade de movimento
m	Massa molecular
$\mathbf{m}$	Quantidade de movimento
q_x	Fluxo do calor da direção x
q_y	Fluxo do calor da direção y
S	Diagonal da matriz de relaxação
t	Tempo
u	Velocidade do fluido
u_a	Velocidade da partícula em relação à velocidade do fluido
u_w	Velocidade da borda superior
x	Posição inicial das partículas
Ω	Operador de Colisão
ρ	Densidade
ε_I	Energia interna
τ	Parâmetro de relaxação
ν_i	Viscosidade do fluido
ω_i	Peso associado a cada uma das direções das velocidades

SUMÁRIO

RESUMO **i**

ABSTRACT **ii**

LISTA DE FIGURAS **v**

LISTA DE TABELAS **vi**

LISTA DE SIGLAS **vii**

LISTA DE SÍMBOLOS **viii**

1 INTRODUÇÃO **1**

1.1 Objetivo 4

1.1.1 Objetivos Específicos 4

1.2 Contribuições do Trabalho 4

1.3 Organização do Trabalho 5

2 REFERENCIAL TEÓRICO **6**

2.1 Automato Celular 6

2.2 Lattice Gas Automata 6

2.3 Equação de Boltzmann 9

2.4 Método de Lattice Boltzmann 10

2.4.1 Equação de Lattice Boltzmann 12

2.4.2 Lattices 13

2.4.3 Condições de Contorno 16

2.4.4 Propagação 18

2.4.5 Operadores de Colisão 19

2.4.6 Algoritmo do Método de Lattice Boltzmann 21

2.5 Lid-Driven Cavity 23

2.6 CUDA 25

2.6.1 Programação 25

2.6.2 Arquiteturas 26

2.6.2.1 Arquitetura Fermi GF100 27

2.6.2.2 Arquitetura Kepler GK110 28

2.6.2.3 Arquitetura Maxwell GM107 29

2.6.3 Hierarquia de Memórias 31

2.6.3.1 Memória Constante 32
2.6.3.2 Memória de Textura 32
2.6.3.3 Memória Alinhada 33
2.7 Framework waLBerla 34
2.7.1 Field 38
2.7.2 GhostLayer 39
2.7.3 PdfField 40

3 REVISÃO DA LITERATURA 42
3.1 Uso de memória compartilhada 42
3.2 Análise do impacto do uso de registradores 44
3.3 Otimizações do acesso a memória global da GPU 45
3.4 Otimizações das condições de contorno 47
3.5 Comparação entre diferentes formas de realizar o passo de propagação . . . 48
3.6 Implementação do LBM em clusters 52

4 DESENVOLVIMENTO 55
4.1 Módulo CUDA 55
4.1.1 GPUField 55
4.1.2 Funções de cópia 56
4.1.3 Classes FieldIndexingLinearMem e FieldIndexingMemPitch 58
4.1.4 Classes FieldAccessorLinearMem e FieldAccessorMemPitch 58
4.1.5 Kernel 59
4.1.6 Bordas 60
4.1.7 GPU kernels 61
4.2 Comparação entre a Estrutura do Algoritmo Lid-Driven Cavity em CPU e em GPU 64
4.3 Implementação do Lid-Driven Cavity em GPU 65

5 MATERIAIS E MÉTODOS 67
5.1 Hardware 67
5.2 Teste de Validação 68
5.3 Testes de Desempenho 68
5.4 Teste de Comparação com a Literatura 70
5.5 Teste de Comparação entre o Tempo de Cópia e o Tempo de uma Iteração 71

6 RESULTADOS EXPERIMENTAIS 73
6.1 Teste de Validação 73
6.2 Testes de Desempenho 73
6.3 Teste de Comparação com a Literatura 81

6.4 Teste de Comparação entre o Tempo de Cópia e o Tempo de uma Iteração 82

7 CONCLUSÃO **85**

7.1 Trabalhos Futuros 86

REFERÊNCIAS BIBLIOGRÁFICAS **87**

CAPÍTULO 1

INTRODUÇÃO

Os primeiros estudos realizados sobre a dinâmica dos fluidos foram feitos de maneira experimental através da observação dos fluidos na natureza, como a água e o ar. Isso porque, ainda não havia nenhuma formulação teórica que explicasse o comportamento e as propriedades dos fluidos. Com o passar do tempo, os primeiros tratados matemáticos surgiram para descrever o movimento dos fluidos por meio do trabalho do matemático Leonard Euler, as chamadas equações de Euler. Mas foi através das equações de Navier-Stokes (NS) que a dinâmica dos fluidos ganhou força.

As equações de NS podem descrever com precisão os mais variados fenômenos físicos que envolvem o movimento dos fluidos, como escoamentos laminares, compressíveis, incompressíveis e isotérmicos. Os principais aspectos das equações de NS são: a conservação da massa e conservação da quantidade de movimento. Tanto as equações de NS, quanto as equações de Euler, tratam as propriedades físicas e as forças que atuam sobre o fluido de maneira macroscópica.

As equações de NS são Equações Diferenciais Parciais (EDP) não lineares e devido à sua complexidade, não há uma maneira de determinar soluções de forma analítica para problemas reais. Uma maneira para obter soluções para as equações de NS é através de soluções numérias utilizando simulações computacionais.

A Dinâmica de Fluidos Computacionais (DFC) é uma área da computação científica que estuda o comportamento de fluidos em movimento. Entre os fenômenos estudados pela DFC podemos citar: estudos do movimento do ar atmosférico, escoamento de fluidos ao redor de um corpo (aerodinâmica) e o estudo do movimento dos fluidos do corpo humano.

A DFC tem como objetivo auxiliar os modelos teóricos e experimentais através da validação de seus métodos, redução do número de experimentos, auxiliar a compreensão da natureza dos fenômenos que envolvem o movimento dos fluidos e permitir com que fenômenos, antes não explorados, possam ser simulados em laboratórios de maneria prática. Além disso, a DFC permite com que os parâmetros das simulações possam ser alterados até atingirem os resultados esperados, de maneira mais conveniente e a um custo e tempo menor (Fortuna, 2012).

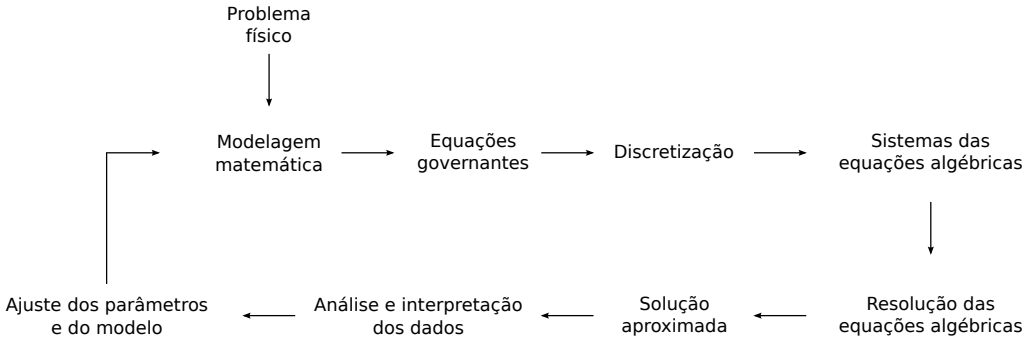
Para representar o comportamento dos fluidos, através da DFC, é necessário expressar as equações que regem o movimento dos fluidos e o domínio da solução[1] de forma adequada. No entanto, como não é possível obter soluções numéricas sobre todo o domínio,

[1]Região contínua que representa o fluido.

é necessário discretizar essa região contínua, ou seja, dividir em pontos. Dessa maneira, a solução é apenas obtida sobre os pontos. O conjunto de pontos discretizados recebe o nome de malha ou *grid* e a escolha de como os pontos serão distribuídos sobre essa malha, tem papel fundamental para convergência da solução numérica da simulação, pois quanto mais densa for a malha escolhida, mais preciso será o resultado do modelo, entretanto, o custo computacional será maior (Fortuna, 2012).

Além da discretização do domínio e da definição das equações, que descrevem o comportamento do fluidos, à ainda outras etapas necessárias para a solução de problemas em DFC, como ilustra a Figura 1.1.

Figura 1.1: Etapas para a solução de problemas em DFC.



Fonte: Adaptado de (Fortuna, 2012).

O Método de Lattice Boltzmann (LBM) surgiu como uma alternativa às equações de NS para soluções numérias da DFC. O LBM é um método mesoscópico e adimensional que simula o comportamento de um fluido através da representação das partículas do fluido sobre uma malha (*lattice*), onde cada ponto da malha representa uma célula. Cada célula contém um número fixo de Funções de Distribuição de Partículas (PDF). Essas funções representam uma porção de fluido que se propaga em direções específicas em relação as células vizinhas e com uma velocidade definida. Durante a propagação podem ocorrer colisões entre as partículas do fluido ou colisões com as bordas do recipiente no qual o fluido está mantido, fazendo com que o sentido das partículas seja alterado.

O LBM é um método iterativo, no qual, a cada iteração são atualizados os valores das PDFs de cada célula do *lattice* com os valores da interação entre as células vizinhas. Podemos citar alguns exemplos da utilização do LBM como: simulação de dispersão de poluentes no ar, simulações do fluxo sanguíneo nas principais artérias do corpo humano (Golbert et al., 2009) e simulações aerodinâmicas (Wang et al., 2008).

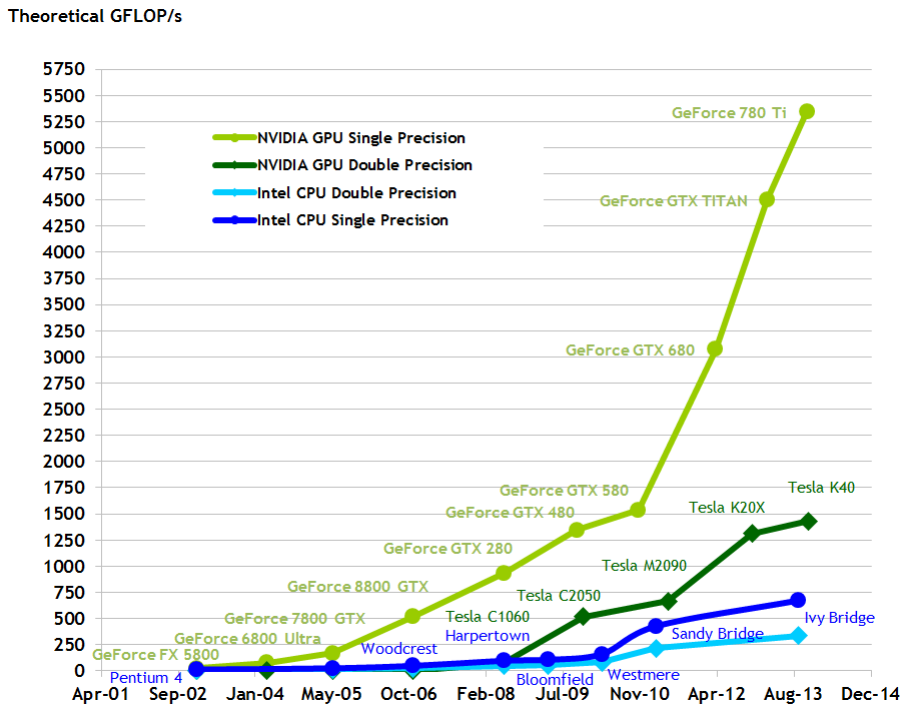
Para que o LBM apresente uma solução numérica (simulação) mais realista, é necessário executar milhares de iterações, ou seja, para cada iteração um novo cálculo será realizado sobre cada ponto do domínio. Portanto, para gerar simulações mais reais o tamanho da malha deve ser menor, com isso, a quantidade de dados processados será maior. Considerando a quantidade de dados e o número de iterações, o tempo necessário para

executar uma simulação se torna um fator crítico. Uma técnica bastante comum para reduzir o tempo total de cada simulação é a utilização de computação paralela, onde todo o domínio pode ser dividido e distribuído entre diferentes processadores, com isso, domínios menores podem ser processados em paralelo reduzindo o tempo total da simulação. Além da redução de tempo, a computação paralela permite com que problemas maiores e mais complexos sejam simulados tornando as soluções de diversos problemas mais confiáveis (Fortuna, 2012).

Na DFC as simulações de fluidos são realizadas através de algoritmos ou através de *frameworks* para simulação de fluidos. O *framework* waLBerla, que será utilizado nesse trabalho, foi desenvolvido para a simulação de fluidos complexos que utilizam o LBM. Desenvolvido em C++, o *framework* possui suporte a computação paralela em *Open Multi-Processing* (OpenMP) e *Message Passing Interface* (MPI), porém, a atual versão, não possui suporte as Unidades de Processamento Gráfico (GPU).

As atuais GPUs possuem uma grande quantidade de processadores paralelos, com isso, sua capacidade de processamento, assim como a quantidade de operações de ponto flutuante de precisão simples e dupla realizadas, são extremamente altas se comparado com uma CPU, como pode ser visto na Figura 1.2, sendo assim adequadas para simulações em DFC.

Figura 1.2: Comparação entre o desempenho de operações de ponto flutuante entre processadores Intel e as placas gráficas da NVIDIA.



Fonte: NVIDIA (2015a)

Uma tecnologia que vem sendo bastante utilizada para esse propósito é a arquitetura *Compute Unified Device Architecture* (CUDA) desenvolvida pela NVIDIA. CUDA é uma

plataforma de computação paralela de propósito geral e um modelo de programação que aproveita a arquitetura das GPUs para resolver problemas computacionais complexos. Além de sua capacidade de processamento, CUDA também mantém uma baixa curva de aprendizado para os programadores familiarizados com as linguagens de programação, como C e C++ (NVIDIA, 2015a).

Com base na tecnologia CUDA, na capacidade de processamento da atuais GPUs e pela falta de suporte à GPU do *framework* waLBerla, optou-se por desenvolver um novo módulo que permita ao usuário do *framework* realizar, de maneira acessível, as simulações em GPU.

1.1 Objetivo

- Extender o *framework* waLBerla para suportar o uso de GPU (CUDA) em simulação do método de Lattice Boltzmann.

1.1.1 Objetivos Específicos

- Definir uma nova estrutura de dados e métodos para execução do método de Lattice Boltzmann em GPU;
- Desenvolver um mecanismo (método) que torne acessível ao usuário do *framework* waLBerla o uso de GPU para simulações desenvolvidas para a CPU;
- Avaliar estratégias de alocação de memória em GPU;
- Avaliar a relação entre o tamanho do domínio da simulação e a quantidade de *threads* em cada bloco em relação ao desempenho computacional.

1.2 Contribuições do Trabalho

As principais contribuições desse trabalho são listas abaixo:

- Desenvolvimento de um módulo para o *framework* waLBerla que permite realizar as simulações do LBM em GPU;
- Possibilita ao usuário do *framework* waLBerla o uso de GPU em simulações LBM de forma acessível;
- O desenvolvimento do módulo foi realizado considerando o uso de múltiplas GPUs;
- Uma análise do desempenho entre alocação de memória de forma linear e alinhada para simulações do LBM em três dimensões;

- Uma análise do desempenho entre diferentes arquiteturas de GPUs NVIDIA em simulações do LBM em três dimensões.

1.3 Organização do Trabalho

No Capítulo 2 será apresentado o referencial teórico utilizado nesse trabalho. Inicialmente, o capítulo apresenta o LBM, descrevendo a origem do método a partir do *Lattice Gas Automata* e/ou a partir da equação de Boltzmann, apresenta ainda as equações do método, os modelos de *lattices*, as condições de contorno, os operadores de colisão e por fim, apresenta um esquema do algoritmo de LBM. Em seguida, introduz a plataforma de computação paralela CUDA e as principais arquitetura de GPUs, e, finalmente, apresenta os principais conceitos sobre o *framework* waLBerla. O Capítulo 3, apresenta os trabalhos relacionados. Já o Capítulo 4, mostra todas as etapas do desenvolvimento do módulo CUDA e uma comparação entre a estrutura dos algoritmos no *framework* waLBerla e o módulo CUDA. Os equipamentos utilizados, bem como os testes realizados, são apresentados no Capítulo 5. Os resultados obtidos são encontrados no Capítulo 6, e a conclusão do trabalho assim como possíveis trabalhos futuros, são apresentados no Capítulo 7.

CAPÍTULO 2

REFERENCIAL TEÓRICO

Neste capítulo são apresentados os principais conceitos para a compreensão do LBM, desde sua evolução até uma representação do algoritmo utilizado para o desenvolvimento desse trabalho. Além disso, são apresentados os conceitos básicos para o desenvolvimento de aplicações paralelas utilizando a plataforma CUDA e as atuais arquiteturas de GPUs NVIDIA. Como o *framework* waLBerla é parte fundamental desse trabalho seus principais conceitos também serão introduzidos.

Nas seções 2.1, 2.2 e 2.3 são introduzidos os modelos que deram origem ao LBM. A seção 2.4 aborda os principais conceitos relacionados ao LBM. Na seção 2.5 o modelo *lid-driven cavity* utilizado como caso de testes é apresentado. A plataforma de computação paralela CUDA é explicada na seção 2.6 e, finalmente, na seção 2.7, o *framework* waLBerla é introduzido.

2.1 Automato Celular

Um dos primeiros conceitos, necessário para compreender a origem do LBM, é o Autômato Celular (AC). O AC é um modelo utilizado para representar o comportamento de fenômenos naturais e foi introduzido pelos matemáticos John von Neumann e Stanislaw Ulam, na década de 40. Consiste em uma malha (*lattice*), onde cada posição da malha possui um número finito de estados, geralmente variáveis Booleanas.

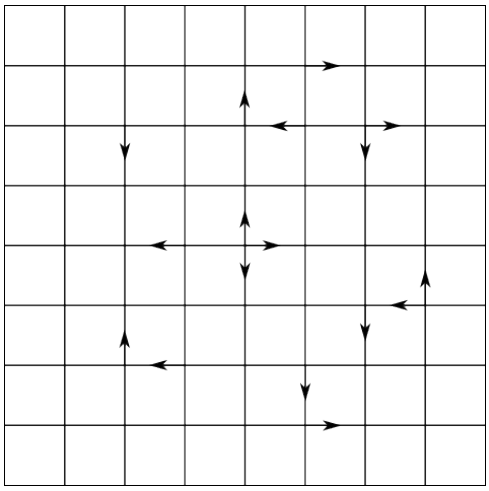
Os autômatos evoluem em passos discretos sendo simultaneamente atualizados por uma regra determinística ou não determinística. Tipicamente, apenas um número finito de vizinhos são envolvidos na atualização de uma determinada posição, ou seja, um vizinho influencia no comportamento do outro. O exemplo mais popular de autômato celular é o *Game of Life* (jogo da vida), criado pelo também matemático Jonh Horton Conway (Frisch et al., 1987).

2.2 Lattice Gas Automata

Lattice Gas Automata (LGA) é uma classe de autômato celular usado para simulações em dinâmica de fluidos. O primeiro modelo de *lattice gas* foi introduzido por Hardy, de Pazzis e Pomeau, e ficou conhecido como modelo HPP. Esse modelo é totalmente determinístico tendo como finalidade analisar questões fundamentais na mecânica estatística, como a divergência dos coeficientes de transporte em duas dimensões.

O modelo HPP consiste de um *lattice* quadrado em duas dimensões, como mostra a Figura 2.1, onde cada nó está conectado a quatro vizinhos. As partículas, com massa e velocidade unitária, se propagam ao longo das ligações do *lattice* e obedecem ao princípio da exclusão mútua, ou seja, apenas uma única partícula pode ocupar uma posição do *lattice* no mesmo instante de tempo.

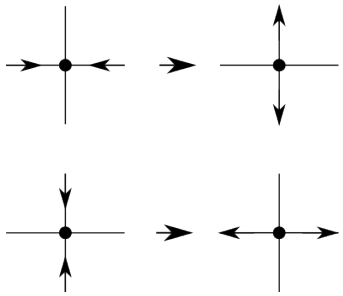
Figura 2.1: Modelo HPP - As setas representam as posições ocupadas pelas células.



Fonte: Autor.

A colisão entre partículas ocorre quando duas partículas, que se propagam na mesma direções e sentidos opostos, se encontram na mesma posição do *lattice*. Logo após a colisão as partículas ocupam as posições que estavam livres no instante de tempo anterior a colisão, conforme ilustra a Figura 2.2. Mesmo após a colisão há conservação de massa e quantidade de movimento (Frisch et al., 1987; Wolf-Gladrow, 2000).

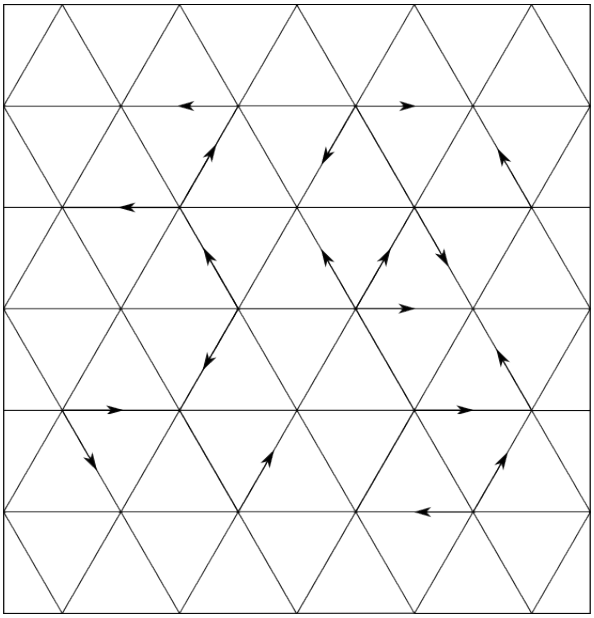
Figura 2.2: Regras de colisão para o modelo HPP.



Fonte: Autor.

Outro modelo de LGA foi introduzido por Frisch, Hasslacher e Pomeau (Frisch et al., 1987) em 1987, e ficou conhecido como modelo FHP. Esse modelo é baseado no modelo HPP e possui um *lattice* na forma hexagonal, onde cada nó está conectado a seis vizinhos, e não mais a quatro como no modelo HPP, assim cada partícula pode se deslocar por seis direções diferentes, como mostra a Figura 2.3.

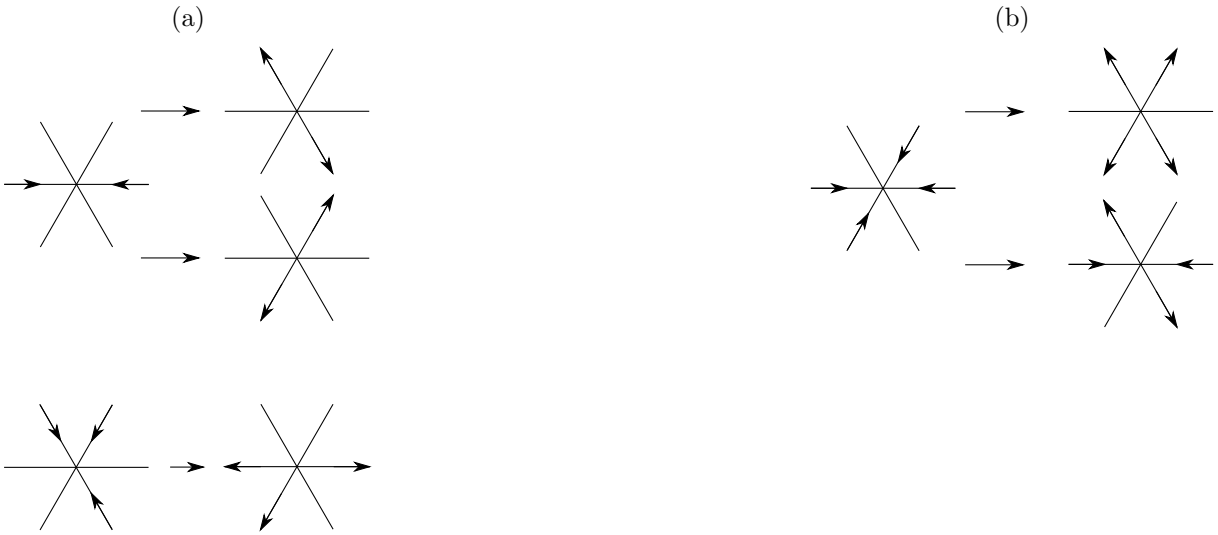
Figura 2.3: Modelo FHP.



Fonte: Autor.

A forma de propagação das partículas é a mesma do modelo HPP, porém as regras de colisão são diferentes como mostra a Figura 2.4. A Figura 2.4 (a) representa o modelo determinístico, onde as partículas que sofreram colisão são rotacionadas em $\pi/3$. Já a Figura 2.4 (b) representa o modelo não-determinístico, onde a rotação das partículas é definida de forma aleatória (Frisch et al., 1987; Wolf-Gladrow, 2000).

Figura 2.4: Regras de colisão para o modelo (a) Determinístico e (b) Não-Determinístico.



Fonte: Autor.

2.3 Equação de Boltzmann

Ludwig Eduard Boltzmann (1844-1906) foi um físico austríaco que teve papel fundamental na teoria cinética dos gases, principalmente, pela sua equação de distribuição das velocidades das partículas (Kremer and Medeiros, 2005). A função de distribuição $f(x, e, t)$, conforme apresentado em Mohamad (2011), fornece o número de partículas que no instante de tempo t encontram-se posicionadas entre x e $x + dx$ com velocidades entre e e $e + de$, onde x e e representam, respectivamente, os vetores de posição espacial e de velocidade das partículas em um instante de tempo t.

Quando uma força externa atua sobre as partículas, gerando a aceleração, a velocidade e a posição inicial serão alteradas de e para $e + Fdt$ e de x para $x + edt$, respectivamente. O número de partículas após a perturbação deve ser igual ao estado inicial. Assim, se nenhuma colisão ocorrer entre as partículas devido a força externa temos:

$$f(x + edt, e + Fdt, t + dt)dxde - f(x, e, t)dxde = 0 \tag{2.1}$$

onde $f(x, e, t)$ é o estado inicial e $f(x + edt, e + Fdt, t + dt)$ representa o estado seguinte à atuação da força.

Entretanto, se houveram colisões entre as partículas será necessário representá-las na Equação 2.1, para isso, o operador de colisão Ω é inserido na equação. O operador Ω representa a taxa de mudança entre o estado inicial e final da equação de distribuição. A equação 2.2 mostra a função de distribuição com o operador de colisão:

$$f(x + edt, e + Fdt, t + dt)dxde - f(x, e, t)dxde = \Omega(f)dxdedt \tag{2.2}$$

Dividindo a equação acima por $dxdedt$ e como o limite de $dt \to 0$, resulta em

$$\frac{df}{dt} = \Omega(f) \tag{2.3}$$

A Equação 2.3 indica que a taxa de mudança da função de distribuição é igual à taxa de colisão. Como f é uma função de x, e e t, a taxa total de mudança pode ser expressa como,

$$df = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial e}de + \frac{\partial f}{\partial t}dt \tag{2.4}$$

e dividindo a equação 2.4 por dt, temos:

$$\frac{df}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial e}\frac{de}{dt} + \frac{\partial f}{\partial t} \tag{2.5}$$

A equação 2.5 pode ser escrita da seguinte maneira,

$$\frac{df}{dt} = \frac{\partial f}{\partial x}e + \frac{\partial f}{\partial e}a + \frac{\partial f}{\partial t} \tag{2.6}$$

onde a representa a aceleração e possui valor igual a de/dt.

De acordo com a Segunda lei de Newton ($F = m \cdot a$), a pode ser escrito como $a = F/m$. Portanto a equação de transporte de Boltzmann (Equação 2.3), pode ser definida como:

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial x} \cdot e + \frac{F}{m} \cdot \frac{\partial f}{\partial e} = \Omega \tag{2.7}$$

Para um sistema onde não há nenhuma força externa atuando a equação de Boltzmann pode ser representada da seguinte maneira:

$$\frac{\partial f}{\partial t} + e \cdot \triangle f = \Omega \tag{2.8}$$

A relação entre a equação acima e as quantidades macroscópicas tal como densidade (ρ), velocidade do fluido (u) e energia interna (ε_I) são as seguintes:

$$\rho(x,t) = \int m f(x,e,t) de \tag{2.9}$$

$$\rho(x,t)u(x,t) = \int mc f(x,e,t) de \tag{2.10}$$

$$\rho(x,t)\varepsilon_I(x,t) = \frac{1}{2}\int m u_a^2 f(x,e,t) de \tag{2.11}$$

onde m é a massa molecular e u_a é a velocidade da partícula em relação à velocidade do fluido, $u_a = e - u$;

As equações 2.9, 2.10 e 2.11 representam a conservação da massa, quantidade de movimento e energia, respectivamente (Mohamad, 2011).

2.4 Método de Lattice Boltzmann

O Método de Lattice Boltzmann (LBM) surgiu como uma eficiente alternativa às equações de Navier-Stokes (NS) para soluções numérias na Dinâmica de Fluidos Computacionais (DFC), pois no LBM a discretização da equação de Boltzmann resulta em um conjunto de Equações Diferenciais Parciais (EDP), enquanto que as equações de NS são EDP de segunda ordem (Raabe, 2004; Yu et al., 2003).

O LBM simula o comportamento de um fluido através da representação de suas partículas sobre uma malha (*lattice*). Cada ponto da malha representa uma célula contendo um número fixo de Funções de Distribuição de Partículas (PDF). Essas funções representam uma porção de fluido que se propaga em direções específicas, em relação as células vizinhas, e com velocidade definida. Durante a propagação, o sentido das partículas pode ser alterado quando ocorrerem colisões entre as partículas do fluido ou colisões com as bordas do recipiente no qual o fluido está mantido.

A colisão entre as partículas é definida de acordo com o operador de colisão, como será visto na Seção 2.4.5, enquanto que o passo de propagação é realizado através da interação entre células vizinhas do *lattice*, assim como no autômato celular. Segundo Mohamad (2011), o LBM pode ser considerado como um método explícito, pois os passos de colisão e propagação são locais, tornando-o um método altamente paralelizável.

Além disso, o LBM é considerado um modelo mesoscópico, pois considera o comportamento de um conjunto de partículas, ao invés de considerar o comportamento individual de cada partícula, como no modelo molecular, e também leva em consideração valores macroscópicos como densidade e velocidade.

Historicamente, o LBM foi proposto por McNamara e Zanetti (McNamara and Zanetti, 1988) em 1988, como uma evolução do *lattice gas automata*. Embora, o LBM também possa ser derivado diretamente de uma simplificação da equação de Boltzmann (He and Luo, 1997).

Do ponto de vista computacional, o fluido no LBM pode ser discretizado de acordo com a Figura 2.5, onde b representa as partículas em contato com as bordas do recipiente, sobre as quais são aplicadas as condições de contorno, e f representa as demais partículas e sobre elas são aplicados o operador de colisão e o passo de propagação.

Figura 2.5: Discretização de um fluido em duas dimensões.

b	b	b	b	b	b	b	b
b	f	f	f	f	f	f	b
b	f	f	f	f	f	f	b
b	f	f	f	f	f	f	b
b	f	f	f	f	f	f	b
b	f	f	f	f	f	f	b
b	f	f	f	f	f	f	b
b	b	b	b	b	b	b	b

Fonte: Autor.

O LBM possui muitas vantagens, como: a aplicação em domínios complexos (Fietz et al., 2012; Godenschwager et al., 2013), tratamento de fluxos multifase e multicomponente, sem a necessidade de tratar as interfaces entre diferentes fases, e pode ser facilmente paralelizável.

Outra vantagem do LBM é que não há a necessidade de resolver a equação de Laplace

em cada iteração para satisfazer à equação de continuidade para fluxos incompressível e fluxos não estáveis, como é feito nas equações de NS. E também, pode ser utilizado para tratar problemas em escala microscópica e macroscópica com precisão confiável (Mohamad, 2011). No entanto, segundo Mohamad (2011), o LBM requer mais recursos computacionais, como memória, em comparação com simulações que utilizam as equações de NS.

2.4.1 Equação de Lattice Boltzmann

A equação de Lattice Boltzmann é definida através da função de distribuição de partículas $f(x,t)$, que representa a probabilidade de encontrar no instante de tempo t um conjunto de partículas na posição x e com velocidade e_i, como mostra a equação abaixo:

$$f_i(x + e_i\Delta t, t + \Delta t) - f_i(x,t) = \Omega_i \tag{2.12}$$

O lado esquerdo da Equação 2.12 corresponde ao termo de propagação das partículas, enquanto que o lado direito (Ω_i) representa o operador de colisão, introduzido em 1954 por Bhatnagar, Gross e Krook (BGK). Esse operador também é conhecido por *Single Relaxation Time* (SRT) e é definido como:

$$\Omega_i = -\frac{1}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) \tag{2.13}$$

Onde τ é conhecido como parâmetro de relaxação, o qual define o tempo médio entre a colisão das partículas e está relacionado com a viscosidade ν_i do fluido de acordo com:

$$\nu_i = \frac{(2\tau - 1)}{6} \tag{2.14}$$

A função de distribuição de equilíbrio (f^{eq}), utilizada no operador de colisão (Equação 2.13), descreve a função de distribuição de partículas local com velocidade u e densidade ρ quando o fluido alcança um estado de equilíbrio (Qian et al., 1992). Essa função é derivada da função de distribuição de equilíbrio de Maxwell-Boltzmann. A distribuição de equilíbrio é uma função que depende da velocidade e da densidade, dessa maneira, as grandezas macroscópicas devem ser determinadas antes do cálculo da função de distribuição de equilíbrio. Portanto, para calcular a densidade e a velocidade, temos:

$$\rho(x,t) = \sum_i f_i(x,t) \tag{2.15}$$

$$u(x,t) = \frac{\sum_i f_i(x,t)e_i}{\rho(x,t)} \tag{2.16}$$

Assim, é possível calcular a função de distribuição de equilíbrio:

$$f_i^{eq}(\rho, u) = \omega_i \rho \left[1 + \frac{3}{c_s^2} e_i \cdot u + \frac{9}{2c_s^4}(e_i \cdot u)^2 - \frac{3}{2c_s} u \cdot u\right] \tag{2.17}$$

onde c_s representa a velocidade do som e ω_i é um peso associado a cada uma das direções das velocidades de acordo com o modelo de *lattice*, como será visto na Seção 2.4.2 (Succi, 2001).

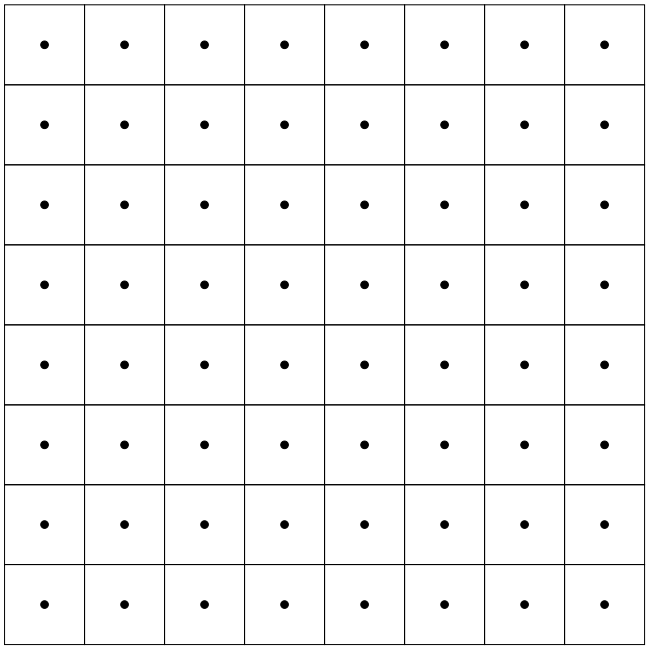
E, por fim, temos a equação de Lattice Boltzmann discretizada com o operador de colisão SRT:

$$f_i(x + e_i \Delta t, t + \Delta t) - f_i(x, t) = -\frac{1}{\tau}(f_i(x, t) - f_i^{eq}(x, t)) \tag{2.18}$$

2.4.2 Lattices

No LBM o domínio da solução é dividido em *lattices*, conforme a Figura 2.6, sendo que cada ponto está associado a um conjunto de valores que representam as funções de distribuição de partículas. Cada *lattice* possui uma dimensão e o número de velocidades microscópicas (e_i), e é representado por *DdQb*, onde d é a dimensão do problema e b é a quantidade de velocidades. A seguir serão apresentados os modelos de *lattices* mais utilizados para uma, duas e três dimensões:

Figura 2.6: Modelo de *lattice* em duas dimensões.



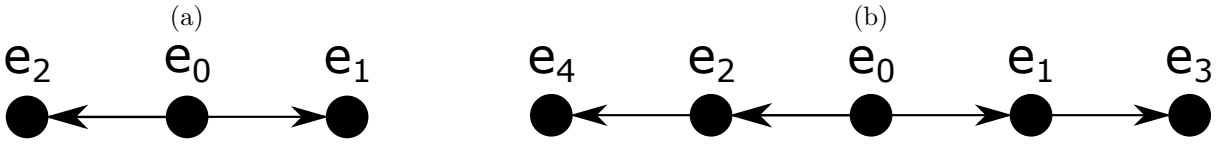
Fonte: Autor.

Os modelos mais utilizados em uma dimensão são conhecidos como: D1Q3 e D1Q5, como mostra a Figura 2.7. O modelo D1Q3 possui três vetores de velocidades (e_0, e_1e e_2) com valores 0, 1 e -1 respectivamente. Assumindo que $\Delta x = \Delta t$, temos $e_1 = \Delta x/\Delta t$ e $e_2 = -\Delta x/\Delta t$, onde Δx representa o comprimento de cada célula do *lattice* e Δt é tempo

necessário para percorrer cada célula. Sendo que a velocidade da função de distribuição de partículas central (f_0) é zero.

Além da velocidade, há também um peso associado a cada direção (w_i), ou seja, quanto cada direção interfere na densidade do ponto, esses valores podem ser encontrados na Tabela 2.1. Já o modelo D1Q5 possui um arranjo com cinco velocidades (e_0, e_1, e_2, e_3e e_4) e os pesos associados a cada direção estão na Tabela 2.1. Note que, para ambos os modelos de *lattices*, as funções de distribuição de partículas são apenas propagadas para a esquerda e para a direita no passo de propagação, ou seja, apenas na direção de cada seta.

Figura 2.7: *Lattices* (a) D1Q3 e (b) D1Q5.



Fonte: Autor.

Tabela 2.1: Peso w_i de cada direção para os *lattices* D1Q3 e D1Q5.

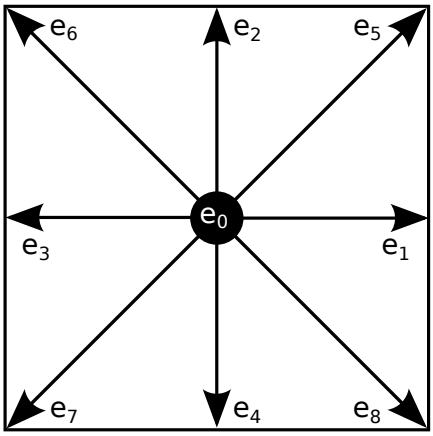
lattice	PDF	w_i
D1Q3	f_0	4/6
	f_1 e f_2	1/6
D1Q5	f_0	6/12
	f_1 e f_2	2/12
	f_3 e f_4	1/12

Para duas dimensões temos os modelos D2Q5 e D2Q9. O modelo D2Q5 possui cinco vetores de velocidade, sendo que o valor da velocidade para a função f_0 é zero ($e_0(0,0)$). As funções de distribuição f_1 e f_3 propagam-se com velocidade de $e_1(1,0)$ e $e_3(-1,0)$ nas direções leste e oeste, respectivamente, enquanto que as funções f_2 e f_4 propagam-se com velocidade $e_2(0,1)$ e $e_4(0,-1)$ nas direções norte e sul, respectivamente.

O modelo D2Q9 (Figura 2.8) é o mais utilizado para simulações do LBM em duas dimensões. Esse modelo possui nove vetores de velocidade, $e_0(0,0)$, $e_1(1,0)$, $e_2(0,1)$, $e_3(-1,0)$, $e_4(0,-1)$, $e_5(1,1)$, $e_6(-1,1)$, $e_7(-1,-1)$ e $e_8(1,-1)$ para as funções f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7 e f_8, respectivamente.

Os pesos associados a cada direção das funções de distribuição de partículas dos modelos D2Q5 e D2Q9, estão na Tabela 2.2.

Figura 2.8: *lattice* D2Q9.



Fonte: Autor.

Tabela 2.2: Peso w_i de cada direção para os *lattices* D2Q5 e D2Q9.

lattice	PDF	w_i
D2Q5	f_0	$2/6$
	f_1, f_2, f_3 e f_4	$1/6$
D2Q9	f_0	$4/9$
	f_1, f_2, f_3 e f_4	$1/9$
	f_5, f_6, f_7 e f_8	$1/36$

Os modelos de *lattices* D3Q15 e D3Q19 são os modelos mais usados em três dimensões. O modelo D3Q15 possui 15 vetores de velocidades para as funções de distribuição (f_i) com velocidades:

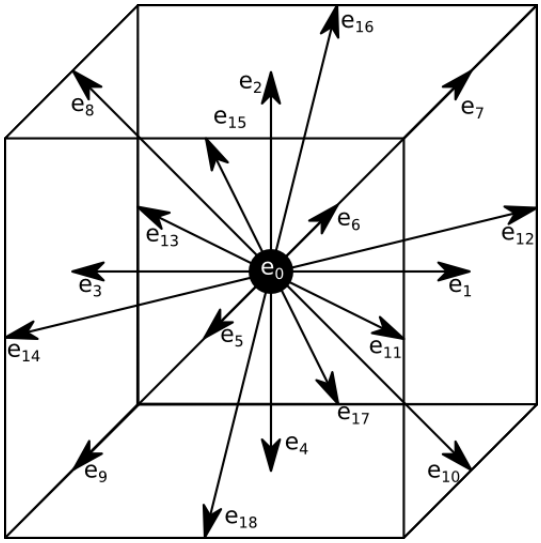
$$e_i = \begin{cases} (0,0,0) & i = 0 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1) & i = 1,2,...,5,6 \\ (\pm 1,\pm 1,\pm 1) & i = 7,8,...,13,14. \end{cases}$$

Já o modelo D3Q19 possui 19 vetores de velocidades, como mostra a Figura 2.9, para as funções de distribuição (f_i) com velocidades:

$$e_i = \begin{cases} (0,0,0) & i = 0 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1) & i = 1,2,...,5,6 \\ (\pm 1,\pm 1,0),(\pm 1,0,\pm 1),(0,\pm 1,\pm 1) & i = 7,8,...,17,18. \end{cases}$$

Os pesos para cada direção dos modelos D3Q15 e D3Q19, estão na Tabela 2.3 (Mohamad, 2011).

Figura 2.9: *lattice* D3Q19.



Fonte: Autor.

Tabela 2.3: Peso w_i de cada direção para os *lattices* D3Q15 e D3Q19.

lattice	PDF	w_i
D3Q15	f_0	16/72
	f_1 a f_6	8/72
	f_7 a f_{14}	1/72
D3Q19	f_0	12/36
	f_1 a f_6	2/36
	f_7 a f_{18}	1/36

2.4.3 Condições de Contorno

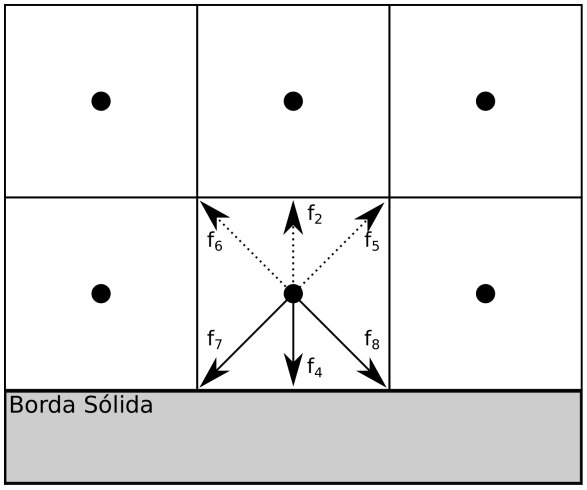
Um quesito importante nas simulações de LBM é a maneira como as condições de contorno são definidas. A escolha correta dessas condições determina melhores soluções e resultados precisos. Para isso, é necessário determinar equações apropriadas para calcular as funções de distribuição de partículas em cada borda do domínio, ou seja, como as partículas irão se comportar ao atingirem as bordas do recipiente no qual o fluido se encontra ou quando colidirem com um objeto dentro do fluido.

A condição de contorno conhecida como *Bounce Back* é amplamente utilizada em simulações de LBM e é aplicada sobre bordas sólidas ou objetos estáticos. Nessa condição de contorno as partículas movendo-se em direção à borda (setas contínuas) colidem e retornam na mesma direção mas em sentido contrário (setas pontilhadas), conforme a Figura 2.10.

Para atualizar as funções de distribuição de partículas as PDFs são trocadas da seguinte forma: $f_2 = f_4$, $f_5 = f_7$ e $f_6 = f_8$, onde f_2, f_5 e f_6 são chamadas de funções de distribuição de partículas desconhecidas. A condição de contorno *bounce back* também é conhecida como *no slip*, ou seja, não escorregadia, pois são bordas que não alteram

o direção as partículas após a colisão. Segundo Mohamad, essa condição de contorno garante a conservação de massa e a quantidade de movimento nas bordas (Mohamad, 2011).

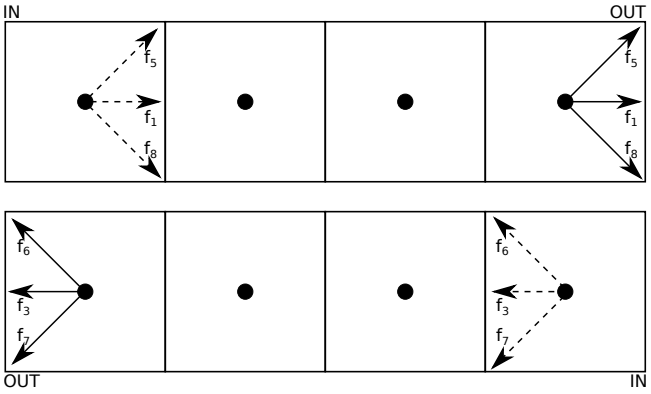
Figura 2.10: Condição de contorno *bounce back*.



Fonte: Autor.

Outro tipo de condição de contorno bastante utilizado é a condição de contorno periódica. Essa condição é implementada e utilizada em condições de fluxo repetitivo. Para ilustrar essa condição a Figura 2.11 é apresentada, onde as funções de distribuição de partículas desconhecidas são atualizadas da seguinte forma: $f_1^{IN} = f_1^{OUT}$, $f_5^{IN} = f_5^{OUT}$ e $f_8^{IN} = f_8^{OUT}$ para um fluxo da esquerda para direita e $f_3^{IN} = f_3^{OUT}$, $f_6^{IN} = f_6^{OUT}$ e $f_7^{IN} = f_7^{OUT}$, para um fluxo da direita para a esquerda (Mohamad, 2011).

Figura 2.11: Condição de contorno periódica.



Fonte: Autor.

Há um terceiro tipo de condição de contorno chamado paredes móveis, semelhante à condição *bounce back*. Porém, como é uma borda móvel, isto é, possui velocidade, é necessário incluir um novo termo ao cálculo das funções de distribuição de partículas das células vizinhas à borda, pois a velocidade da borda móvel será transferida para as PDFs das células em contato com a parede. A velocidade dessa borda se mantém

constante durante toda a simulação. Essa condição de contorno pode ser definida da seguinte maneira (Yu et al., 2003):

$$f_{i'}(x, t + \Delta t) = f_i(x, t + \Delta t) + 6 w_i \rho_i e_i \cdot u_w \tag{2.19}$$

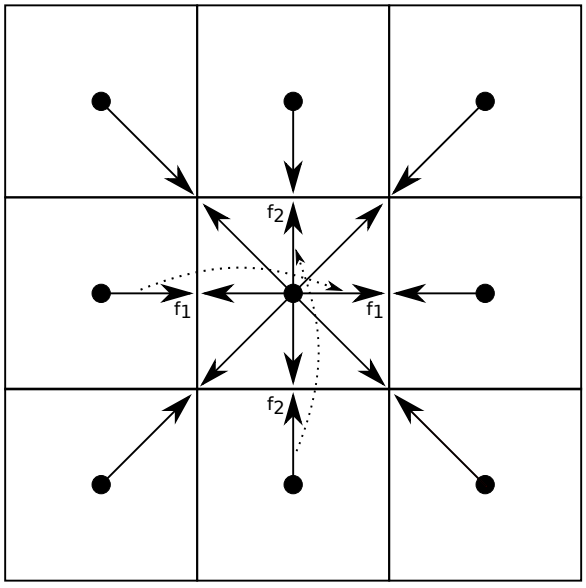
onde, o índice i representa o sentido de uma célula de fluido em relação à uma célula da borda, i' indica o sentido oposto, $f_{i'}$ representa a função de distribuição de partículas de uma célula da borda, f_i representa a função de distribuição de partículas de uma célula de fluido, u_w é a velocidade da borda, ρ_i é a densidade do fluido e e_i representa as velocidades microscópicas de cada PDF.

2.4.4 Propagação

No LBM os valores das funções de distribuição de partículas são propagados para as células vizinhas, semelhante a um autômato celular, conforme a Figura 2.12. Essa propagação também é conhecida como *streaming-push* e ocorre a seguinte maneira: a cada iteração os valores das PDFs da célula atual (célula central da Figura 2.12) são atualizados com os valores das PDFs das células vizinhas.

Cada PDF é atualizada apenas com o valor da mesma direção da célula vizinha, como mostra a Figura 2.12, onde o valor da PDF f_1 da célula central é atualizado com o valor de f_1 da célula à esquerda e o valor da PDF f_2 é atualizado com o valor de f_2 da célula inferior. As demais PDFs são atualizados da mesma maneira até que todas as PDFs da célula atual sejam atualizados.

Figura 2.12: Atualização da célula central do *lattice* com os valores das células vizinhas através da propagação das funções de distribuição de partículas.



Fonte: Autor.

2.4.5 Operadores de Colisão

Os operadores de colisão são responsáveis por determinar o tempo médio entre a colisão das partículas. Além do operador SRT (Equação 2.13), há outro operador que pode ser utilizado para simulações de LBM, denominado *Multi-Relaxation Time* (MRT).

O MRT, proposto por d'Humières em 1992, permite um grau de liberdade maior das partículas, pois para cada função de distribuição há um valor de relaxação (τ) diferente associado, dessa maneira é mais estável e preciso que o operador SRT, entretanto, segundo Mohamad (2011), seu desempenho é inferior (d'Humières, 2002; Mohamad, 2011).

O operador de colisão MRT pode ser generalizado como:

$$f_i(x + e_i\Delta t, t + \Delta t) - f_i(x, t) = -\Omega[f_i(x, t) - f_i^{eq}(x, t)] \tag{2.20}$$

onde Ω é a matriz de colisão.

Para resolver essa equação é necessário transformar a velocidade em quantidade de movimento, assim a Equação 2.20 pode ser transformada da seguinte forma:

$$f_i(x + e_i\Delta t, t + \Delta t) - f_i(x, t) = -M^{-1}S(\mathbf{m}(x, t) - \mathbf{m}^{eq}(x, t)) \tag{2.21}$$

onde $\mathbf{m}(x, t)$ e $\mathbf{m}^{eq}$ são vetores de quantidade de movimento, $\mathbf{m} = (m_0, m_1, m_2, ...m_n)^T$ e S é a diagonal da matriz de relaxação.

A relação entre a função de distribuição de partículas f e a quantidade de movimento $\mathbf{m}$ pode ser expressa com a ajuda de uma transformação da matriz M:

$$\mathbf{m} = Mf \quad \text{e} \quad f = M^{-1}\mathbf{m}. \tag{2.22}$$

Sendo que a matriz M para o *lattice* D2Q9 é:

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -1 & -1 & -1 & -1 & 2 & 2 & 2 & 2 \\ 4 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & -2 & 0 & 2 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & -2 & 0 & 2 & 1 & 1 & -1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \end{pmatrix}$$

A matriz inversa M^{-1} é:

$$M^{eq} = a \begin{pmatrix} 4 & -4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & -1 & -2 & 6 & -6 & 0 & 0 & 9 & 0 \\ 4 & -1 & -2 & 0 & 0 & 6 & -6 & -9 & 0 \\ 4 & -1 & -2 & -6 & 6 & 0 & 0 & 9 & 0 \\ 4 & -1 & -2 & 0 & 0 & -6 & 6 & -9 & 0 \\ 4 & 2 & 1 & 6 & 3 & 6 & 3 & 0 & 9 \\ 4 & 2 & 1 & -6 & -3 & 6 & 3 & 0 & -9 \\ 4 & 2 & 1 & -6 & -3 & -6 & -3 & 0 & 9 \\ 4 & 2 & 1 & 6 & 3 & -6 & -3 & 0 & -9 \end{pmatrix}$$

Onde $a = 1/36$ e o vetor da quantidade de movimento $\mathbf{m}$ é:

$$\mathbf{m} = (\rho, e, \epsilon, j_x, q_x, j_y, q_y, p_{xx}, p_{xy})^T. \tag{2.23}$$

O equilíbrio da quantidade de movimento $\mathbf{m}^{eq}$ é:

$$\begin{aligned} m_0^{eq} &= \rho \\ m_1^{eq} &= -2\rho + 3(j_x^2 + j_y^2) \\ m_2^{eq} &= -\rho - 3(j_x^2 + j_y^2) \\ m_3^{eq} &= j_x \\ m_4^{eq} &= -j_x \\ m_5^{eq} &= j_y \\ m_6^{eq} &= -j_y \\ m_7^{eq} &= (j_x^2 - j_y^2) \\ m_8^{eq} &= j_x j_y \end{aligned} \tag{2.24}$$

onde

$$\begin{aligned} j_x = \rho u_x = \sum_i f_i^{eq} e_i x \\ j_y = \rho u_y = \sum_i f_i^{eq} e_i y. \end{aligned} \tag{2.25}$$

A diagonal da matriz S é:

$$S = diag(1.0, 1.4, 1.4, s_3, 1.2, s_5, 1.2, s_7, s_8) \tag{2.26}$$

onde $s_7 = s_8 = 2/(1 + 6\nu)$, s_3 e s_5 são valores arbitrários e podem ser 1.0 (d'Humières, 2002; Mohamad, 2011). As equações do operador de colisão MRT em três dimensões são

análogas às equações em duas dimensões e podem ser encontradas em Mohamad (2011) ou d'Humières (2002).

Além dos operadores SRT e MRT, há um terceiro modelo conhecido como *Two-Relaxation Time* (TRT). Esse operador foi proposto por Ginzburg em 2005 e representa a função de distribuição de partículas em duas partes: a simétrica (Equação 2.27) e a anti-simétrica (Equação 2.28).

$$f_i^s = \frac{1}{2}(f_i + f_{-i}) \tag{2.27}$$

e

$$f_i^a = \frac{1}{2}(f_i + f_{-i}) \tag{2.28}$$

onde, $-i$ representa a direção inversa à i.

Dessa maneira, o operador de colisão TRT pode ser definido como:

$$\Omega_i = \frac{1}{\tau_s}(f_i^s(x,t) - f_i^{seq}(x,t)) + \frac{1}{\tau_a}(f_i^a(x,t) - f_i^{aeq}(x,t)) \tag{2.29}$$

onde τ_s e τ_a são os parâmetros de relaxação. Se os parâmetros de relaxação forem iguais, o operador TRT se reduz ao SRT.

Para o operador de colisão TRT, a propagação das partículas é a mesma do modelo SRT (Mohamad, 2011). Segundo Godenschwager et al. (2013), o operador TRT é mais preciso e estável que o modelo SRT, porém seu custo computacional é maior.

2.4.6 Algoritmo do Método de Lattice Boltzmann

O LBM é um método iterativo e, dessa forma, seu algoritmo é implemento sobre um laço de repetição. Nesse laço os passos de colisão e propagação das partículas, assim como as condições de contorno e o cálculo das grandezas mascroscópicas, são aplicados sobre cada célula do *lattice*. Há duas maneiras de implementar o algoritmo do LBM: Na primeira, conhecida como colisão-propagação (*collide-stream*), a colisão entre as partículas é realizada antes da propagação e, em seguida, as funções de distribuição são propagadas para as células vizinhas e, por fim, é realizado o cálculo das grandezas macroscópicas.

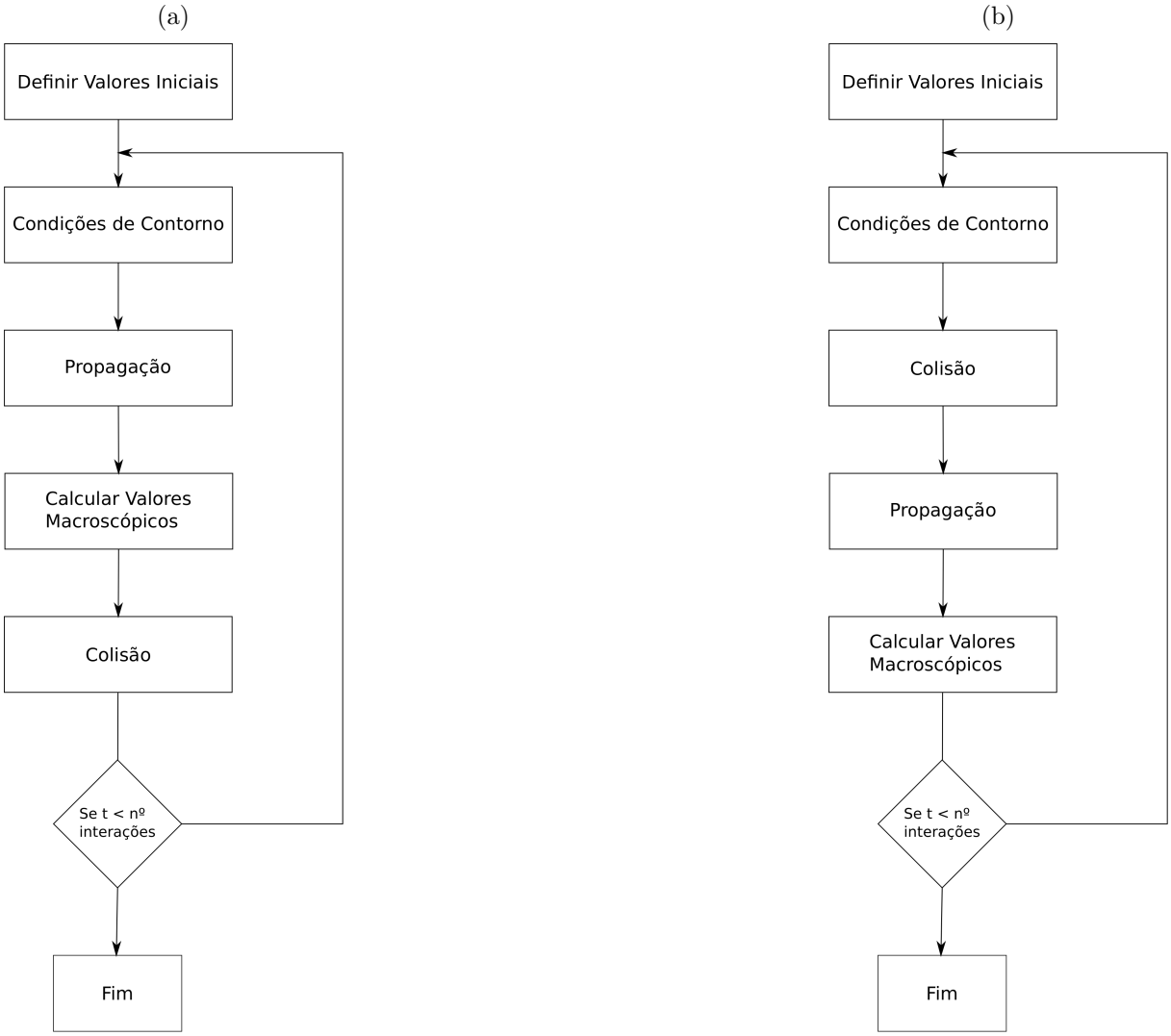
Já a segunda forma é conhecida como propagação-colisão (*stream-collide*) e a propagação ocorre primeiro, em seguida, é calculado os valores das grandezas macroscópicas e por último é realizado a colisão.

Para o LBM a ordem entre os passos de colisão e de propagação não importa, entretanto, quanto ao desempenho essa ordem é relevante. Segundo Habich et al. (2011a), a leitura dos dados de maneira desalinhada tem impacto menor no desempenho do algoritmo com relação a escrita desalinhada, ou seja, é melhor realizar o passo de propagação

primeiro, pois os dados são lidos das células vizinhas, isto é, de forma desalinhada e armazenados na célula atual de maneira ordenada. E, no algoritmo *stream-collide* a colisão é sempre realizada com os valores atuais de densidade e velocidade e, caso haja necessidade de plotar esses valores, eles estarão atualizados. A sequência do algoritmo *stream-collide* pode ser descrita da seguinte forma (Figura 2.13 (a)):

- Passo 1 - Definir valores iniciais para as grandezas mascroscópicas e para a função de distribuição de partículas. As PDFs são geralmente iniciadas com os valores da função de distribuição de equilíbrio pela Equação 2.16;
- Passo 2 - Calcular as funções de distribuição desconhecidas para as partículas da borda, de acordo com as condições de contorno;
- Passo 3 - Propagar a função de distribuição das partículas para as células vizinhas;
- Passo 4 - Calcular a densidade (ρ) e a velocidade ($\vec{u}$) para todas as partículas usando as Equações 2.15 e 2.16 respectivamente;
- Passo 5 - Calcular as colisões entre as partículas através dos operadores de colisão SRT, TRT ou MRT usando as Equações 2.13, 2.29 ou 2.17 respectivamente;
- Passo 6 - Retornar ao passo 2 até que o número de iterações seja executado.

Figura 2.13: Sequência dos passos dos algoritmos (a) *stream-collide* e (b) *collide-stream*.



Fonte: Autor.

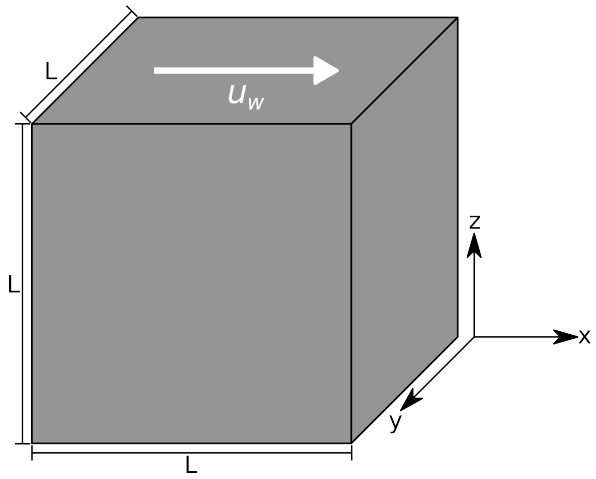
2.5 Lid-Driven Cavity

Para validação do modelo proposto foi utilizado o caso de testes conhecido como *lid-driven cavity* (Ghia et al., 1982). Esse caso de testes consiste em um fluxo laminar incompressível sobre uma cavidade quadrada, como pode ser visto na Figura 2.14. Essa cavidade possui uma borda superior que desliza com velocidade constante (u_w) em uma direção. As demais bordas do problema são estáticas.

Com o passar do tempo, as partículas que representam o fluido no LBM colidem com a borda superior fazendo com que sua velocidade seja propagada para as demais partículas do fluido. À medida em que o tempo passa, as partículas também colidem com as bordas estáticas fazendo com que o fluido forme um vórtice no centro da cavidade (Figura 2.15). *Lid-driven cavity* é um caso de testes bastante simples, porém muito utilizado em simulações computacionais em dinâmica dos fluidos para testar e avaliar

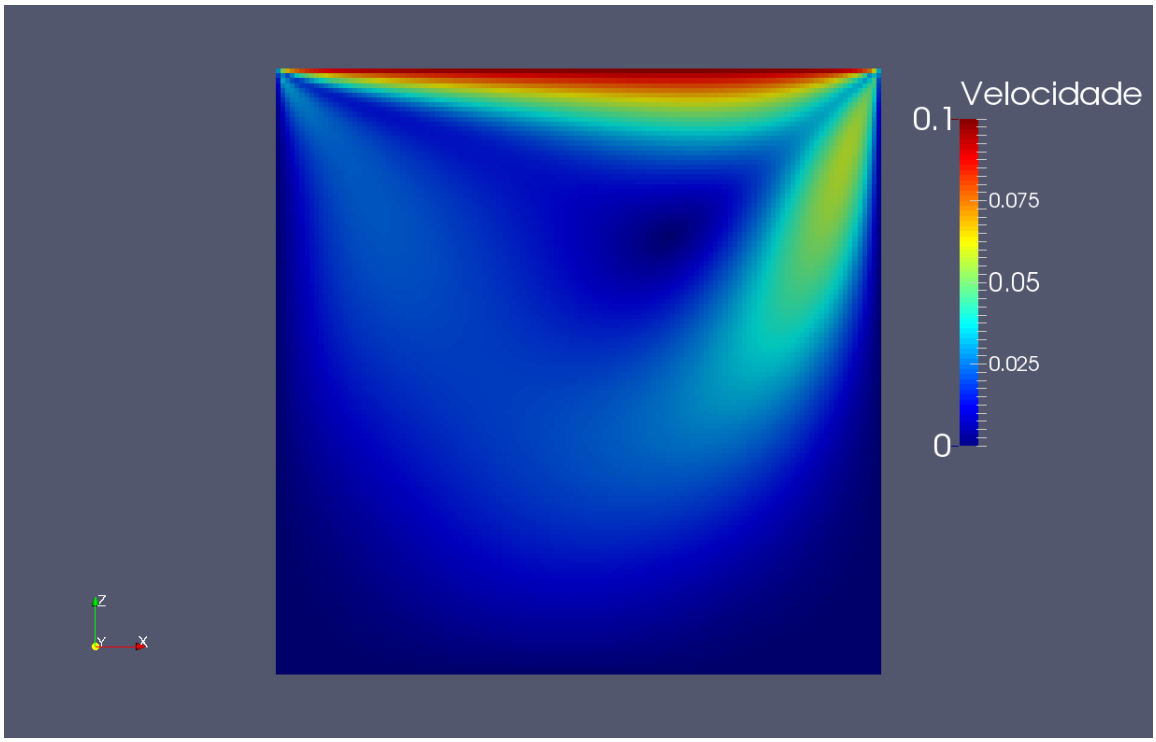
técnicas numéricas.

Figura 2.14: Arranjo do caso de testes *lid-driven cavity*.



Fonte: Autor.

Figura 2.15: Corte vertical sobre o ponto médio do eixo y. Imagem obtida após 10 mil iterações do algoritmo *lid-driven cavity* em GPU para um domínio de 128×128×128 células com número de Reynolds Re = 100 e com velocidade da borda superior $U_w = 0.1$.



Fonte: Autor.

2.6 CUDA

Em novembro de 2006, a empresa NVIDIA apresentou a tecnologia *Compute Unified Device Architecture* (CUDA) com o objetivo de atender a demanda por processamento de propósito geral em placas gráficas. CUDA é uma plataforma de computação paralela de propósito geral e um modelo de programação que aproveita a arquitetura das placas gráficas para resolver problemas computacionais de maneira altamente paralela.

O modelo de arquitetura paralela utilizado pelas GPUs é conhecido como *Single-Instruction, Multiple-Thread* (SIMT), ou seja, uma única instrução é executada paralelamente por diversas *threads* independentes (NVIDIA, 2009). Esse modelo é semelhante ao modelo *Single-Instruction, Multiple-Data* (SIMD), entretanto as instruções SIMT são específicas para execução e comportamento de uma única *thread* (NVIDIA, 2015a). A seguir será apresentado o modelo de programação em CUDA, os principais conceitos sobre as arquiteturas das GPUs e a hierarquia de memórias das placas gráficas.

2.6.1 Programação

O modelo de programação CUDA assume um sistema composto por *host* (CPU), *device* (GPU) e um conjunto de extensões das linguagens de programação C e C++. CUDA também introduz o conceito de *kernel*, definido como uma função executada N vezes em paralelo por um conjunto de N *threads*, ou seja, todas as *threads* executam o mesmo código simultaneamente.

Para definir um *kernel* em CUDA é necessário usar o qualificador __global__ (Código 2.1), isto é, define uma função que será executada apenas no *device*. Junto com a definição do *kernel* é necessário configurar a maneira com o *kernel* será executado na GPU. Para isso, é utilizada a expressão <<<DG, DB>>> entre o nome da função *kernel* e seus parâmetros, como mostra o trecho de Código 2.2, onde DG define a dimensão da *grid* e DB a dimensão do bloco. A dimensão DG representa a quantidade de blocos em cada *grid* e DB a quantidade de *threads* em cada bloco.

Listing 2.1: Definição do *kernel*.

```
1 __global__ void kernel( parâmetros_do_kernel ) {}
```

Listing 2.2: Exemplo de chamada do *kernel* no *host*.

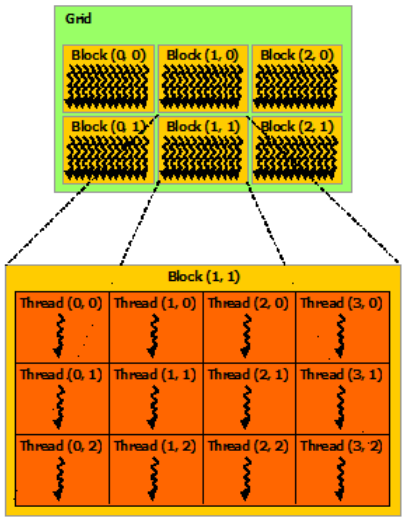
```
1 kernel <<< DG, DB >>> ( parâmetros_do_kernel );
```

CUDA possui uma hierarquia de grupos de *threads* na qual um conjunto de *threads* é organizado em forma de blocos. *Threads* do mesmo bloco podem cooperar através do compartilhamento de informação, por meio da memória compartilhada, e sincronizar sua

execução pelo acesso coordenado à memória através de *__syncthreads()*. Essa sincronização atua como uma barreira na qual as *threads* do mesmo bloco devem aguardar até que todas as *threads* tenham terminado sua execução. Cada *thread* executa uma instância do *kernel* e possui um identificador único, acessado através da variável $threadIdx$.

Os blocos, por sua vez, são organizados em *grids* de uma, duas ou três dimensões, como ilustrado na Figura 2.16 e também possuem um identificador que pode ser acessado através da variável $blockIdx$. A dimensão de um bloco pode ser acessada pela variável $blockDim$. O número de blocos que será utilizado em um *grid* é geralmente definido pela quantidade de dados a serem processados ou pelo número de processadores no sistema (NVIDIA, 2015a).

Figura 2.16: *Grid* de blocos de *threads*.



Fonte: NVIDIA (2015a).

2.6.2 Arquiteturas

As arquiteturas das GPUs NVIDIA são construídas em torno de multiprocessadores conhecidos como *Streaming Multiprocessor* (SM). Cada SM possui vários núcleos de processamento (*cores*), memória compartilhada, memória *cache* e escalonadores e são responsáveis pela execução dos blocos de *threads* na GPU. Essas *threads* são criadas, executadas e escalonadas dentro de cada SM em grupos de 32 *threads* chamados de *warps*. Cada *warp* executa uma única instrução por vez, com isso, é necessário que todas as *threads* do mesmo *warp* executem a mesma instrução a fim de se obter um melhor desempenho. Caso isso não ocorra, as instruções serão serializadas e o desempenho será prejudicado (NVIDIA, 2015a).

A primeira arquitetura de GPU com suporte à CUDA foi a arquitetura G80. Essa arquitetura foi a primeira com suporte à linguagem de programação C, permitindo aos

programadores usarem as placas gráficas sem a necessidade de aprender uma nova linguagem, também introduziu o modelo de execução SIMT, memória compartilhada, para a comunicação entre *threads*, e barreiras de sincronização.

A segunda geração de arquiteturas de GPUs NVIDIA, conhecida como GT200, foi apresentada em 2008 e é uma evolução da arquitetura G80. Entre as principais melhorias está o aumento na quantidade de CUDA *cores*, acesso a memória mais eficiente, pois essa nova arquitetura permite o acesso coalescente a memória da GPU, e foi a primeira arquitetura com suporte a ponto flutuante de precisão dupla (NVIDIA, 2009).

As arquiteturas Fermi, Kepler e Maxwell, sucessoras à GT200, serão apresentadas a seguir, assim como suas principais características.

2.6.2.1 Arquitetura Fermi GF100

Fermi é uma arquitetura de GPU que destaca-se em processamento gráfico e computação de alta performance (NVIDIA, 2014a, 2009). As principais melhorias da arquitetura Fermi, em relação a arquitetura anterior GT200, são:

- Aumento na performance de operações de ponto flutuante de dupla precisão em até 8x;
- Aumento de 4x a quantidade de CUDA *cores*;
- Permite a execução de *kernels* em paralelo;
- Realiza troca de contexto 10x mais rápida;
- Primeira arquitetura com suporte a ECC [1] (*Error Check and Correction*);
- Primeira arquitetura com suporte completo a linguagem C++.

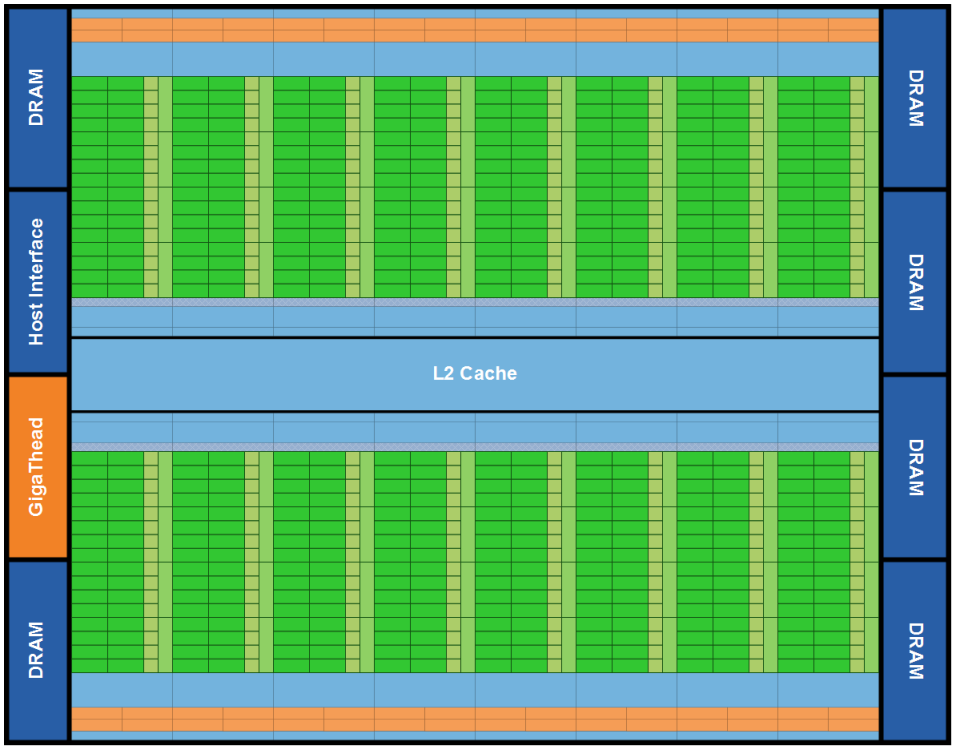
Na arquitetura Fermi as instruções de adição e multiplicação foram unidas para diminuir a perda de precisão (*Fused Multiply-Add*), sendo assim mais preciso do que as instruções realizadas separadamente. Cada SM possui 64 KB de memória que pode ser configurada como 48KB de memória compartilhada com 16KB de *cache* L1 e vice-versa. Isso aumenta o desempenho de aplicações que fazem uso intensivo de memória compartilhada e, por outro lado, aplicações que não fazem uso de grande quantidade de memória compartilhada são beneficiadas por terem uma memória *cache* maior (NVIDIA, 2014a, 2009).

A Figura 2.17 mostra a arquitetura Fermi GF100 com 16 SMs posicionados em torno da *cache* L2. Seus principais recursos incluem seis partições de memória de 64-bit para

[1] Permite a detecção e correção de erros de *software* que possam acontecer no armazenamento dos dados em qualquer tipo de memória da GPU.

uma interface de memória de 384-bit com suporte até 6GB de memória DRAM (GDDR5) e um escalonador *GigaThread* (NVIDIA, 2014a, 2009).

Figura 2.17: Arquitetura Fermi GF100.



Fonte: NVIDIA (2014a).

2.6.2.2 Arquitetura Kepler GK110

A arquitetura Kepler possui maior poder de processamento que sua geração anterior (Fermi) fornecendo mais de 1 TFlop de dupla precisão em operações gerais de multiplicação de matrizes (DGEMM) atingindo até 80% de eficiência contra 60-65% da arquitetura Fermi. Além de alta performance, a arquitetura Kepler oferece um eficiente consumo de energia sendo até 3x a performance por watt em relação a Fermi (NVIDIA, 2014b). A seguir os novos recursos da arquitetura Kepler são apresentados:

- *Dynamic Parallelism*: foi introduzido na arquitetura Kepler GK110 e permite que a GPU gere trabalho para si mesma, sincronize os resultados e controle os escalonadores de trabalho sem o envolvimento da CPU. Isso permite que *kernels* possam lançar outros *kernels*.

- *Hyper-Q*: Permite que múltiplos núcleos da CPU lancem trabalho para uma única GPU simultaneamente, assim aumentando a utilização da GPU e reduzindo o tempo ocioso da CPU. Isso ocorre porque *Hyper-Q* aumenta o número total de conexões

entre o *host* e o *device* permitindo até 32 conexões simultâneas comparada com uma única na arquitetura Fermi.

- *Grid Management Unit*: Esse recurso foi projetado para melhorar a carga de trabalho da CPU para a GPU, visto que na arquitetura Fermi uma *grid* de blocos de *threads* deveria ser lançado apenas pela CPU criando um único fluxo de trabalho do *host* para o *device*. A nova arquitetura permite que a GPU gerencie eficientemente a carga de trabalho criada pela CPU e pelo CUDA, assim uma nova *grid* também pode ser criada através de programação CUDA dentro de um SM.

- NVIDIA *GPUDirect*: Permite que GPUs dentro de um mesmo computador, ou em diferentes nós em um *cluster*, troquem diretamente dados sem a necessidade de interação com a CPU ou sistemas de memória.

A arquitetura Kepler foi construída com a nova arquitetura de multiprocessadores SMX com 192 CUDA *cores*. Além disso, oferece capacidade de *cache* adicional, com a adição de um *cache* de somente leitura com 48KB, maior largura de banda em cada nível de arquitetura e uma implementação I/O DRAM mais rápida. Todos esses recursos estão organizados na nova arquitetura Kepler GK110, como pode ser visto na Figura 2.18, que inclui 15 unidades SMX, seis controladores de memória de 64-bit e *cache* L2.

Figura 2.18: Arquitetura Kepler GF100.

Fonte: NVIDIA (2014b).

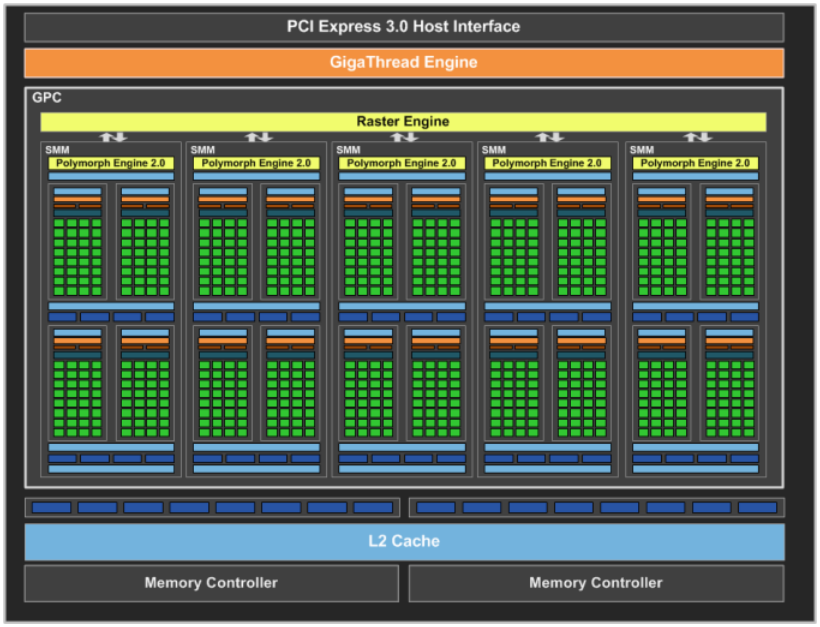
2.6.2.3 Arquitetura Maxwell GM107

A arquitetura Maxwell possui duas gerações de placas gráficas, porém nessa seção será apresentada apenas a primeira geração (GM107). A principal melhoria da arquitetura

Maxwell é o consumo mais eficiente de energia comparado a arquitetura anterior (Kepler). Isso só foi possível graças a arquitetura dos novos *streaming multiprocessores*, conhecidos como SMM (Figura 2.19).

Outra mudança significativa, observada na Figura 2.19, é a redução da quantidade de CUDA cores por *streaming multiprocessors*. Essa redução garante um melhor escalonamento das instruções, pois esse escalonamento se torna mais simples e permite que cada escalonador *warp* (*warp scheduling*) lance instruções no tamanho do *warp*. Mesmo com a redução na quantidade de CUDA cores a arquitetura Maxwell mostrou maior performance e consumo mais eficiente de energia que as arquiteturas anteriores (NVIDIA, 2015d, 2014c).

Figura 2.19: Arquitetura Maxwell GM107.



Fonte: NVIDIA (2015d).

A arquitetura Maxwell é similar em muitos aspectos a arquitetura Kepler, além de melhorias voltadas ao aumento na eficiência e baixo consumo de energia. Entre essas melhorias podemos citar: aumento do *cache* L2 de 256KB para 2048KB, com mais *cache* o acesso a memória global é reduzido e dessa maneira a performance aumenta, pois a latência da memória *cache* é inferior a latência da memória global. Além do aumento na *cache* L2, o número máximo de blocos ativos por multiprocessador dobrou, com relação a SMX, para 32 resultando em uma melhora na ocupação automática para *kernels* que usam blocos de tamanho menor que 64 *threads* ou *kernels* que usam poucas *threads*. Também foram realizadas melhorias no controle lógico de particionamento, no balanceamento de carga e aumento do número de instruções geradas por ciclo de *clock* (NVIDIA, 2015d, 2014c).

Diferente das arquiteturas anteriores, a arquitetura Maxwell possui 64KB de memória

compartilhada dedicada por SMM. Esse aumento pode levar a uma melhora na taxa de ocupação e só foi possível pela combinação da funcionalidade dos *caches* L1 e de textura em uma única unidade. Além da tradicional memória compartilhada, a nova arquitetura Maxwell fornece também memória compartilhada para operações atômicas para inteiros de 32-bits (NVIDIA, 2015d, 2014c).

A Tabela 2.4 mostra os principais recursos encontrados nas arquiteturas apresentadas anteriormente.

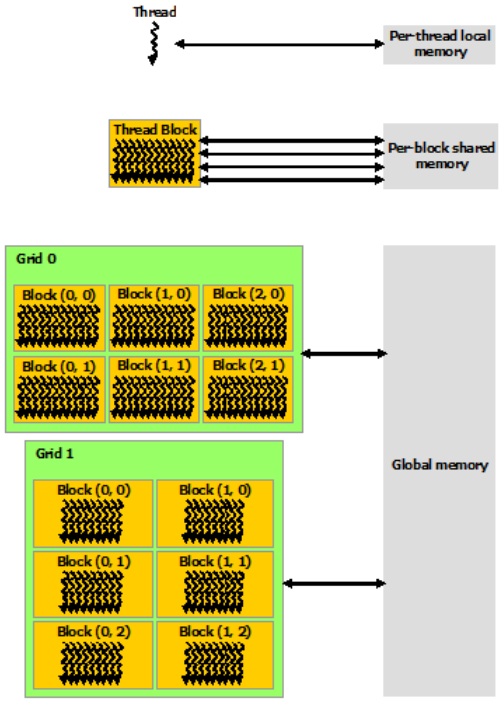
Tabela 2.4: As principais propriedades de cada arquitetura.

GPU	Tesla C2075	Tesla K40m	GTX 750 Ti
Arquitetura	Fermi	Kepler	Maxwell
Compute Capability	2.0	3.5	5.0
Qtde. de Memória Global (GB)	5.25 (ECC = ON) 6.0 (ECC = OFF)	11.2 (ECC = ON) 12.0 (ECC = OFF)	2.0
CUDA Cores	448	2880	960
GPU Clock (MHz)	1147	745	1084
Clock Memória (MHz)	1566	3004	2700
Largura de Banda de Memória	384-bit	384-bit	128-bit
Tamanho Cache L2 (KB)	768	1536	2048
Qtde. Memória Constante (KB)	64	64	64
Qtde. de Memória Compartilhada por Bloco (KB)	48	48	48
Qtde. de Registradores por Bloco	32768	65536	65536
Warp size	32	32	32
Número Máximo de threads por Multiprocessador	1536	2048	2048
Número Máximo de threads por Bloco	1024	1024	1024
Performance máxima de Dupla precisão (Gflops)	515	1464.3	40.8
Suporte à ECC	Sim	Sim	Não

2.6.3 Hierarquia de Memórias

Nas GPUs NVIDIA as *threads* podem acessar os dados em múltiplos espaços de memória durante sua execução, como ilustrado na Figura 2.20. Cada *thread* possui uma memória local privada, cada bloco de *threads* possui uma memória que pode ser compartilhada por todas as *threads* do mesmo bloco e todas as *threads* tem acesso à memória global da GPU. Há também outros dois espaços de memórias que podem ser acessados por todas as *threads*: memória constante e memória de textura.

Figura 2.20: Hierarquia de memória.



Fonte: NVIDIA (2015a).

2.6.3.1 Memória Constante

Memória constante possui tamanho de 64KB e é declarada com o qualificador __constant__ que restringe seu uso para somente leitura. Quando dados são lidos da memória constante pode haver conservação da largura de banda quando comparado com o mesmo dado lido da memória global. Segundo Sanders (Sanders and Kandrot, 2010), há duas razões para ler da memória constante:

- Uma simples leitura da memória constante por uma *thread* pode ser enviada para outras *threads* que estão próximas à *thread* que realiza a leitura, e economizar até 15 leituras. Isso ocorre porque quando se trata de manipulação de memória constante o *hardware* da GPU envia uma simples leitura de memória para cada *half-warp*, ou seja, 16 *threads*. Se toda *thread* em um *half-warp* recebe a informação do mesmo endereço a GPU irá gerar apenas uma única leitura na memória constante e reduzir assim o acesso a memória;

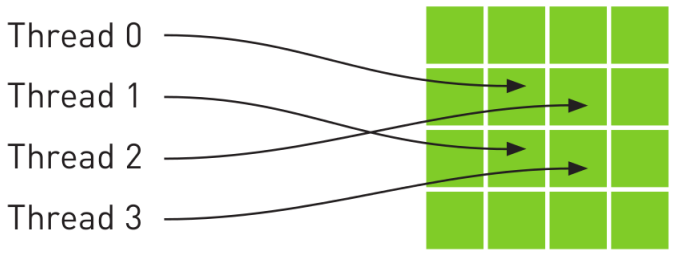
- Memória constante é armazenada em *cache*, assim consecutivas leituras do mesmo endereço não irão implicar em um tráfico de memória adicional.

2.6.3.2 Memória de Textura

Como a memória constante, a memória de textura é armazenada em *cache* somente de leitura, por isso, em algumas situações ela irá fornecer uma maior largura de banda, redu-

zindo os acessos a memória global. Especificamente, os *caches* de textura são projetados para aplicações gráficas onde os padrões de acesso à memória apresentam uma grande quantidade de localidade espacial. Em uma aplicação computacional, isso implica que uma *thread* fará a leitura de um espaço de memória próximo ao endereço que uma *thread* vizinha leu, como mostra a Figura 2.21. Ao contrário dos tradicionais *caches* de CPU que possuem endereços de *cache* de forma sequencial, a memória de textura é otimizada para localidade espacial de duas dimensões (Sanders and Kandrot, 2010).

Figura 2.21: Mapeamento das *threads* de uma região bidimensional da memória.



Fonte: Sanders and Kandrot (2010).

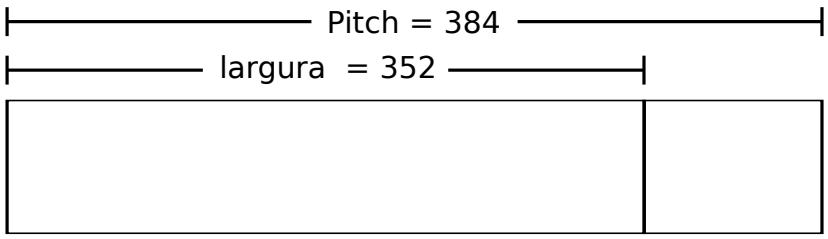
2.6.3.3 Memória Alinhada

Além da função *cudaMalloc()* para alocação de memória global na GPU, CUDA também possui a função *cudaMalloc3D()* que pode ser usada quando houver a necessidade de alocar *arrays* em três dimensões. A função *cudaMalloc3D()* aloca, pelo menos, uma quantidade de N_x, N_y e N_z *bytes* de memória linear, onde N_x, N_y e N_z representam a largura, a altura e a profundidade do domínio, respectivamente e retorna um ponteiro (*cudaPitchedPtr*) referente a memória alocada. Entretanto, essa função adiciona *bytes* à memória alocada para garantir o alinhamento de memória, caso seja necessário. Esse alinhamento é apenas realizado na dimensão x do domínio.

Conforme apresentado na Seção 2.6.2, o acesso à memória global da GPU é realizado através do *warp*, dessa maneira, o alinhamento dos dados contribui para o acesso coalescente a memória da GPU, aumentando assim o *throughput* da memória e, com isso, a performance.

A Figura 2.22 mostra como é feito o processo de alinhamento de memória na GPU, onde o tamanho do *array* é alinhado até o próximo múltiplo de 64 *bytes* antes de alocar memória. O termo *pitch*, que aparece na Figura 2.22, se refere a quantidade de memória total alocada, ou seja, o tamanho do vetor (352 *bytes*) mais o *padding*. O ponteiro retornado pela função *cudaMalloc3D()* é na verdade uma *struct*, que possui um ponteiro para memória alocada (*ptr*), além dos campos *xsize* e *ysize*, que são os valores lógicos da largura e altura da memória alocada (Farber, 2009; NVIDIA, 2015c; Wilt, 2013).

Figura 2.22: Memória alinhada.



Fonte: Adaptado de (Wilt, 2013).

2.7 Framework waLBerla

waLBerla (widely applicable Lattice Boltzmann solver from Erlangen) é um *framework* massivamente paralelo desenvolvido pelo Laboratório de Simulação de Sistemas (LSS) [2] do departamento de informática da universidade de Erlangen-Nürnberg na Alemanha para simulações de fluidos que utilizam o método de Lattice Boltzmann (Götz et al., 2007). O *framework* waLBerla atualmente está na terceira versão, a qual não possui suporte à GPU, porém a versão anterior do *framework* era desenvolvida especialmente para GPUs. Entretanto, a segunda versão não foi mais mantida, pois não permitia computação heterogênea. A atual versão do *framework*, desenvolvida na linguagem C++, é capaz de simular vários fenômenos físicos como: fluxo em um canal aberto, fluxo em um canal aberto com objetos flutuando, fluxo através de meios porosos (Donath et al., 2011) e permite ainda, simular o processo de coagulação em vasos sanguíneos, como apresentado no trabalho de Feichtinger et al. (2011).

Além desses aspectos, outras características relevantes do *framework* são sua escalabilidade e alto desempenho, como visto no trabalho de Godenschwager et al. (2013), alcançando a marca de 1,93 trilhões de células de fluidos atualizadas por segundo usando 1,8 milhões de *threads* em supercomputadores como o JUQUEEN (TOP500, 2015a) e o SuperMUC (TOP500, 2015b) para uma simulação do fluxo sanguíneo.

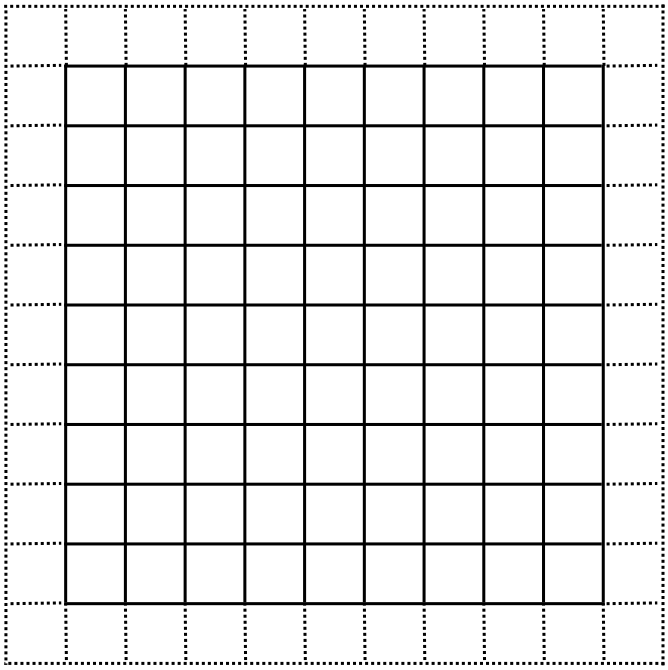
Outra característica importante do *framework* é sua capacidade em controlar centenas de milhares de processadores, ou seja, o *framework* waLBerla foi desenvolvido para lidar com grandes problemas. Além dessa característica, o *framework* waLBerla faz uso intensivo de *templates* em suas classes para permitir com que os dados sejam definidos em tempo de compilação, dessa forma mantém a flexibilidade do *framework* sem perder desempenho (Godenschwager et al., 2013).

No *framework* waLBerla o domínio da simulação é estruturado em blocos, isto é, todo o espaço da simulação é subdividido em blocos de tamanhos iguais. Esses blocos possuem uma *grid* de células uniformes que é estendida em cada direção por uma camada adicional de células conhecida como *ghost layer*, como mostra a Figura 2.23.

[2] https://www10.informatik.uni-erlangen.de/

A camada *ghost layer* é utilizada em cada passo da iteração, durante o processo de comunicação entre os blocos, para sincronizar os valores das células das bordas, ou seja, cada *ghost layer* recebe os valores da camada mais externa (linhas contínuas da Figura 2.23) do bloco vizinho. Após essa sincronização, apenas os valores armazenados nas células do *ghost layer* são enviados e recebidos durante a comunicação entre os blocos. Isso é feito para reduzir o acesso à todas as células dos blocos adjacentes a cada iteração. Após a divisão do domínio, cada bloco pode ser atribuído a um processador disponível, utilizando MPI, ou todos os blocos podem ser atribuídos a um mesmo processador. Além da paralelização utilizando MPI o *framework* waLBerla também possui suporte a OpenMP (Godenschwager et al., 2013; waLBerla, 2014).

Figura 2.23: Um bloco com 9x9 células mais uma camada adicional onde as células tracejadas representam a camada *ghost layer*.



Fonte: Autor.

Todo algoritmo ou simulação no waLBerla é executado sobre um laço de repetição principal, definido como *time loop*. Esse *loop* é um componente central de cada simulação do *framework*, pois executa a cada iteração passos de trabalho (*work steps*), chamados de *sweep*. Cada *sweep* consiste em três partes principais: pré-processamento, processamento e pós-processamento. A etapa de pré-processamento é responsável pela comunicação dos dados necessários pela simulação, enquanto que a etapa de pós-processamento, pode ser usada para visualização e análise dos dados. Já a segunda etapa é a parte principal do *sweep*, pois é responsável pelo processamento dos dados da simulação.

Um exemplo de *sweep* é o passo de colisão do LBM, entretanto qualquer passo do algoritmo, descrito na Seção 2.4.6, pode ser considerado um *sweep*. Geralmente esses passos de trabalho são operações executadas sobre um único bloco do domínio. No *framework*

waLBerla é possível criar e compartilhar *sweeps* entre diferentes aplicações a fim de evitar a duplicação de código. Essa é outra característica importante do waLBerla, e do ponto de vista de engenharia de software, essa característica é definida como manutenibilidade (Feichtinger et al., 2011; Feichtinger, 2012).

Quando o domínio da simulação é dividido em vários blocos é necessário realizar a troca de dados entre eles, ou seja, é necessário realizar a comunicação entre blocos. Para que essa comunicação seja possível, o *framework* possui um mecanismo responsável pela comunicação, composto por duas partes: A primeira parte, chamada de "PackInfo", é implementada em uma classe responsável por extrair os dados da camada *ghost layer* e empacotá-los em uma mensagem. Após isso, essa mensagem é enviada aos blocos vizinhos e desempacotada pela mesma classe, onde os dados recebidos formam as respectivas camadas de *ghost layer*.

A segunda parte do mecanismo de comunicação é chamada de "scheme", e é responsável por controlar todo o processo de comunicação, ou seja, é ela quem gerencia o envio, o empacotamento e o desempacotamento das mensagens. Essas mensagens são enviadas usando as bibliotecas MPI. Além disso, ela também é responsável por controlar quais os processos que devem comunicar uns com os outros (waLBerla, 2014).

O tratamento correto das condições de contorno é parte fundamental nas simulações de fluidos, pois elas podem ter influência direta no resultado da simulação. Além disso, as condições de contorno dependem diretamente do tipo de simulação que está sendo realizada. O *framework* waLBerla possui várias condições de contorno implementadas, como: *inflow*, *outflow*, paredes sólidas e fontes de calor. Essas condições podem ser compartilhadas, isto é, diferentes simulações podem usar as mesmas condições de contorno, uma vez que muitas simulações utilizam o mesmo tratamento de bordas.

Para determinar o tipo de cada borda, o *framework* waLBerla utiliza *flags*. Na inicialização da simulação, as *grids* de células contidas em cada bloco são percorridas e cada célula é marcada de acordo com seu tipo. Essas *flags* podem ser, por exemplo, fluidos ou paredes sólidas. No waLBerla as condições de contorno são sempre executadas antes ou depois da execução do *sweep* principal, fazendo com que o *sweep* se torne independente de suas bordas (Feichtinger, 2012).

Além de fornecer toda a estrutura necessária para simulações de fluidos que utilizam o LBM, o *framework* waLBerla fornece ainda ferramentas e módulos que auxiliam as simulações. Entre elas, podemos citar um módulo para leitura de arquivos de configuração, que carrega os parâmetros de configuração de uma determinada simulação, como: densidade e velocidade, a partir de um arquivo. O mesmo módulo também gera arquivos de saída para visualização dos dados da simulação.

Uma funcionalidade importante para um *framework* é a ferramenta de *logs*. Para isso, o *framework* waLBerla oferece ferramentas para reportar *logs* de erro e *warnings*, além de *logs* que exibem o progresso da simulação. Além dessas ferramentas, o *framework* possui

módulos que permitem medir o tempo gasto (*wall-clock time*) em uma determinada parte do algoritmo, por exemplo, uma função ou *sweep*, ou ainda, medir o tempo total da simulação. Além desse módulo, há também o módulo que permite medir o desempenho da simulação em MLUPS (Feichtinger, 2012). Mais sobre o cálculo de MLUPS será visto no capítulo 5.

No *framework* waLBerla não são apenas as simulações que suportam múltiplos processadores, mas também o módulo que gera a saída com os resultados da simulação. Como o domínio é dividido em vários blocos, onde cada bloco pode ser atribuído a um processo, uma opção para criar esse arquivo de saída é utilizar um processo principal (*master*), que recebe as informações necessárias dos demais processos e escreve em um único arquivo. Entretanto, isso leva a uma enorme comunicação entre os processos fazendo com que o desempenho diminua mesmo se o arquivo de saída é raramente escrito. Além do mais, um único arquivo pode ser tornar muito grande.

Para evitar isso e reduzir a comunicação entre os processos, o *framework* waLBerla, cria vários arquivos de saída no formato *vtu* que são referenciados por um único arquivo no formato *pvd*. Durante a simulação cada processo cria seu próprio arquivo de dados com as informações do seu bloco e o processo *master* cria apenas um arquivo *pvd* com os nomes de todos os arquivos *vtu*, dessa maneira todas as informações da simulação estão agregadas em um arquivos de saída. Assim, apenas durante a configuração uma comunicação global é necessária, enquanto que durante a simulação não há troca de dados entre os processos (Feichtiner et al., 2007). Para analisar o arquivo de saída é utilizado o *software* ParaView.

ParaView é um *software open-source* utilizado para visualização e análise de dados. Pode ser aplicado em diversas áreas como: dinâmica de fluidos computacional, astrofísica, robótica e medicina. Além disso, suporta computação paralela em ambientes com memória compartilhada e distribuída (ParaView, 2015). No *framework* waLBerla, além de gerar os arquivos com os dados da simulação, o *framework* converte esses dados em grandezas físicas como densidade, velocidade e pressão do fluido, o que é muito útil para validação da simulação.

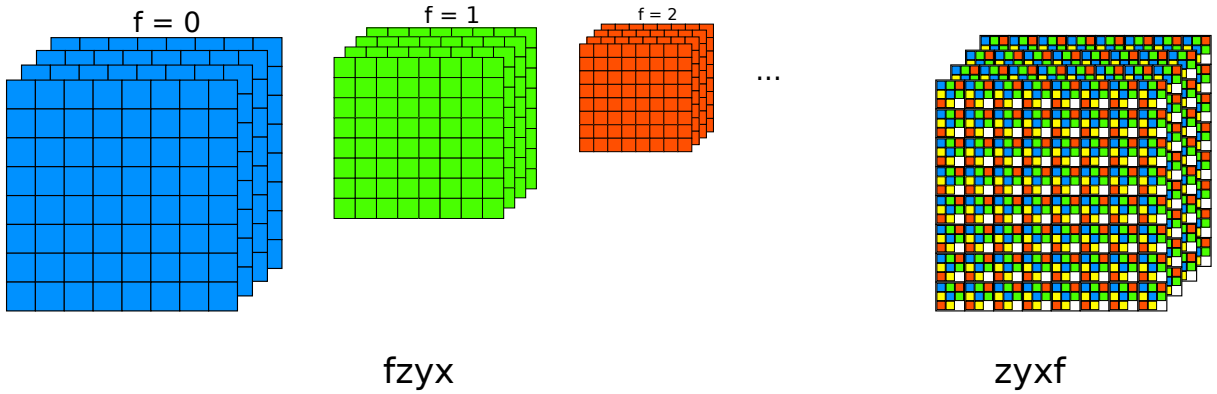
O *framework* waLBerla possui interface gráfica, porém ela permite apenas a visualização de simulações em duas dimensões. Entretanto, é útil para depuração da simulação, pois permite a visualização de simulações em pequena escala, além de permitir uma compreensão e visualização do domínio dividido em blocos, da *grid* de células, contida em cada bloco, e dos dados armazenados em cada célula. Para fazer uso da interface gráfica é necessário a biblioteca Qt.

Além da interface gráfica para depuração, há também o modo *Debug*. O modo *Debug* é responsável por verificar se os índices de acessos aos dados da simulação estão dentro dos limites, detectar possíveis ponteiros não inicializados e monitorar as grandezas físicas como densidade e velocidade. Esse monitoramento ocorre durante a leitura de um arquivo de configuração e antes de serem gravados em um arquivo de saída para verificar se os

valores estão dentro dos limites válidos. Quando o modo *Debug* encontra algum tipo de erro, a simulação é encerada. Além do modo *Debug*, o *framework* waLBerla possui o modo *Release*, ambos os modos são configurações de compilação (Feichtinger, 2012).

O *framework* waLBerla suporta ainda dois *layouts* de dados: *Structure-of-Array* (SoA) e *Array-of-Structure* (AoS), representados na Figura 2.24 por *fzyx* e *zyxf*, respectivamente. Na Figura 2.24 cada cor representa uma PDF. Esses *layouts* são as principais formas de organizar os dados em simulações do LBM. É importante observar, que a diferença entre os *layouts* não está apenas na forma de organizar os dados, mas também no desempenho das simulações. Para a CPU, que é uma arquitetura baseada em *cache*, o *layout* AoS apresenta desempenho melhor, pois as PDFs das células do *lattice* são armazenadas de maneira contígua na memória e, portanto, poucas linhas de *cache* são necessárias para acessar os dados. Dessa maneira, a leitura é beneficiada, porém os dados serão armazenados em espaços não contíguos na memória o que resulta em várias linhas de *cache* (Körner et al., 2005). Já para a GPU, o melhor *layout* é SoA, pois, dessa maneira, garante o acesso coalescente à memória global da GPU (Obrecht et al., 2011).

Figura 2.24: Layout dos dados em memória.



Fonte: Autor.

Além do que já foi apresentado sobre o *framework*, há também outros conceitos relevantes, do ponto de vista de implementação, que serão apresentados a seguir:

2.7.1 Field

Como visto anteriormente, cada bloco do domínio da simulação é formado por uma *grid* de células. As células de cada *grid* podem armazenar qualquer tipo de dado, como: valores inteiros, valores de ponto flutuante e caracteres. Porém, para simulações do LBM é necessário armazenar os dados em um *lattice*, ou seja, os valores das funções de distribuição de partículas, como explicado na Seção 2.4.2. Dessa maneira, para simulações do LBM no *framework* waLBerla cada bloco possui uma estrutura de *lattice*. Entretanto essa estrutura no *framework* é definida como *field*.

No *framework* a classe responsável por armazenar e manipular os *fields* é chamada de *Field*. Essa classe é definida por uma estrutura de quatro dimensões, onde as três primeiras dimensões correspondem as coordenadas cartesianas x, y, e z, enquanto que a quarta coordenada é geralmente utilizada para indexar os índices das PDFs de acordo com o modelo de *lattice* (waLBerla, 2014).

A classe *Field* foi implementada usando o conceito *container* estilo vector da linguagem C++ (C++, 2015b), que permite com que os dados sejam armazenados de maneira consecutiva na memória e fornece tempo constante de acesso a qualquer dado. Os dados manipulados pela classe *Field* podem ser armazenados em memória de duas maneiras distintas de acordo com o *layout* escolhido.

Os principais métodos dessa classe são aqueles que permitem o acesso aos dados (*get()*), iteradores para percorrer os índices em memória (*begin()* e *end()*, por exemplo), métodos que fornecem informações sobre o tipo do *layout* (*layout()*) e a quantidade de células em cada direção (*xSize()* e *ySize()*, por exemplo). Com relação aos atributos principais podemos citar o ponteiro para memória alocada, número de células em cada direção e o modelo de *layout* (waLBerla, 2014).

Outro detalhe importante é que a maioria das classes no *framework* waLBerla, são classes *templates* (C++, 2015a). A classe *Field* possui dois parâmetros de *template*, o primeiro específica o tipo de dado que será armazenado (T) e o segundo o tamanho da quarta dimensão (fSize_). Apenas os principais atributos e métodos da classe *Field* foram inseridos no diagrama de classe, que pode ser visto na Figura 2.25.

2.7.2 GhostLayer

Como cada bloco do *framework* waLBerla pode ser atribuído a um processador diferente, surge então a necessidade de comunicação entre eles, visto que o LBM depende dos valores dos PDFs das células vizinhas. A maneira como a comunicação entre os blocos é realizada já foi apresenta anteriormente. Entretanto, além da comunicação, há a necessidade de criar uma classe que manipule essas estruturas, para isso o *framework* waLBerla possui a classe *GhostLayerField*.

Do ponto de vista de orientação à objetos, a classe *GhostLayerField* é uma especialização da classe *Field*, ou seja, a classe *GhostLayerField* herda as propriedades e os métodos da classe *Field*, porém possui ainda características próprias. A diferença principal entre essas classes, é que a classe *GhostLayerField* possui métodos que manipulam a camada mais externa de cada bloco, ou seja, o *ghost layer*.

Entre os principais métodos podemos destacar iteradores que percorrem todas as células, inclusive as células da camada *ghost layer*, e também métodos que percorrem apenas a camada *ghost layer*, o que é muito útil durante a comunicação entre blocos (waLBerla, 2014). O único atributo da classe *GhostLayerField* é a quantidade de ca-

madas de *ghost layer* e os parâmetros de *templates* são os mesmos da classe *Field*. A representação da classe pode ser vista no diagrama da Figura 2.25.

2.7.3 PdfField

As duas classes apresentadas nos parágrafos anteriores permitem que o usuário armazene qualquer tipo de dado nas células de cada *grid*. Com isso, é possível criar diversas simulações como o exemplo do *Game of Life*, o qual é implementado no waLBerla, como um tutorial, usando a classe *GhostLayerField*. Entretanto, para simulações do LBM, o *framework* waLBerla possui uma classe específica: a classe *PdfField*.

A classe *PdfField* herda a classe *GhostLayerField*, porém apresenta ainda um conjunto de novos métodos necessários para simulações do LBM, como: métodos para calcular as gradezas macroscópicas e um método para determinar a função de distribuição de equilíbrio (waLBerla, 2014). A classe *PdfField* recebe um único parâmetro de *template*, porém esse parâmetro é um tipo de dado definido da seguinte maneira:

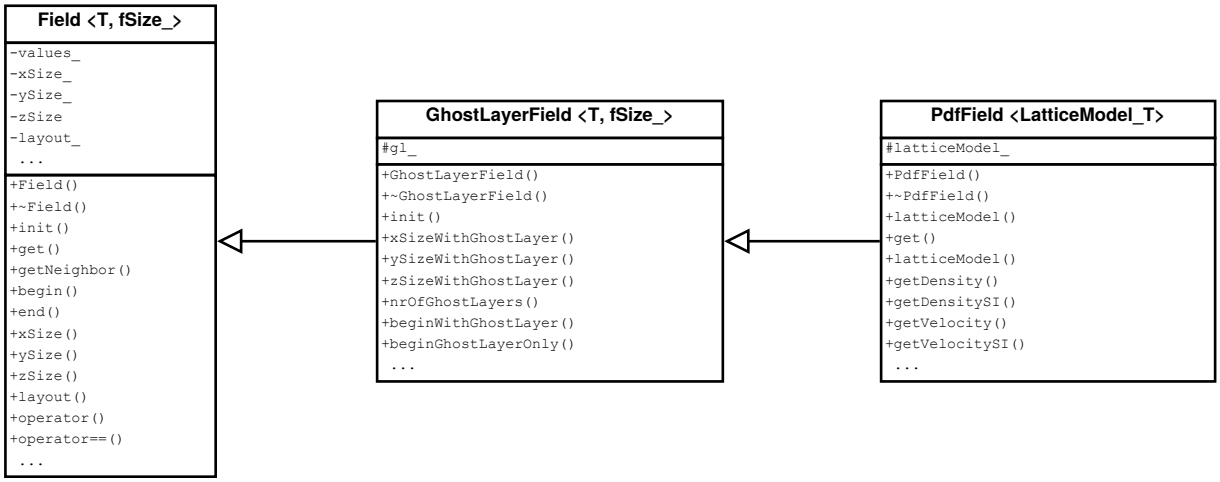
Listing 2.3: LatticeModel_T

```
1 typedef lbm::D3Q19< lbm::collision_model::SRT > LatticeModel_T;
```

onde lbm::D3Q19 é a classe definida para o modelo de *lattice* D3Q19. Note, que a classe D3Q19 também recebe um parâmetro de *template*, que é o tipo do operador de colisão, nesse caso definido por: lbm::collision_model::SRT, que representa a classe do operador de colisão SRT.

O diagrama de classes da Figura 2.25, mostra as classes descritas anteriormente com seus principais métodos e atributos, bem como a relação entre elas.

Figura 2.25: Diagrama de classe com o relacionamento entre as classes *Field*, *GhostLayerField* e *PdfField*.



Fonte: Autor.

Além das informações disponíveis no site do departamento de informática da universidade Erlangen-Nürnberg sobre o *framework* waLBerla, há também vários relatórios técnicos, artigos e a tese de doutorado de Christian Feichtinger (Feichtinger, 2012).

CAPÍTULO 3

REVISÃO DA LITERATURA

Este capítulo apresenta os trabalhos mais relevantes encontrados na literatura. Nestes trabalhos é possível encontrar implementações do LBM usando GPUs com suporte à CUDA. Essas implementações vão desde propostas eficientes para implementação do LBM usando memória compartilha em duas e três dimensões, uso de diferentes tipos de *stencil* em uma única GPU e implementações de domínios maiores em *clusters*. Além disso, também são encontrados trabalhos que apresentam diferentes condições de contorno e outras formas de realizar o passo de propagação das partículas do fluido.

3.1 Uso de memória compartilhada

Uma implementação do LBM em duas dimensões, usando o *stencil* D2Q9 (Seção 2.4.2), é proposta por Tölke (2008). Neste trabalho diferentes estratégias para implementação de *kernels*, de maneira eficiente em GPUs, são apresentadas. Primeiramente, a simulação foi dividida em três *kernels*: O primeiro *kernel*, definido como LBCollProp, é responsável por realizar os passos de colisão e propagação das partículas e por realizar as condições de contorno *bounce back* sobre as células do tipo NoSlip. Os passos de colisão e propagação são executados no mesmo *kernel*, a fim de reduzir a troca de dados entre eles, visto que essas operações geralmente são realizadas usando a memória global, que possui alta latência. Neste *kernel*, o domínio do problema é distribuído em uma *grid* com blocos em duas dimensões (x e y) e cada bloco possui apenas *threads* na dimensão x.

O segundo *kernel* é responsável pela sincronização das PDFs através das extremidades dos blocos de *threads*, devido à troca de dados entre os blocos. A maneira como os blocos são organizados é a mesma do *kernel* anterior, porém a *grid* está configurada de maneira diferente, com blocos apenas na direção y, onde cada *thread* executa toda uma linha do domínio de maneira sequencial, porém as demais *threads* são executadas de forma paralela. O último *kernel* é responsável pelas condições de contorno *inlet* e *outlet* e sua configuração é igual ao segundo *kernel*.

Para realizar o passo de propagação das partículas foi utilizado memória compartilhada. Foram realizados testes entre três operadores de colisão, o operador LBGK, também conhecido como SRT (Equação 2.13), o operador LBGKL, uma variação linear do operador LBGK e o operador MRT (Seção 2.4.5). O operador que obteve o melhor desempenho, foi o operador LBGKL, alcançando 670 MLUPS, os demais operadores alcançaram 568 e 527 MLUPS, para os operadores LBGK e MRT, respectivamente. Esses resultados foram obtidos utilizando a GPU 8800 ultra com precisão simples e usando 128

threads para cada bloco. Já para o processador Intel Core 2 Duo, a performance foi de apenas 23 MLUPS, ou seja, o resultado obtido pela GPU, foi mais de uma ordem de grandeza superior à performance da CPU.

Uma abordagem semelhante à Tölke (2008) é proposta por Kuznik et al. (2010). Neste trabalho todos os passos do algoritmo do LBM são realizados em um único *kernel*, chamado de collision_propagation. Um segundo *kernel* foi criado para troca dos dados entre as extremidades dos blocos de *threads* da GPU. No primeiro *kernel* o passo de propagação é realizado usando memória compartilhada, isto é, a propagação é realizada durante a cópia dos dados da memória global para a memória compartilhada.

No artigo os autores analisaram ainda vários tamanhos de *lattices* e diferentes quantidades de *threads* por bloco. O resultado que obteve o melhor desempenho foi com um *lattice* de tamanho de 1024×1024 e com 256 *threads* por bloco, atingindo 947 MLUPS usando apenas operações de ponto flutuante de precisão simples para o caso de testes *lid-driven cavity* em uma GPU GT200, usando o operador de colisão SRT. Para os testes com operações de ponto flutuante de precisão dupla, o resultado foi de 239 MLUPS para um *lattice* de dimensões 1024×1024 e com 256 *threads* por bloco. Os resultados obtidos mostram uma eficiente implementação, se compararmos os resultados obtidos por Tölke, além disso, os autores compararam os resultados obtidos utilizando precisão simples e dupla, e afirmam que precisão simples é suficiente para simulações do LBM. Esse resultado superior à Tölke, pode ter relação com o uso de uma arquitetura mais nova de GPU.

Tölke e Krafczyk (Tölke and Krafczyk, 2008) publicaram um artigo apresentando uma nova abordagem para o LBM, primeiro, utilizaram o tipo de *stencil* D3Q13, e segundo, utilizaram um *lattice* onde cada célula é representada por um dodecaedro, ou seja, cada face do dodecaedro possui uma PDF. Essa abordagem entretanto, não é encontrada com frequência na literatura.

O domínio do problema foi dividido de modo que a quantidade de células na direção x do *lattice*, definisse a quantidade de *threads* em cada bloco, dessa maneira, cada *thread* fica responsável por uma célula do *lattice*. Já a quantidade de células nas direções y e z, foram usadas para definir a quantidade de blocos. Na implementação proposta pelos autores, os passos de colisão e propagação das partículas são implementadas no mesmo *kernel*, porém durante o passo de propagação, foi utilizado memória compartilhada para armazenar os valores das PDFs. Ao término desta etapa, os valores são armazenados novamente na memória global, isso porque a memória compartilhada possui latência menor que a memória global da GPU. Esse mesmo *kernel*, também é responsável por tratar as condições de contorno *bounce back*.

Os testes foram feitos utilizando o operador de colisão MRT para o caso de testes *lid-driven cavity*. O melhor resultado atingiu 592 MLUPS utilizando operações de ponto flutuante de precisão simples com 64 *threads* por bloco para um tamanho do domínio de 128×128 células nas direções y e z. Esse valor representa 44% do pico de performance

máximo e 61% da largura de banda da memória (*throughput*) da GPU. Com relação a quantidade de memória, foram utilizados dois *lattices*, com isso o consumo de memória foi extremamente alto, próximo da capacidade total da GPU. Um detalhe interessante nesse trabalho, foi a utilização de uma camada adicional as direções y e z do domínio, definida como *ghost layer*, o que segundo os autores, permite uma propagação uniforme sem adicionar uma expressão condicional.

Outra proposta em três dimensões utilizando memória compartilhada é apresentada por Rinaldi et al. (2012). No esquema proposto pelos autores, os dados são copiados da memória global para a memória compartilhada da GPU a cada iteração, isso é feito durante o passo de propagação das partículas, porém a propagação é realizada antes da colisão das partículas, esse esquema é conhecido por *streaming-pull*. Em seguida todos os passos do algoritmo de LBM são calculados usando memória compartilhada e após a conclusão de todas as etapas, os valores das PDFs atualizadas, são escritos novamente na memória global. Como a memória compartilhada possui um tamanho pequeno comparado à memória global, isso acaba se tornando uma desvantagem, pois limita a quantidade de *threads* por SM.

Outra questão com relação à memória compartilhada é que cada vez que os dados são lidos dela, há a necessidade de sincronização entre as *threads* do mesmo bloco, para manter a integridade dos dados. Apesar disso, os autores afirmam que o uso de memória compartilhada é vantajoso porque reduz significativamente o número de acessos a memória global.

Com relação a implementação os autores utilizaram funções macros para representação dos dados e um único vetor, como estrutura de dados, para armazena os valores das PDFs nas quatro direções (x, y, z, e f). Além disso, foram utilizados dois *lattices* para armazenar os dados em memória, porém não fica claro qual o modelo de *layout* utilizado. Um único *kernel* foi implementado para realizar todos os passos do algoritmo de LBM e cada tipo de célula é verificado através de uma condição *if*. Nesse *kernel*, cada *thread* é responsável por uma célula da direção x do *lattice*. Os resultados apresentados mostram que o método utilizado atingiu até 400 MLUPS para a GPU GTX 260 com 50% da taxa de ocupação e com uma largura de banda de memória máxima de 61,6GB/s para o caso de testes *lid-driven cavity* utilizando apenas operações de ponto flutuante de precisão simples.

3.2 Análise do impacto do uso de registradores

Habich et al. (2011b) analisaram o alinhamento dos dados em memória e o uso de registradores em sua implementação do LBM usando o *stencil* D3Q19. Utilizando o *STREAM benchmark*[1], para medir a largura de banda de memória consumida por um algoritmo, verificaram qual a quantidade de blocos e *threads* que apresentavam os melhores resultados.

[1] https://www.cs.virginia.edu/stream/

Com isso, obtiveram até 72 GB/s de largura de banda de memória, para a GPU 8800 GTX, o que corresponde à 83% da performance máxima de memória com 8192 blocos com 64 *threads* cada. Com a GPU GTX 280 alcançaram um de resultado de 116 GB/s de largura de banda que corresponde à 73% da performance máxima de memória com 128 blocos com 64 *threads* cada.

Um aspecto importante em CUDA com relação a quantidade de registradores para cada *thread* é que quanto mais *threads* são usadas, menos registradores ficam disponíveis para cada *thread*, embora uma grande quantidade de *threads* venha a produzir melhores resultados com relação a largura de banda da memória. Na implementação proposta, os dados são organizados em *structure-of-arrays*, isto é, os valores das PDFs da mesma direção são armazenados em memória de forma consecutiva, e o passo de propagação foi feito da mesma maneira como no artigo de Tölke e Krafczyk, entretanto a performance não foi semelhante, visto que o modelo D3Q19 requer mais variáveis do que o modelo D3Q13, portanto a quantidade de registradores para cada *thread* é maior.

Segundo os autores, sua implementação consumia 70 registradores para cada *thread*, ou seja, uma grande quantidade. Para resolver esse problema, eles analisaram o código *assembly*, gerado pelo compilador NVCC, e observaram que para cada índice do vetor, um registrador era utilizado. Dessa maneira, a solução encontrada pelos autores, foi calcular os índices do vetor manualmente, reduzindo a quantidade de registradores para cada *thread* para 40.

O desempenho, com a redução da quantidade de registrados, dobrou e para a GPU 8800 GTX o resultado obtido foi de 200 MFLUPS (*Million Fluid Lattices Cells Updates per Second*)[2] e para a GPU GTX 280 foi de 400 MFLUPS, esses resultados foram obtidos com operações de ponto flutuante de precisão simples e com os dados alinhados em 128 Bytes, já com precisão dupla, o valor obtido foi de apenas 100 MFLUPS. Segundo eles, o uso de dados alinhados tem impacto maior sobre as arquiteturas anteriores (8800 GTX). O artigo também apresenta uma estimativa para uma implementação do LBM para um *cluster* híbrido (CPU/GPU), porém os resultados se mostram inferiores a implementação feita apenas para uma GPU, segundo os autores isso se deve a comunicação entre a CPU e a GPU via barramento PCIe e a rede de comunicação entre os nós do *cluster*.

3.3 Otimizações do acesso a memória global da GPU

Visto que a memória global das placas gráficas apresenta alta latência e o LBM possui uma grande quantidade de dados que precisam ser trocados entre a memória global e a GPU, o acesso padrão à esses dados em memória pode contribuir para um baixo desempenho da simulação. Uma forma de resolver esse problema é a utilização de memória compartilhada, entretanto no artigo de Obrecht et al. (2011), os autores apresentam uma implementação

[2]Apenas as células de fluido do domínio são consideradas no cálculo do desempenho.

em CUDA do LBM em três dimensões (D3Q19) usando apenas memória global. Nesse artigo os autores mostram uma forma eficiente de acesso aos dados em memória e além disso, há uma seção que trata dos princípios de otimização para algoritmos em CUDA.

Uma forma de obter maior desempenho, segundo os autores, é com relação à taxa de ocupação de cada SM, isto é, a razão entre a quantidade máxima de *threads* disponíveis em cada SM pela quantidade de *threads* em execução. Uma maneira de ajustar isso, é distribuir melhor o domínio do problema entre as *grids* de blocos de *threads*. Outra maneira de obter melhores resultados é utilizar um número de *threads* múltiplo do tamanho do *warp* (32). Com relação à alta latência da memória global, uma forma de otimizar a transferência dos dados é efetuar acesso coalescente e alinhado dos dados, para isso, as *threads* devem acessar os dados contíguos na memória. Apesar de ser uma otimização básica, o uso de variáveis temporárias, deve ser eliminado, pois essas variáveis são geralmente armazenadas em registradores o que leva a uma redução na taxa de ocupação. Por fim, expressões condicionais devem ser evitadas ao máximo, pois serializam a execução do código.

Na implementação proposta pelos autores, todo o passo de propagação das partículas foi realizado na memória global, e para evitar que os dados fossem sobrescritos utilizaram dois *lattices*, assim como no trabalho de Tölke e Krafczyk. Além disso, os dados foram organizados em *structure-of-arrays*. Com relação a propagação das partículas, os autores propuseram duas formas para realizar esse passo, a *split scheme* e a *reversed scheme*. Durante os testes eles observaram que a leitura dos dados desalinhados na memória global tem um custo menor do que a escrita desalinhada, por isso, no *split scheme* eles dividiram em duas partes: mudanças que induzem desalinhamento são feitas na leitura e as demais são feitas na escrita. A condição de contorno utilizada nesse esquema foi a *bounce back on-grid* e o operador de colisão foi o SRT. No *reversed scheme* a propagação é totalmente realizada na leitura, a condição de contorno utilizada é a *bounce back* em *mid-grid* e o operador de colisão foi o MRT.

Para o caso de testes *lid-driven cavity*, eles alcançaram até 86% do *throughput* máximo da memória global em uma placa gráfica GTX 295, mesmo com uma taxa de ocupação de 25%, o que pode ser considerada baixa, mas segundo eles, aumentar a taxa de ocupação dos SMs, não é particularmente crucial. Além disso, o operador de colisão que obteve melhor desempenho foi o MRT, alcançando 516 MLUPS para operações de ponto flutuante de precisão simples. Para os autores o principal fator limitante é a transferência de dados para a memória global, assim como a troca de *kernels*.

No artigo de Habich et al. (2011a), os autores demonstram estratégias de otimização em *kernels* que realização a simulação do LBM em GPUs e CPUs que suportam CUDA e OpenCL. O principal objetivo do trabalho consiste em melhorar o acesso a memória global da GPU, visto que sua implementação é limitada pela largura de banda da memória. Uma otimização feita pela autores é a mudança na ordem entre o passo de colisão e o passo de

propagação, ou seja, o passo de propagação é feito antes do passo de colisão (streaming-pull), com isso a colisão é calculada com base nos valores dos PDFs das células vizinhas e armazenado na célula atual. Entretanto, essa abordagem não é a mais comum encontrada na literatura, porém apresentou um desempenho melhor pois a leitura dos dados desalinhados tem impacto menor no desempenho do algoritmo com relação a escrita desalinhada. Além dessa estratégia, os autores também utilizaram otimizações aritméticas, redução de variáveis temporárias e uma técnica de otimização de index, que reduziu a quantidade de registradores para 32, porém está última otimização não fica claro como foi realizada.

Outra abordagem proposta pelos autores, é a utilização de dados alinhados em 128 Bytes na memória da GPU. Para garantir que os dados sejam acessados de maneira alinhada, ou seja, coalescente, é utilizada a técnica de *padding*. Para realizar os testes eles utilizaram o processador Intel Xeon Westmere X5650, as GPUs da NVIDIA 8800 GTX, Tesla C1060 e Tesla C2070, além da GPU AMD 9670. O caso de testes utilizado foi o *lid-driven cavity* em três dimensões. O resultado, obtido pelas GPUs da NVIDIA, mostra que a GPU Tesla C2070 obteve o melhor resultado para um domínio com 200^3 células e com o ECC desabilitado usando precisão simples, atingindo 650 MLUPS sem utilização de memória alinhada. Entretanto, testes realizados apenas com a GPU Tesla C2070, mostram que a utilização de memória alinhada teve melhores resultados com o ECC habilitado, tanto para precisão simples e dupla. O melhor resultado usando precisão dupla foi de 290 MLUPS. Nos testes não foi considerado o tempo gasto para copiar os dados da CPU para GPU pelo barramento PCIe. Com relação a largura de banda de memória, o melhor resultado obtido utilizou 83% da largura de memória usando precisão simples e com ECC desabilitado, já com ECC habilitado houve perda de performance de 30 a 40%.

3.4 Otimizações das condições de contorno

Outro aspecto relevante em implementações do LBM são as condições de contorno, Seção 2.4.3. No trabalho de Ho et al. (2009) os autores propõem a formulação de novas condições de contorno para bordas com velocidade e pressão para simulações do LBM em duas e três dimensões. Além disso, as novas condições de contorno também podem ser aplicadas a bordas estacionárias e não estacionárias e também para as células dos cantos do domínio da simulação.

Para determinar as funções de distribuição de partículas desconhecidas $f_i(\vec{x}, t)$ ao longo das bordas é utilizado a função de distribuição partículas local $f_i^*(\vec{x}, t)$ já conhecida, mais a adição de corretores, como pode ser visto na Equação 3.1, onde $\vec{Q}$ atua como uma força, para garantir a quantidade de movimento necessária.

$$f_i(\vec{x},t) = f_i^*(\vec{x},t) + \frac{\omega_i}{C}\vec{e}_i \cdot \vec{Q} \tag{3.1}$$

Por exemplo, para o *stencil* D2Q9 as funções de distribuição de partículas desconhecidas f_4, f_7 e f_8, podem ser definidas da seguinte maneira: $f_4 = f_4^* - \omega_4 Q_y, f_7 = f_7^* - \omega_7(Q_x + Q_y)$ e $f_8 = f_8^* + \omega_8(Q_x - Q_y)$. Esses corretores são obtidos diretamente da definição de densidade e da quantidade de movimento.

Para determinar qual a função de distribuição de partículas local deve ser usada, são propostas três maneiras: (a) utilizar a PDF local, porém na direção oposta, (b) usar a PDF local do passo anterior e (c) usar a função de distribuição de equilíbrio para a PDF local. Para avaliar a precisão das novas condições de contorno propostas, os autores utilizaram o fluxo Poiseuille, o fluxo Couette e o *lid-driven cavity* em duas dimensões, e apenas o fluxo Poiseuille em três dimensões. Segundo os autores, essas condições de contorno podem ser implementadas facilmente usando a mesma formulação das demais bordas. Os resultados obtidos alcançaram uma precisão de segunda ordem para os casos de testes e a escolha da função de distribuição local é arbitrária.

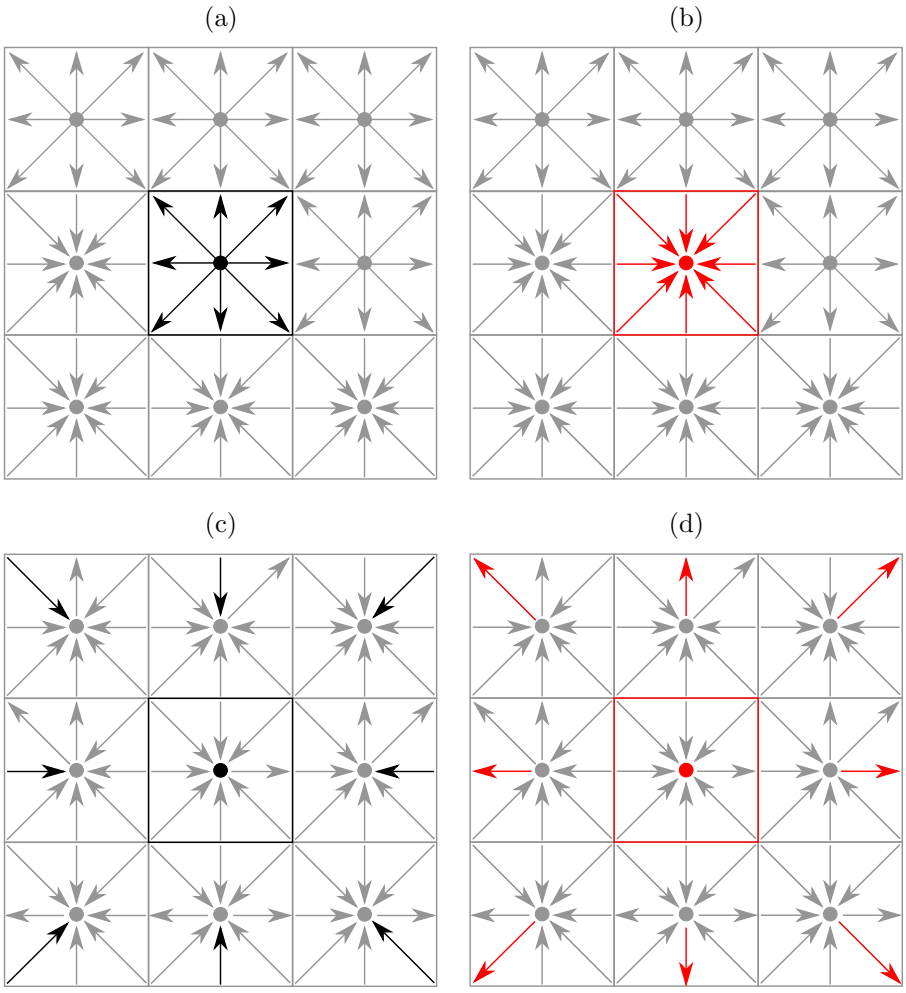
3.5 Comparação entre diferentes formas de realizar o passo de propagação

Além das condições de contorno, a maneira como a propagação das partículas é realizada também é uma questão a ser considerada. Bailey et al. (2009) analisaram e implementaram dois padrões de acesso a memória global da GPU, o primeiro deles, adotada pela maioria dos trabalhos que utilizam o LBM para simulação de fluidos, é aquele que utiliza dois *lattices* para armazenar os dados em memória e foi definida pelos autores como A-B. O segundo padrão, definido como A-A, utiliza apenas um *lattice* para armazenar os dados na memória global. Nesse padrão são utilizados dois *kernels* alternados, o primeiro *kernel* (A-A:1) é executado nas iterações pares e apenas o passo de colisão é realizado. Nesse *kernel*, os valores das PDFs da célula atual que ainda não foram atualizados estão em sua posição natural, Figura 3.1(a).

Para realizar a colisão, os PDFs da célula atual são lidos, em seguida aplicado o passo de colisão e por fim os valores pós-colisão (setas vermelhas) são armazenados na mesma célula, porém com sentido oposto Figura 3.1(b). Consequentemente os valores das PDFs do passo de tempo anterior $t-1$ são sobrescritos pelos valores do passo de tempo atual t. O segundo *kernel* (A-A:2) é executado nas iterações impares e os passos de propagação, colisão e propagação são realizados. O primeiro passo de propagação é realizado utilizando as PDFs das células vizinhas, que apontam para a célula atual Figura 3.1(c), após isso, o passo de colisão é realizado sobre esses valores e por fim o segundo passo de propagação é realizado armazenando os valores pós-colisão sobre as PDFs utilizadas anteriormente,

porém com sua orientação natural Figura 3.1(d). Com isso, o passo de tempo $t + 1$ sobrescreve os valores do passo de tempo anterior t. Porém no artigo, não fica muito claro como essa técnica é feita.

Figura 3.1: (a) Iteração par: leitura dos PDFs da célula atual. (b) Iteração par: armazena os valores pós-colisão na célula atual, mas com sentido oposto. (c) Iteração impar: leitura dos PDFs da células vizinhas. (d) Iteração impar: armazena os valores dos PDFs atualizados.



Fonte: Autor.

Nos testes realizados com o padrão A-B, o *kernel* implementado utilizou 32 registradores e atingiu uma taxa de ocupação de 33%, usando como caso de testes o fluxo de Poiseuille através de uma caixa e o operador de colisão SRT e seu desempenho foi de 300 MLUPS. Para o padrão A-A, usando o mesmo caso de testes e o mesmo operador, o *kernel* A-A:2 chegou a usar 64 registradores por *threads*, atingindo assim 17% de taxa de ocupação, com isso seu desempenho foi de 260 MLUPS. Ambas as implementações utilizaram operações de ponto flutuante de precisão simples. Essa quantidade alta de registradores, adiciona um significante quantidade de *overhead* a execução do *kernel*.

Em comparação a dissertação de mestrado de Habich, seu desempenho foi superior,

aumentado em 20% a velocidade máxima da simulação e reduzindo em 50% o uso de memória. Porém essas técnicas apresentaram um comportamento não ortogonal, isto é, quando menos memória era utilizada, ou seja, o padrão A-A, o ganho de performance era menor, e quando a velocidade máxima da simulação era utilizada, o padrão A-A não podia ser usado. Os testes foram feitos utilizando a GPU 8800 GTX. Um ponto relevante desse artigo, é que com a redução de 50% na quantidade de memória utilizada, é possível simular domínios maiores, visto que a quantidade de memória global das GPUs é menor, se compararmos com a quantidade de memória das máquinas atuais.

Wittmann et al. (2013) analisaram diferentes implementações para o passo de propagação das partículas implementadas em CPU e em GPU, além disso, compararam cada técnica implementada de acordo com o número de operações realizadas em memória para atualizar as células do *lattice*. Um modelo de performance, baseado na largura de banda da memória, é usado para obter uma estimativa da performance máxima obtida em diferentes máquinas.

As maneiras mais tradicionais que podem ser usadas para implementar a propagação das partículas em um fluido são apresentadas a seguir: o algoritmo mais simples consiste em implementar o operador de colisão e o passo de propagação de forma separada, utilizando dois *lattices* A e B, onde A armazena os valores das PDFs de cada célula do *lattice* e B armazena os valores das PDFs pós-colisão. Dessa maneira, para cada célula do *lattice*, é realizada a colisão das partículas e os novos valores são armazenados na mesma posição mas no *lattice* B. Durante o passo de propagação os valores armazenados em B são propagados para as PDFs vizinhas e armazenados em A para o próximo passo. Apesar da simplicidade do algoritmo, a quantidade de acessos a memória, devido à troca de dados entre os *lattices* A e B, é muito grande e utiliza o dobro de memória devido aos dois *lattices*.

A próxima implementação, também utiliza dois *lattices* A e B, porém a colisão e propagação são combinadas em um mesmo passo. E, após a colisão das partículas no *lattice* A os valores são propagados para os nós vizinhos e armazenados em B. Depois que todos as células são atualizadas, os ponteiros dos *lattices* são trocados e a iteração é reiniciada. Com a união dos passos de colisão e propagação os dados são lidos uma única vez em A e escritos uma vez em B, reduzindo o acesso à memória em 50%.

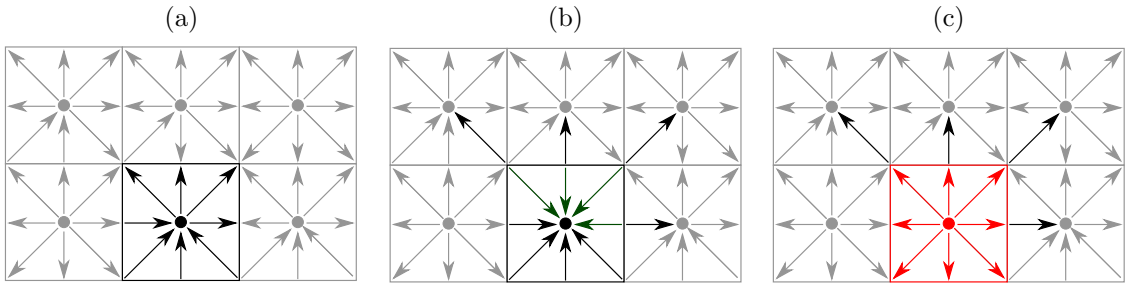
Outra implementação analisada pelos autores, é conhecida como *Compressed Grid*, e foi proposta por Pohl et al. (2003). Essa técnica usa apenas um *lattice* e cada direção do *lattice* é estendida por uma camada de células adicional. A dependência os dados é resolvida armazenando os valores das PDFs atualizadas, isto é, após realizar os passos de colisão e propagação, na célula localizada sobre a diagonal da célula atual. Entretanto, para que isso seja possível, essa técnica depende se a iteração da simulação é par ou impar. Para iterações pares, a iteração começa na célula localizada no canto superior direito, percorrendo todas as células até alcançar a célula localizada no canto inferior

esquerdo, ou seja, os valores das PDFs serão armazenados na camada adicional deslocada uma unidade para direita e uma unidade para cima. Quando a iteração é impar, a iteração tem início sobre a célula localizada no canto inferior esquerdo, percorrendo todas as células do *lattice* até atingir a célula do canto superior direito, e os valores das PDFs são armazenados na célula deslocada uma unidade para a esquerda e uma unidade para baixo.

Nesse mesmo trabalho, os autores analisaram ainda a implementação de Mattila et al. (2007), que propuseram outra maneira de realizar o passo de propagação, definido como *Swap Algorithm*. Essa nova abordagem, também um único *lattice* é necessário e a dependência dos dados é revolvida definindo uma ordem estrita de processamento das células do *lattice* e por uma contínua troca de metade dos PDFs de cada célula com as células vizinhas. A iteração sobre o *lattice* é feita em ordem lexicográfica e essa técnica depende do modelo de propagação adotado (*streaming-pull* ou *streaming-push*).

Para o modelo *streaming-pull*, as PDFs da célula atual devem ser trocadas primeiro, como mostra a Figura 3.2 (a), em seguida, as PDFs que apontam para as células vizinhas e que ainda não foram visitadas, são trocadas com as PDFs vizinhas que apontam para ela (setas verdes na Figura 3.2 (b)). Agora todas as PDFs necessárias no passo de colisão estão localizadas na célula local, ou seja, todas as PDFs estão localizadas em sentido contrário. Esses PDFs agora são lidos e o passo de colisão pode ser realizado. Dessa maneira, todos os valores das PDFs necessários para o passo de colisão, estão armazenados na célula atual, por fim, os valores pós-colisão são armazenados novamente na sua posição original na mesma célula, como pode ser visto na Figura 3.2 (c). A ordem da iteração é iniciada no canto inferior esquerdo e percorre todas as células do *lattice* até atingir o canto superior direito.

Figura 3.2: Exemplo do *Swap Algorithm* para o esquema de propagação *streaming-pull* usando o *stencil* D2Q9. (a) Célula atual, (b) Após a troca e (c) Colisão.



Fonte: Autor.

Uma abordagem semelhante à técnica *Swap Algorithm*, porém para GPUs, foi proposta por Bailey et al. (2009). Schönherr et al. (2011) apresentam uma abordagem semelhante ao padrão A-A proposto por Bailey et al. (2009), entretanto essa implementação requer apenas um *kernel* para realizar o passo de propagação do LBM. Porém, para essa abor-

dagem é necessário que o *layout structure-of-array* seja usado. Apesar de todos os testes realizados pelos autores nesse trabalho terem sido realizados apenas em CPUs, esse artigo permite, além de reunir as principais implementações do passo de propagação do LBM, elucidar como cada técnica é realizada.

3.6 Implementação do LBM em clusters

Para simulações do LBM em grande escala, ou seja, simulações com domínios maiores, é necessário utilizar uma grande quantidade de memória. Entretanto, a quantidade de memória global de uma única GPU não é suficiente, para isso, implementações em *clusters* são inevitáveis. Obrecht et al. (2013) descrevem uma eficiente implementação do LBM para *clusters* com GPUs NVIDIA. A solução apresentada, consiste na divisão do domínio da simulação em subdomínios. Além disso, é implementado um conjunto de rotinas de comunicação e um conjunto de rotinas de inicialização em MPI e um CUDA *kernel* para simulação do LBM usando o *stencil* D3Q19.

O *kernel*, além de realizar todos os passos do LBM, também é responsável pela troca dos dados entre os subdomínios e por sua ordenação. Dessa maneira, a troca dos dados pode ser realizada em todas as direção do subdomínio, permitindo o uso de particionamento do domínio em três dimensões, o que, segundo os autores, fornece mais flexibilidade para o balanceamento de carga e contribui para reduzir o volume de comunicação. O *kernel* foi implementado de maneira que cada *thread* receba uma célula do *lattice*, assim como a maioria dos trabalhos apresentados, e cada bloco possui apenas o tamanho de um *warp*. Cada *kernel* possui ainda um vetor auxiliar, alocado usando memória compartilhada, para realizar a troca de dados entre os subdomínios. Já as rotinas de inicialização e comunicação, são responsáveis por distribuir a carga de trabalho através do *cluster* e gerenciar a passagem dos dados entre os subdomínios, respectivamente.

Para os testes foi utilizado o caso de testes *lid-driven cavity* junto com o operador de colisão MRT. Os testes foram realizados sobre 8 nós do *cluster*, cada nó equipado com dois processadores hexa-core X5650 Intel Xeon, 36 GB de memória e três Tesla M2070. A conexão utilizada entre os nós do *cluster* é QDR infiniBand. A performance máxima alcançada foi de 8928 MLUPS utilizando operações de ponto flutuante de precisão simples em 24 GPUs Tesla M2070 com ECC habilitado, para um *lattice* com 768 células em cada direção. Nos testes de escalabilidade, os resultados se mostraram escaláveis e o principal fator limitante da performance foi a comunicação.

Outra implementação utilizando *cluster* com GPUs para simulação do LBM é apresentado por Xiong et al. (2012). Nesse artigo os autores utilizam um modelo de *stencil* proposto por Nan-Zhong et al. (2004) conhecido como iD2Q9, onde os passos de colisão e propagação das partículas são realizados da mesma maneira, entretanto, a função de distribuição de equilíbrio das partículas é diferente do modelo tradicional (Equação 2.17),

pois considera também o valor da pressão sobre as partículas e ao invés de usar o valor da densidade da célula atual, usa o valor da densidade do fluido no estado inicial.

Além disso, foi utilizado CUDA *stream* para execução de funções assíncronas, OpenMP e rotinas não bloqueantes em MPI. As funções CUDA *stream* permitem lançar *kernels* para a GPU de maneira assíncrona, ou seja, enquanto o *kernel* está executando, a CPU pode realizar outras tarefas até o *kernel* terminar sua execução. Para uma simulação do LBM em um *cluster*, isso implica que os passos de colisão e propagação podem ser realizados paralelamente à cópia os dados das extremidades do *lattice*, necessários no *lattice* do nó vizinho, para a CPU ou até mesmo transferir os dados para as CPUs vizinhas.

Nesse trabalho a sequência do algoritmo é feita da seguinte maneira: no início de cada iteração, o *kernel* Boundary_Condition é lançado de maneira assíncrona sobre o stream[0] para realizar o passo de colisão sobre as células das extremidades do *lattice* (não confundir com células de borda, onde as condições de contorno são realizadas). Nesse *kernel* apenas o passo de colisão das partículas é realizado, em seguida os valores das PDFs pós-colisão são armazenadas em um *buffer* e enviadas para a memória global da GPU. Após isso, é lançado o *kernel* Collision_Propagation sobre o stream[1], que realiza os passos de colisão e propagação sobre todas as células do *lattice*. A cópia dos dados entre a GPU e a CPU é realizada sobre o stream[0], dessa maneira garante que os dados serão copiados após o *kernel* Boundary_Condition ter terminado sua execução.

Embora essas operações sobre o stream[0] sejam serializadas, a execução do *kernel* Collision_Propagation, pode ser feita de maneira paralela. Um detalhe importante, como a cópia dos dados entre a CPU e a GPU foi feita usando a função *cudaMemcpyAsync* é necessário alocar o *buffer*, para troca dos dados, usando memória pinada. Depois da cópia dos dados entre a CPU e a GPU é realizado a troca de dados entres CPUs, isso é feito usando rotinas do MPI não bloqueantes. Finalmente, os dados recebidos do nó vizinho são copiados para a GPU e os valores das extremidades do *lattice* são atualizados. Caso os processos estejam no mesmo nó, os dados são transferidos usando memória compartilhada com OpenMP.

Os testes fora realizados no *cluster* Mole-8.5, o qual possui 362 nós com seis GPUs cada, entretanto, nesse trabalho apenas dois nós foram usados. Para validação dos testes foi utilizado o fluxo de Couette para um domínio tamanho 2048×2048 utilizando 12 GPUs NVIDIA Tesla C2050 e o resultado obtido foi de 1192 MLUPS para cada GPU usando operações de ponto flutuante de precisão simples. Com relação a escalabilidade, foram realizados testes variando a quantidade de GPUs de 12 até 1728 e os resultados mostram uma performance escalável.

Na Tabela 3.1 são apresentados os pontos relevantes sobre cada trabalho descrito nesse capítulo.

Tabela 3.1: Pontos relevantes de cada artigo.

Título	Autor(es)	Arquitetura	Operador de Colisão	Stencil	Relevância
LBM based flow simulation using GPU computing processor	Kuznik et al. (2010)	GT200	SRT	D2Q9	Uso de memória compartilha
Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA	Tölke (2008)	G80	MRT	D2Q9	Uso de memória compartilha
TeraFLOP computing on a desktop PC with GPUs for 3D CFD	Tölke and Krafczyk (2008)	G80	MRT	D3Q13	Uso de memória compartilha
A Lattice-Boltzmann solver for 3D fluid simulation on GPU	Rinaldi et al. (2012)	GT200	SRT	D3Q19	Uso de memória compartilha
Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA	Habich et al. (2011b)	G80/GT200	SRT	D3Q13	Análise do impacto do uso de registradores
A new approach to the lattice Boltzmann method for graphics processing units	Obrecht et al. (2011)	GT200	SRT	D3Q19	Otimizações do acesso a memória global da GPU
Performance engineering for the Lattice Boltzmann method on GPGPUs: Architectural requirements and performance results	Habich et al. (2011a)	G80/GT200/Fermi	SRT	D3Q19	Otimizações do acesso a memória global da GPU
Consistent boundary conditions for 2D and 3D lattice Boltzmann simulations	Ho et al. (2009)		SRT	D2Q9/D3Q13/D3Q19	Otimização das condições de contorno
Accelerating lattice Boltzmann fluid flow simulations using graphics processors	Bailey et al. (2009)	G80	MRT	D3Q19	Comparação entre diferentes formas para realizar o passo de propagação
Comparison of different propagation steps for lattice Boltzmann methods	Wittmann et al. (2013)		SRT		Comparação entre diferentes formas para realizar o passo de propagação
Scalable lattice Boltzmann solvers for CUDA GPU clusters	Obrecht et al. (2013)	GT200	MRT	D3Q19	Implementação do LBM em clusters
Efficient parallel implementation of the lattice Boltzmann method on large clusters of graphic processing units	Xiong et al. (2012)	Fermi	SRT	D2Q9	Implementação do LBM em clusters

CAPÍTULO 4

DESENVOLVIMENTO

Neste capítulo serão apresentados os principais conceitos utilizados para o desenvolvimento do módulo CUDA para o *framework* waLBerla, assim como as etapas de implementação do módulo a fim de atingir o objetivo principal desse trabalho. Como citado anteriormente, a atual versão do waLBerla não possui suporte à paralelização em GPUs. Tendo em vista isso, optou-se por desenvolver um módulo que fosse capaz de fornecer todo o suporte necessário para a realização de simulações do LBM em GPUs NVIDIA. Esse capítulo mostra ainda uma comparação entre a estrutura de um algoritmo em CPU, já utilizada pelo *framework* waLBerla, e em GPU, usando o novo módulo CUDA. Além disso, foi desenvolvido o caso de testes *lid-driven cavity* em GPU para validar o módulo proposto.

4.1 Módulo CUDA

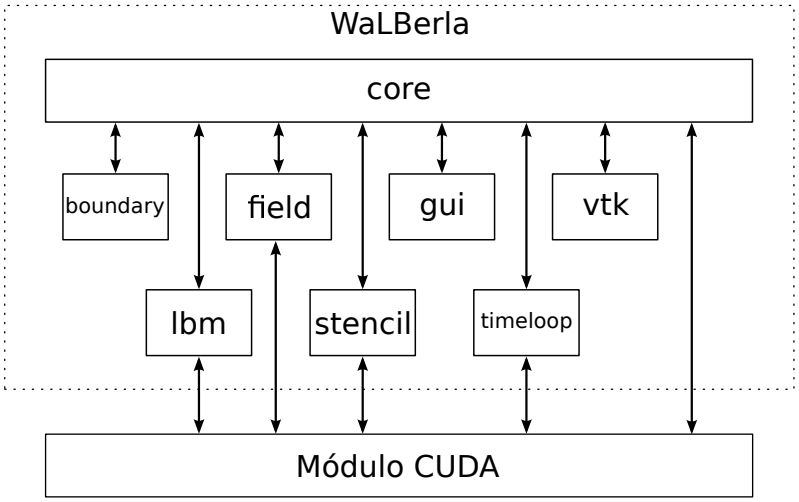
O principal objetivo do módulo CUDA é permitir que as diversas simulações que utilizam o LBM, já implementadas no *framework* waLBerla, sejam simuladas também em GPUs, visto que as placas gráficas da NVIDIA apresentam um alto poder computacional, além de serem adequadas para simulações de fluidos, como visto na maioria dos trabalhos relacionados. Para isso, foi necessário desenvolver um conjunto de classes, métodos e funções que permitissem a integração do novo módulo ao *framework* já existente. Para que essa integração fosse possível foi necessário desenvolver o novo módulo utilizando os conceitos e a estrutura já definidos pelo *framework* waLBerla.

A Figura 4.1 mostra os principais módulos do *framework* waLBerla (*boundary*, *field*, *lbm* e *timeloop*) assim como a relação deles com o *core* do *framework*. Além disso, a Figura 4.1 mostra a adição do novo módulo CUDA ao *framework* e sua relação com os demais módulos. Essa primeira versão do módulo foi desenvolvida para executar as simulações em apenas uma única GPU. A seguir serão apresentadas as principais classes que compõem o módulo.

4.1.1 GPUField

A principal classe do módulo CUDA é a classe *GPUField*. Essa classe representa a estrutura dos dados que são processados pela GPU. Foi desenvolvida com base na classe *GhostLayerField* e possui informações como tamanho do domínio nas dimensões x, y, z e f, além do número de *ghost layers*. Possui ainda, suporte aos *layouts structure-of-array*

Figura 4.1: Principais módulos do *framework* waLBerla e a adição do módulo CUDA.



Fonte: Autor.

e *array-of-structure*, semelhante a classe *Field*.

Os principais métodos da classe GPUField são: os métodos que retornam a quantidade de células em cada direção, os métodos que retornam a quantidade de células alocadas em cada direção e os métodos que manipulam a camada *ghost layer*.

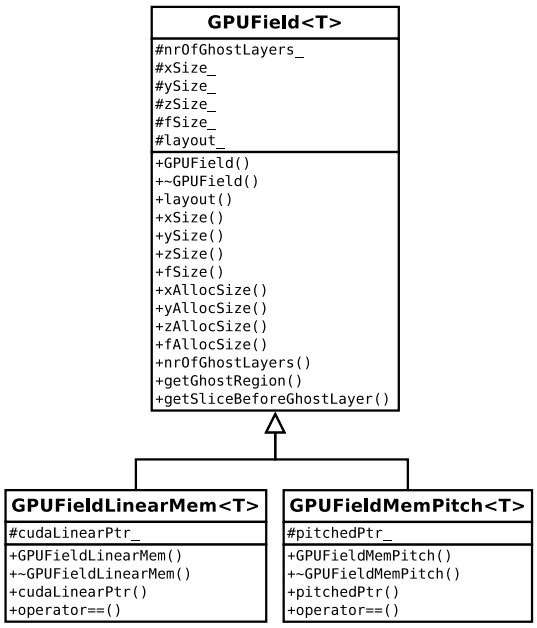
Assim como as classes *Field*, *GhostLayerField* e *PdfField*, a classe *GPUField* também é uma classe *template*. O único parâmetro de *template* que a classe *GPUField* recebe representa o tipo dos dados a serem alocados na memória da GPU. Os atributos e métodos da classe *GPUField* podem ser vistos no diagrama de classe da Figura 4.2.

Além da classe *GPUField*, foi desenvolvido as classes *GPUFieldLinearMem* e *GPUFieldMemPitch* que herdam os atributos e métodos da classe *GPUField*. As classes *GPUFieldLinearMem* e *GPUFieldMemPitch* são responsáveis por alocar memória linear e memória alinhada (Seção 2.6.3.3), respectivamente. A alocação de memória é feita no construtor de cada classe através das APIs *cudaMalloc()*, para a alocação de memória linear, e *cudaMalloc3D()*, para a alocação de memória alinhada. As classes filhas possuem um único atributo, que é o ponteiro da memória alocada de acordo com cada classe, e o operador de igualdade (*operator==*). Os atributos e os métodos das classes *GPUFieldLinearMem* e *GPUFieldMemPitch* podem ser visto na Figura 4.2, assim como o relacionamento entre elas com a classe pai.

4.1.2 Funções de cópia

Para copiar os dados entre a CPU e a GPU foram criadas as funções de cópia *fieldCpyLinearMem()* e *fieldCpyMemPitch()* com base nas funções já definidas pela API CUDA, ou seja, a função *cudaMemcpy()*, para copiar os dados para memória alocada de forma linear, e a função *cudaMemcpy3D()*, para copiar os dados para memória alinhada, como

Figura 4.2: Diagrama de classe com o relacionamento entre as classes *GPUField*, *GPUFieldLinearMem* e *GPUFieldMemPitch*.



Fonte: Autor.

pode ser visto no código 4.1.

Como visto no código, cada função recebe dois parâmetros, o primeiro deles, definido como *destination*, é o espaço de memória alocado na GPU para a estrutura de dados *GPUField*. O segundo parâmetro é o *source*, ou seja, os dados que foram inicializados na CPU utilizando a classe *PdfField*. Essa abordagem é bastante utilizada em implementações do LBM, como pode ser visto os trabalhos de Wittmann et al. (2013) e Habich et al. (2011a), por ser a maneira mais simples de resolver o problema da dependência dos dados, entre o passo atual do algoritmo, com relação aos dados do passo anterior, pois cada célula do *lattice* utiliza os valores das funções de distribuição de partículas das células vizinhas para atualizar seus valores. Essa abordagem, entretanto, consome o dobro de memória.

O sentido de cópia dos dados pode ser alterado quando houver à necessidade de copiar os dados da GPU para a CPU, para gerar um arquivo de saída. É importante ressaltar que os dados são copiados da CPU (*source*) a para GPU (*destination*), pois todo o processamento será realizado apenas na GPU.

Listing 4.1: Funções de cópia

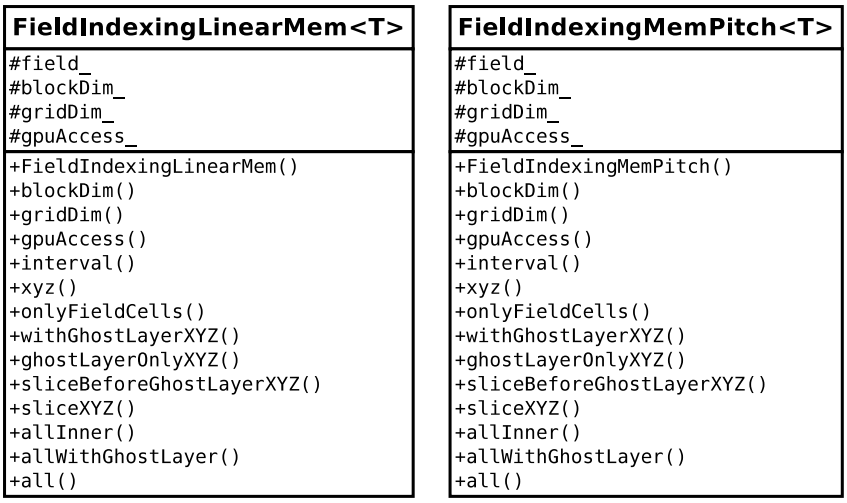
```
1 void fieldCpyLinearMem(const cuda::GPUFieldLinearMem<T> & dst, const field
      ::Field<T, fs> & src);
2
3 void fieldCpyMemPitch(const cuda::GPUFieldMemPitch<T> & dst, const field::
      Field<T, fs> & src);
```

4.1.3 Classes FieldIndexingLinearMem e FieldIndexingMemPitch

Para percorrer os dados em memória foram implementadas duas classes, a primeira, definida como *FieldIndexingLinearMem*, percorre os dados alocados através da classe *GPUFieldLinearMem* e a segunda classe foi definida como *FieldIndexingMemPitch* para percorrer os dados alocados pela classe *GPUFieldMemPitch*. Essas classes possuem métodos similares a iteradores, que definem qual o espaço do domínio será percorrido durante a simulação. Os métodos dessas classes são os mesmos, a única diferença é que os métodos da classe *GPUFieldMemPitch* devem considerar o alinhamento de memória.

Dentre os principais métodos podemos destacar o método *xyz()*, que permite percorrer todas as células do *lattice*, sem incluir a camada *ghost layer*, temos também o método *allInner()*, que além de percorrer todas as células do *lattice*, itera também sobre as funções de distribuição de partículas, ou seja, percorre as quatro dimensões (x, y, z e f). Podemos citar ainda, os métodos que percorrem apenas as células da camada *ghost layer* e também o método *all()*, que itera sobre todos os dados, incluindo a camada *ghost layer* e os PDFs. Todos os atributos e métodos das classes *FieldIndexingLinearMem* e *FieldIndexingMemPitch* podem ser vistos na Figura 4.3. Lembrando, que o modo como os dados serão percorridos, depende do *layout* escolhido na classe *GPUField*.

Figura 4.3: Diagrama de classe das classes *FieldIndexingLinearMem* e *FieldIndexingMemPitch*.



Fonte: Autor.

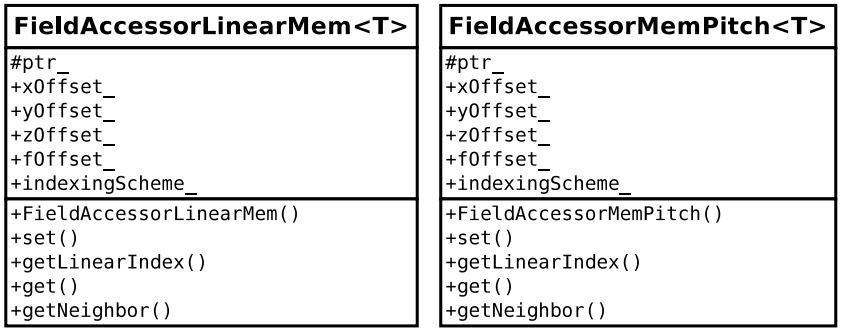
4.1.4 Classes FieldAccessorLinearMem e FieldAccessorMemPitch

Para acessar os dados na memória da GPU foram implementadas as classes *FieldAccessorLinearMem* e *FieldAccessorMemPitch*, que possuem métodos para manipulação dos dados. Os métodos dessas classes são semelhantes, porém há uma diferença com relação ao tipo de memória alocada, mesma situação que ocorre para as classes *FieldIndexin-*

gLinearMem e *FieldIndexingMemPitch*. Para definir novos valores para as funções de distribuição de partículas temos o método *get()*. Esse método permite atribuir novos valores para as PDFs, pois retorna o endereço de memória de cada variável.

Visto que o LBM depende dos valores de seus vizinhos, foi implementado o método *getNeighbor()* para auxiliar a atualização de cada PDF de acordo com a posição x, y, z de cada célula do *lattice*. O método *set()* foi implementado para definir a quantidade de blocos e *threads* em cada *kernel*. Todos os atributos e métodos das classes *FieldAccessorLinearMem* e *FieldAccessorMemPitch* podem ser vistos na Figura 4.4.

Figura 4.4: Diagrama de classe das classes *FieldAccessorLinearMem* e *FieldAccessorMemPitch*.



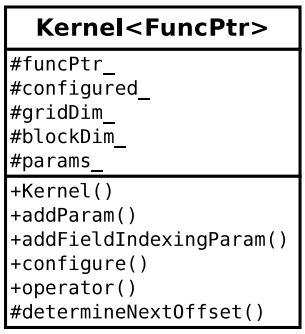
Fonte: Autor.

4.1.5 Kernel

Para que fosse possível integrar o novo módulo CUDA ao *framework* waLBerla foi necessário criar a classe *Kernel*. A classe *Kernel* foi implementada usando o conceito *Adapter* de padrões de projeto (Gamma et al., 1994). Essa classe é responsável por configurar o *kernel* da GPU, ou seja, definir a quantidade de blocos em cada *grid* e a quantidade de *threads* em cada bloco. Além disso, é responsável também por: receber os parâmetros que serão enviados ao *kernel* e lançar o *kernel* para a GPU.

Como o *framework* waLBerla é desenvolvido utilizando o padrão C++11 da linguagem de programação C++, a classe *Kernel* se tornou fundamental para o módulo CUDA, pois durante o período de desenvolvimento do módulo o compilador da NVIDIA (nvcc) ainda não suportava nenhuma funcionalidade do C++11, por isso todos os códigos com a extensão ".cu" e todos os arquivos de cabeçalho (*headers*), incluídos nesses arquivos, não podem conter nenhum elemento do C++11. Com isso, a forma tradicional de chamar *kernels*, como mostrado na Seção 2.6.1, não pode ser utilizada nesse módulo. Os atributos e métodos da classe *Kernel* podem ser vistos no diagrama de Figura 4.5.

Figura 4.5: Diagrama de classe da classe *Kernel*.



Fonte: Autor.

4.1.6 Bordas

Como todo o processamento do módulo CUDA é feito apenas na GPU, foi necessário implementar as condições de contorno para tratar as bordas do domínio da simulação para a GPU, pois as condições de contorno implementadas no *framework* waLBerla não poderiam ser utilizadas. Entretanto, as condições de contorno para a GPU foram desenvolvidas com base na implementação do *framework* e de acordo com as características de cada modelo, como visto na Seção 2.4.3.

Para o módulo CUDA foram implementadas apenas duas condições de contorno, uma para bordas estáticas, conhecidas também como *bounce back*, e outra para as bordas que possuem velocidade, como a borda superior do *lid-driven cavity*. Para as condições de contorno do tipo *bounce back* foi implementado a classe *GPUNoSlip*. Essa classe é responsável por tratar as bordas estáticas da simulação, ou seja, bordas que não possuem velocidade. Para isso, a classe possui os métodos *noSlip()* e o *operator()*, que apenas recebem os dados inicializados na CPU, através da classe *PdfField*, e os envia para o *kernel* na GPU, que realiza todo o processamento.

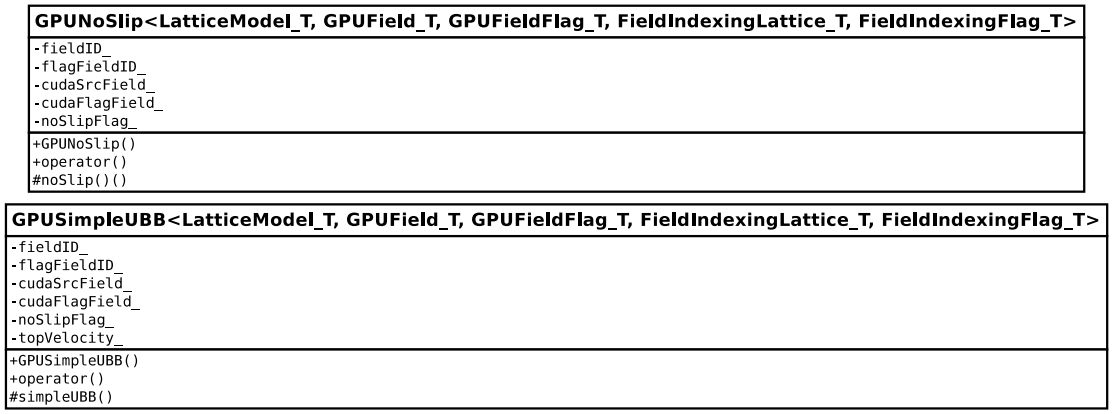
Para as bordas que possuem velocidade foi desenvolvido a classe *GPUSimpleUBB*. Essa classe é semelhante a classe *GPUNoSlip*, ou seja, possui apenas os métodos *simpleUBB()* e o *operator()*, porém além de receber os dados inicializados na CPU, também recebe como parâmetro o valor da velocidade da borda superior.

Além das classes apresentadas anteriormente, ambas as condições de contorno possuem versões de bordas otimizadas. Essas condições de contorno otimizadas percorrem apenas as bordas, através dos métodos das classes *FieldIndexingLinearMem* ou *FieldIndexingMemPitch*, sem iterar sobre todo o domínio. A versão otimizada da classe *GPUNoSlip* foi definida como *GPUNoSlipOptimized* e para cada borda do tipo *bounce back* um *kernel* é lançado. Essa foi a maneira encontrada para evitar que o domínio seja percorrido totalmente, assim apenas as bordas são percorridas.

Já para a classe *SimpleUBB* e sua versão otimizada (*SimpleUBBOptimized*), a única diferença entre elas é a porção do domínio percorrida. A Figura 4.6 mostra apenas o

diagrama de classes para as classes *GPUNoSlip* e *GPUSimpleUBB*, pois suas versões otimizadas possuem os mesmos atributos e métodos.

Figura 4.6: Diagrama de classes das classes *GPUNoSlip* e *GPUSimpleUBB*.



Fonte: Autor.

Nos diagramas de classe da Figura 4.6 é possível observar que as classes *GPUNoSlip* e *GPUSimpleUBB* possuem vários parâmetros de *template*. Esses parâmetros são necessários pois definem o modelo de *lattice* utilizado, a estrutura de dados, as *flag fields* e os métodos usados para iterar sobre o domínio.

4.1.7 GPU kernels

Para o módulo CUDA foram implementados três *kernels* na GPU, o primeiro para tratar as condições de contorno do tipo *bounce back*, o segundo para as bordas com velocidade e o terceiro para realizar os passos de colisão e propagação.

O *kernel noSlipKernel* é lançado pela classe *GPUNoSlip* através do método *noSlip()*. A responsabilidade desse *kernel* é atualizar as funções de distribuição de partículas de cada célula do *lattice* das bordas do tipo *bounce back*. Para isso, o *kernel* recebe os parâmetros da CPU e a cada iteração, verifica se a célula do *lattice* é do tipo *bounce back*, e caso seja verdadeiro, atualiza as PDFs, como pode ser visto no trecho de código 4.2. Porém, para atualizar todas essas direções em uma única iteração, só é possível graças a camada *ghost layer*. Essa abordagem é semelhante ao trabalho de Tölke e Krafczyk (Tölke and Krafczyk, 2008), entretanto nosso modelo adiciona a camada *ghost layer* em todas as direções do domínio.

É importante ressaltar que cada *thread* da GPU é responsável por uma célula do *lattice*, essa abordagem também é semelhante ao trabalho de Tölke e Krafczyk (Tölke and Krafczyk, 2008) e foi adotada para todos os *kernels* implementados.

Listing 4.2: Trecho do *kernel noSlipKernel* usado para atualizar as PDF das bordas estáticas.

```
1  if (flag.get() & noSlipMask) {
2
3    src.get(stencil::T)   = src.getNeighbor(0,0,+1,stencil::B);
4    src.get(stencil::TN)  = src.getNeighbor(0,+1,+1,stencil::BS);
5    src.get(stencil::TS)  = src.getNeighbor(0,-1,+1,stencil::BN);
6    src.get(stencil::TW)  = src.getNeighbor(-1,0,+1,stencil::BE);
7    src.get(stencil::TE)  = src.getNeighbor(+1,0,+1,stencil::BW);
8
9    src.get(stencil::E)   = src.getNeighbor(+1,0,0,stencil::W);
10   src.get(stencil::BE)  = src.getNeighbor(+1,0,-1,stencil::TW);
11   src.get(stencil::NE)  = src.getNeighbor(+1,+1,0,stencil::SW);
12   src.get(stencil::SE)  = src.getNeighbor(+1,-1,0,stencil::BW);
13
14   src.get(stencil::W)   = src.getNeighbor(-1,0,0,stencil::E);
15   src.get(stencil::BW)  = src.getNeighbor(-1,0,-1,stencil::TE);
16   src.get(stencil::NW)  = src.getNeighbor(-1,+1,0,stencil::SE);
17   src.get(stencil::SW)  = src.getNeighbor(-1,-1,0,stencil::NE);
18
19   src.get(stencil::S)   = src.getNeighbor(0,-1,0,stencil::N);
20   src.get(stencil::BS)  = src.getNeighbor(0,-1,-1,stencil::TN);
21
22   src.get(stencil::N)   = src.getNeighbor(0,+1,0,stencil::S);
23   src.get(stencil::BN)  = src.getNeighbor(0,+1,-1,stencil::TS);
24
25   src.get(stencil::B)   = src.getNeighbor(0,0,-1,stencil::T);
26   src.get(stencil::C)   = src.getNeighbor(0,0,0,stencil::C);
27
28 }
```

O segundo *kernel* foi definido como *simpleUBBKernel* e é responsável por atualizar as funções de distribuição de partículas da borda superior do domínio. Para isso, após recebe os parâmetros da CPU verifica, a cada iteração, se a célula do *lattice* é do tipo *simpleUBB* e caso seja verdadeiro atualiza as PDFs. Entretanto, as únicas funções de distribuição de partículas atualizadas pelo *kernel* são: B, BW, BE, BN e BS de acordo com as características da borda, como mostra o trecho de código 4.3 do *kernel simpleUBBKernel*.

Listing 4.3: Trecho do *kernel simpleUBBKernel* usado para atualizar as PDF da borda superior.

```
1  if (flag.get() & simpleUBBMask) {
2
3    // Borda superior central
4    src.get(stencil::B)  = src.getNeighbor(0, 0, -1, stencil::T);
5
6    src.get(stencil::BN) = src.getNeighbor(0, +1, -1, stencil::TS) + (6.0 *
       computeRho(&src, 0, +1, -1) * w[stencil::BN] * ((cx[stencil::BN] *
       topVelocity) + (cy[stencil::BN] * 0.0) + (cz[stencil::BN] * 0.0)));
7
8    src.get(stencil::BS) = src.getNeighbor(0,-1,-1,stencil::TN) + (6.0 *
       computeRho(&src,0,-1,-1) * w[stencil::BS] * ((cx[stencil::BS] *
       topVelocity) + (cy[stencil::BS] * 0.0) + (cz[stencil::BS] * 0.0)));
9
10   src.get(stencil::BW) = src.getNeighbor(-1,0,-1,stencil::TE) + (6.0 *
       computeRho(&src,-1,0,-1) * w[stencil::BW] * ((cx[stencil::BW] *
```

```
        topVelocity) + (cy[stencil::BW] * 0.0) + (cz[stencil::BW] * 0.0)));
11
12    src.get(stencil::BE) = src.getNeighbor(+1,0,-1,stencil::TW) + (6.0 *
          computeRho(&src,+1,0,-1) * w[stencil::BE] * ((cx[stencil::BE] *
          topVelocity) + (cy[stencil::BE] * 0.0) + (cz[stencil::BE] * 0.0)));
13  }
```

Para os passos de colisão e propagação foram implementados dois *kernels* para a GPU, o primeiro deles é uma simples implementação, como pode ser visto no código 4.4, enquanto que o segundo é uma versão otimizada. O *kernel* otimizado foi definido dessa maneira porque muitos dos cálculos semelhantes realizados durante as etapas de colisão, propagação e cálculo das grandezas macroscópicas são reaproveitados nessas etapas, com isso há uma redução significativa das operações realizadas. Como essas operações são apenas realizadas em células de fluido há uma verificação condicional, feita a cada iteração, para garantir que nenhuma dessas operações seja realizadas nas células das bordas. Essa verificação é feita através da *flag field* passada como parâmetro para o *kernel*.

Listing 4.4: Trecho do *kernel streamAndCollideDefaultKernel* usado os passos de colisão e propagação.

```
1  if (flag.get() & fluidMask) {
2
3    // Streaming Pull
4    for (int f = 0; f < 19; f++) {
5      dst.get(f) = src.getNeighbor(-cx[f], -cy[f], -cz[f], f);
6    }
7
8    // Calcula a velocidade e a densidade
9    for (int f = 0; f < 19; f++) {
10     rho += dst.get(f);
11     u   += dst.get(f) * cx[f];
12     v   += dst.get(f) * cy[f];
13     w   += dst.get(f) * cz[f];
14   }
15
16   u /= rho;
17   v /= rho;
18   w /= rho;
19
20   // Colisão
21   for (int f = 0; f < 19; f++) {
22     dst.get(f) = (1-omega) * dst.get(f) + omega * feq(rho, u, v, w, f);
23   }
24 }
```

Com relação ao modo de propagação das partículas ambos os *kernels* foram implementados utilizando a abordagem feita por Habich et al. (2011a) definida como *streaming-pull* (linha 5 do Algoritmo 4.4). O operador de colisão implementado para os *kernels* foi o operador SRT (Equação 2.13) (linha 24 do Algoritmo 4.4). Além dos passos de colisão e propagação, o *kernel* também é responsável por calcular os valores das grandezas ma-

croscópicas como a velocidade e a densidade do fluido (linhas 9 à 20 do Algoritmo 4.4). Diferente do algoritmo tradicional do LBM, onde a colisão e propagação das partículas são tratados separadamente, a abordagem adotada pelo módulo CUDA trata essas operações no mesmo *kernel*, pois reduz a quantidade de acessos a memória global da GPU, semelhante ao trabalho de Bailey et al. (2009).

Para percorrer os dados do domínio foi utilizado o método *xzy()* das classes *FieldIndexingLinearMem* ou *FieldIndexingMemPitch* que percorre apenas as células do *lattice*, inclusive as bordas.

Lembrando que em nossas implementações não foram utilizadas memória compartilhada e que o cálculo da função de distribuição de equilíbrio também foi implementado para a GPU.

4.2 Comparação entre a Estrutura do Algoritmo Lid-Driven Cavity em CPU e em GPU

O objetivo desta seção é mostrar que as alterações feitas no *framework* waLBerla para suportar GPU são acessível para o usuário, ou seja, não é necessário fazer nenhuma grande alteração na estrutura do código da simulação para que ela seja executada na GPU. Para isso, apresentamos uma comparação entre a estrutura de um algoritmo em CPU, já definida pelo *framework* waLBerla, e a estrutura de um algoritmo em GPU, usando o módulo CUDA, para o mesmo caso de testes *lid-driven cavity*. O Algoritmo 1 mostra os principais passos para implementação do *lid-driven cavity* em CPU.

Algorithm 1 Implementação do Lid-Driven Cavity em CPU

1: Aloca a estrutura do bloco;
2: Define os parâmetros da simulação (ômega e velocidade da borda superior);
3: Adiciona as estruturas de dados (Field, GhostLayerField ou PdfField) ao bloco;
4: Define as *flag field*;
5: Adiciona ao *loop* principal o *sweep* para tratar as bordas na CPU;
6: Adiciona ao *loop* principal o *sweep* para realizar os passos de colisão e propagação na CPU;
7: Executa o *loop* principal através do método run();

O Algoritmo 2 mostra que basta adicionar os *sweeps* que tratam as bordas e que realizam os passos de colisão e propagação na GPU ao *loop* principal para que a simulação seja realizada pelo módulo CUDA. A principal mudança com relação a implementação feita pelo *framework* waLBerla é que o *sweep* que trata as bordas foi dividido entre os dois tipos de bordas, ou seja, cada *sweep* trata um tipo de borda específica e também é necessário acrescentar ao *loop* principal o *sweep* que realiza a sincronização do *device*. Já o *sweep* que realiza os passos de colisão e propagação das partículas continua em um único *sweep*, porém esses passos são processados pela GPU. Os demais passos continuam semelhantes

ao *framework* waLBerla. Na linha 3 do Algoritmo 2 é possível observar que a estrutura de dados *GPUField* agora pode ser adicionada à estrutura do bloco do waLBerla.

Algorithm 2 Implementação do Lid-Driven Cavity em GPU

1: Aloca a estrutura do bloco;
2: Define os parâmetros da simulação (ômega e velocidade da borda superior);
3: Adiciona as estruturas de dados (Field, GhostLayerField, PdfField ou GPUField) ao bloco;
4: Define as *flag field*;
5: Adiciona ao *loop* principal o sweep para tratar a borda superior na GPU;
6: Adiciona ao *loop* principal o sweep para tratar as demais bordas na GPU;
7: Adiciona ao *loop* principal o sweep para realizar os passos de colisão e propagação na GPU;
8: Adiciona ao *loop* principal o sweep responsável por sincronizar o *device*;
9: Executa o *loop* principal através do método run();

4.3 Implementação do Lid-Driven Cavity em GPU

Para demonstrar o correto funcionamento de todas as classes, métodos e funções criadas no módulo CUDA foi implementado o caso de testes *lid-driven cavity*, pois é um método amplamente utilizado para validação das simulações do LBM, tanto para CPU, quanto para simulações em GPU, como pode ser visto nos trabalhos de Tölke e Krafczyk (Tölke and Krafczyk, 2008), Kuznik et al. (2010) e Obrecht et al. (2011). Essa implementação segue a aequência do Algoritmo 2 da seção anterior.

A implementação desse algoritmo consiste primeiro em alocar a estrutura do bloco na CPU, como mostra o código 4.5, depois definir os parâmetros da simulação como, ômega e a velocidade da borda superior, este caso os valores utilizados foram 1.4 e 0.1, respectivamente.

Listing 4.5: Cria apenas um bloco com N_x, N_y e N_z células em cada direção do domínio.

```
1 // Criação do bloco
2 auto blocks = blockforest::createUniformBlockGrid (
3   uint_t(1), uint_t(1), uint_t(1), // Quantidade de blocos
4   Nx,  Ny, Nz, // Quantidade de células em cada direção do domínio
5   real_c(1) // dx: tamanho de uma célula do lattice em coordenadas físicas
6 );
```

Em seguida, as estruturas de dados *PdfField* e *GPUField* são alocadas e adicionas ao bloco. Após isso, as PDFs são inicializadas utilizando os valores da função de distribuição de equilíbrio (Equação 2.17) no instante inicial, ou seja, quando os componentes da velocidade do fluido ainda são nulos. Com isso, temos apenas os valores dos pesos (w_i) de cada função de distribuição de partículas.

Após inicializar o domínio é utilizado a função *setFlagField()* para definir o tipo de cada célula do *lattice*, isto é, células de fluido, condição de contorno do tipo *bounce back* e borda superior, definida como *simpleUBB*, semelhante ao *framework* waLBerla. Essa configuração do domínio é feita de acordo com o caso de testes.

Depois disso, é definido o laço principal do algoritmo utilizando a classe *SweepTimeloop* do *framework* waLBerla, que permite definir a quantidade de iterações do algoritmo e também é responsável por adicionar os *sweeps* ao *loop* principal.

Após essas etapas, os dados são copiados para a GPU através das funções *fieldCpyMemPitch()* ou *fieldCpyLinearMem()*, dependendo da classe em que os dados foram alocados.

Para a implementação do *lid-driven cavity* foi necessário adicionar quatro *sweeps* ao *loop* principal do algoritmo. O primeiro *sweep*, apenas chama a classe *GPUNoSlipOptimized* e passa os parâmetros necessários à ela. O segundo *sweep* é semelhante ao primeiro, porém chama a classe *GPUSimpleUBBOptimized*. Já o terceiro *sweep* é o principal, pois é responsável pelos passos de colisão e propagação das partículas. O quarto e último *sweep*, apenas chama a função *cudaDeviceSynchronize()*, que bloqueia o *device* até que todas as tarefas sejam concluídas (NVIDIA, 2015b). Por fim, o laço principal é executado e os dados processados são copiados da GPU para a CPU.

Nessa implementação o operador de colisão utilizado foi o operador SRT, por ser simples de implementar, porém é o operador com melhor desempenho e também por ser bastante utilizado em simulações do LBM, como observado na maioria dos trabalhos relacionados.

Essa primeira versão do módulo CUDA não permite que o domínio da simulação seja dividido em blocos, como é feito no *framework* waLBerla para CPUs, portanto apenas um bloco é lançado para a mesma GPU. Para que seja possível dividir o domínio em vários blocos e atribuir cada bloco para uma GPU diferente é necessário implementar um esquema de comunicação entre as GPUs, semelhante ao mecanismo de comunicação entre CPUs desenvolvido no waLBerla.

CAPÍTULO 5

MATERIAIS E MÉTODOS

Neste capítulo serão apresentados os testes realizados, assim como o *hardware* utilizado em cada um dos testes. Foram efetuados testes para validar a implementação proposta, testes para medir o tempo total da simulação, ou seja, o tempo necessário para realizar todos os passos do algoritmo de LBM. Além disso, o consumo de memória em cada simulação também foi considerado. Da mesma forma, foram executados testes para comparar os resultados obtidos com os resultados encontrados na literatura, testes para medir o tempo de uma única iteração e testes para medir o tempo gasto para copiar os dados entre a CPU e a GPU. Foram também realizadas comparações entre o uso de memória alocada de maneira linear e alinhada, assim como comparações entre o uso de ECC habilitado e desabilitado. A seguir serão apresentadas as máquinas utilizadas nos testes.

5.1 Hardware

Nos testes foram utilizados três servidores, cada um com placas gráficas distintas. As placas gráficas utilizadas nos testes foram: NVIDIA Tesla C2075, NVIDIA Tesla K40m e NVIDIA GeForce GTX 750 Ti. As configurações de cada servidor são mostradas a seguir:

Servidor VRI:

- CPU: 2x Intel(R) Xeon(R) E5-2620 2.00GHz com 6 *cores* e 15MB de *cache* cada;
- 32GB de memória RAM;
- GPU: NVIDIA Tesla C2075;

Servidor Latrappe:

- CPU: 2x Intel(R) Xeon(R) E5-2680 v2 2.80GHz com 10 *cores* e 25MB de *cache* cada;
- 62GB de memória RAM;
- GPU 1: NVIDIA Tesla K40m;
- GPU 2: NVIDIA Tesla K10;

Servidor Orval:

- CPU: 2x Intel(R) Xeon(R) E5-4627 v2 3.30GHz com 8 *cores* e 16MB de *cache* cada;

- 252GB de memória RAM;
- GPU 1: NVIDIA Tesla C2075;
- GPU 2: NVIDIA GeForce GTX 750 Ti;

A versão da API CUDA utilizada foi a versão CUDA 7. Mais informações sobre as GPUs podem ser encontrá-das na Tabela 2.4.

Os equipamentos utilizados nos testes pertencem ao grupo de pesquisa VRI e ao grupo de pesquisa C3SL da Universidade Federal do Paraná (UFPR).

5.2 Teste de Validação

Para validar a implementação do módulo CUDA foi utilizado o caso de testes *lid-driven cavity*, como apresentado na Seção 2.5. Para isso, foram realizados testes com o objetivo de comparar os resultados obtidos pelo módulo CUDA com os resultados do *framework* waLBerla.

Este teste consiste em simular o *lid-driven cavity* utilizando o *stencil* D3Q19 para um tamanho de domínio de 128×128×128 células em cada direção. O valor definido para a velocidade da borda superior do caso de testes foi 0,1 e o sentido do deslocamento da borda ocorre na direção de x. Para esse teste foram executadas 10.000 iterações usando a GPU Tesla C2075 e o tempo total da simulação não foi considerado.

5.3 Testes de Desempenho

O intuito destes testes é medir o desempenho da implementação do caso de testes *lid-driven cavity* em MLUPS. *Million Lattices Cells Updates per Second* (MLUPS) é a unidade utilizada para medir o desempenho das simulações de LBM, ou seja, a quantidade de células de *lattice* atualizadas por segundo, geralmente medida em milhões de células (10^6). Para calcular a quantidade de MLUPS é utilizada a seguinte equação:

$$MLUPS = \frac{N_x * N_y * N_z * i}{t * 1000000} \tag{5.1}$$

onde, N_x, N_y e N_z representam a quantidade de células nas direções x, y e z do domínio, respectivamente, i é a quantidade de iterações e t é o tempo da total da simulação medido em segundos.

Para que fosse possível calcular o desempenho em MLUPS foi medido o tempo total da simulação (*wall-clock time*) usando o construtor e o método *end()* da classe *Timer* do *framework* waLBerla, como mostra o trecho de código 5.1.

Listing 5.1: Tempo total da simulação.

```
1  WcTimer timer; // Começa a medir o tempo.
2
3  timeloop.run(); // laço principal.
4
5  timer.end(); // Para de medir o tempo.
6  double time = timer.last(); // Exibe o último tempo medido.
7
8  std::cout << "Time WaLBerla Function: " << time << std::endl; // Imprime o
       valor do tempo medido em segundos.
9
10 return EXIT_SUCCESS;
```

Visto que os *sweeps* são adicionados ao *timeloop* o tempo é medido antes e após o *loop* principal. O tempo total da simulação não leva em conta o tempo necessário para enviar os dados inicializados na CPU para a GPU. Essa medição foi feita separadamente, como será apresentado nas próximas seções.

Para medir o consumo de memória foi implementado a função *GPUMemoryUsed()*, a qual utiliza a função *cudaMemGetInfo()* da API do CUDA para retornar a quantidade de memória consumida pela simulação. O valor de retorno representa a quantidade de memória consumida em megabytes, como pode ser visto no código 5.2.

Listing 5.2: Função utilizada para medir o consumo de memória usado durante a simulação.

```
1  void GPUMemoryUsed() {
2
3      size_t available; // Quantidade de memória disponível da GPU.
4
5      size_t total; // Quantidade total de memória global da GPU.
6
7      WALBERLA_CUDA_CHECK( cudaMemGetInfo( &available, &total ) );
8
9      size_t used = (total - available)/1048576; // Calcula a quantidade de
           memória e retorna o valor em megabytes.
10
11     std::cout << "GPU Memory Used: " << used << "MB" << std::endl; //
           Imprime a quantidade de memória utilizada.
12 }
```

Em todos esses testes foram usados dois tamanhos de domínio, o primeiro com $N_x \times 128 \times 128$ células e o segundo com $N_x \times 256 \times 256$ células. O objetivo é variar a quantidade de *threads* em cada bloco, pois a quantidade de células (N_x) da direção x do domínio, define a quantidade de *threads* em cada bloco da GPU, essa abordagem é semelhante ao trabalho de Habich et al. (2011a). Dessa maneira, N_x varia de 32 até 512 células, ou seja, múltiplos do tamanho do *warp*, assim como apresentado na seção de otimizações do trabalho de Obrecht et al. (2011). Enquanto que a quantidade de blocos na direção x da *grid* (*blockIdx.x*) e a quantidade de blocos na direção y da *grid* (*blockIdx.y*) permanecem constantes.

Além de diferentes tamanhos de domínio também foi levado em consideração o uso

de ECC nas GPUs. Esses testes foram feitos com o intuito de poder comparar as GPUs, visto que a GPU GTX 750 Ti não possui suporte a ECC, com isso os testes descritos anteriormente foram feitos com ECC habilitado e desabilitado, assim como no trabalho de Habich et al. (2011a).

Outra comparação feita em nossos testes foi com relação ao desempenho das simulações utilizando memória linear e memória alinhada. O alinhamento de memória utilizado em nossa classe *GPUFieldMemPitch* é feito pela própria API CUDA através da estrutura *cudaPitchedPtr* e usando a função *cudaMalloc3D()* para alocação de memória.

Outra questão analisada nesses testes foi a taxa de ocupação dos *kernels* executados nas GPUs. A taxa de ocupação é definida pela razão entre a quantidade de *threads* que serão executadas em um *kernel* pela quantidade total de *threads* em cada SM, de acordo com a Equação 5.2. A taxa de ocupação depende da *compute capability*, da quantidade de *threads* por bloco, da quantidade de registradores utilizados em cada *thread*, da configuração de memória compartilhada e da quantidade de memória compartilhada (Wilt, 2013).

$$\frac{warp\ por\ SM}{Max.\ warp\ por\ SM} \tag{5.2}$$

A taxa de ocupação pode ter impacto na performance do algoritmo, mas segundo Wilt (2013), não é um fator determinante para que isso realmente aconteça, pois geralmente é melhor usar mais registradores por *threads*.

Durante os testes de desempenho cada simulação executou 200 iterações e cada teste foi repetido 10 vezes, e apenas o melhor resultado, isto é, o menor tempo encontrado foi utilizado para medir o desempenho em MLUPS.

5.4 Teste de Comparação com a Literatura

Como nossa implementação do *lid-driven cavity* utilizou alguns conceitos semelhantes apresentados no trabalho de Habich et al. (2011a), optou-se por realizar um teste com o objetivo de comparar os resultados alcançado pelos autores com o resultado obtido pelo módulo CUDA. Além disso, a abordagem proposta pelos autores era utilizada na segunda versão do *framework* waLBerla. Nesse teste foi apenas utilizado memória linear, pois o alinhamento encontrado no trabalho de Habich é diferente da nossa implementação, e o tamanho do domínio foi de 200×200×200 células. Entretanto, esse teste foi realizado apenas com operações de ponto flutuante de precisão dupla.

Os testes também foram realizados usando ECC habilitado e desabilitado. A GPU utilizada pelo autores foi a Tesla C2070, porém como observado na Seção 5.1, nossos servidores não possuem essa mesma GPU. Apesar disso, utilizamos a GPU Tesla C2075 que é semelhante à GPU Tesla C2070, ou seja, possui a mesma quantidade de CUDA

cores, mesma quantidade de memória e o *clock* dos processadores é o mesmo. A única diferença encontrada foi o consumo de energia.

5.5 Teste de Comparação entre o Tempo de Cópia e o Tempo de uma Iteração

O objetivo deste teste é comparar o tempo total de uma única iteração, ou seja, o tempo para realizar apenas um passo do algoritmo de LBM, e o tempo necessário para copiar os dados da CPU para a GPU. Nesse teste o tamanho do domínio variou de acordo com a quantidade de *threads* em cada bloco, assim como no Teste de Desempenho (Seção 5.3), para alterar a quantidade de dados transferidos. Esse teste foi realizado usando apenas a GPU Tesla K40m com memória linear e com ECC habilitado. O trecho de código 5.3 mostra a maneira como o tempo de cópia foi medido.

Listing 5.3: Medida do tempo para copiar os dados entre a CPU e a GPU.

```
1 WcTimer timer; // Começa a medir o tempo.
2
3 cuda::fieldCpyLinearMem(*cudaSrcField.gpuField, *field); // copia os dados
      do field (src) para o gpuField (dst).
4 cuda::fieldCpyLinearMem(*cudaFlagField, *flagField); // copia a flag field
      (src) para a flag field (dst).
5
6 timer.end(); // Para de medir o tempo.
7 double time = timer.last(); // Exibe o último tempo medido.
8
9 std::cout << "Time WaLBerla Function: " << time << std::endl; // Imprime o
      valor do tempo medido em segundos.
```

Onde a função *fieldCpyLinearMem()* é a função utilizada para copiar os dados alocados usando memória linear.

Em todos os testes realizados cada arquitetura diferente testada era utilizado o parâmetro *-arch=sm_xx* para compilar o módulo CUDA para uma arquitetura específica, garantindo que o executável gerado fosse para a arquitetura que estava sendo testada, pois como visto na Seção 5.1, os servidores onde os testes estavam sendo realizados possuíam GPUs de arquiteturas diferentes. No parâmetro de compilação *-arch=sm_xx*, *xx* representa a *compute capability* (cc) da GPU, para a GPU Tesla C2075 o valor de cc é 20, para a GPU Tesla K40m o valor é 35 e para a GPU GTX 750 Ti o valor é 50, como mostra a Tabela 2.4.

Diferente do que foi comparado no trabalho de Habich et al. (2011a) nossa implementação utilizou apenas operações de ponto flutuante com precisão dupla, sem levar em conta operações de ponto flutuante com precisão simples. Além disso, optamos por não realizar testes de comparação entre CPU e GPU, como muitos autores fazem, pelo fato de serem arquiteturas diferentes onde uma única GPU possui centenas de *cores* enquanto que

uma única CPU possui uma quantidade inferior, na ordem de dezenas de processadores. A Tabela 5.1 apresenta a relação dos testes realizados.

Tabela 5.1: Relação dos testes realizados

Tipo de Teste	Memória	ECC	Domínio	GPU(s)
Teste de Validação	Linear	ON	128×128×128	Tesla C2075
Teste de Desempenho	Linear e Alinhada	OFF	N_x×128×128 e N_x×256×256	Tesla C2075, Tesla K40m e GTX 750 Ti
	Linear e Alinhada	ON	N_x×128×128 e N_x×256×256	Tesla C2075 e Tesla K40m
Teste de Comparação com a Literatura	Linear	ON e OFF	200×200×200	Tesla C2075
Teste de Comparação entre o tempo de cópia e o tempo de uma iteração	Linear	ON	N_x×256×256	Tesla K40m

CAPÍTULO 6

RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados obtidos pelo módulo CUDA em simulações do LBM para o caso de testes *lid-driven cavity* em três dimensões utilizando o *stencil* D3Q19. A seguir serão apresentados e discutidos os resultados obtidos em nossos testes, assim como a validação do *lid-driven cavity*.

6.1 Teste de Validação

Esta seção apresenta uma comparação entre os resultados encontrados durante uma simulação do LBM em CPU e em GPU para validação do caso de testes *lid-driven cavity*. Essa comparação é realizada, pois o caso de testes *lid-driven cavity* desenvolvido pelo *framework* waLBerla foi devidamente validado. Para isso, foram comparados os valores das velocidades (V_x e V_y) obtidos pela GPU com os resultados da CPU, como pode ser visto nos gráficos da Figura 6.1. As velocidades obtidas pela simulação na GPU (linha azul) estão de acordo com os valores encontrados pelo *framework* waLBerla. Esse teste é bastante utilizando para validação do *lid-driven cavity*, como pode ser visto nos trabalhos de Ghia et al. (1982) e Rinaldi et al. (2012).

Para obter os valores das velocidades foi realizado um corte sobre o eixo y do domínio. Esse corte foi feito sobre metade do eixo y e para obter os valores das velocidades foi utilizado o *software* Paraview. Dessa maneira, o domínio foi reduzido a apenas uma fatia em duas dimensões e os valores das velocidades V_x e V_y de cada célula do *lattice* foram usados para validação. Além disso, na Figura 6.2 é possível observar a formação do vórtice principal ao redor do eixo y. Com esses resultados é possível afirmar que a simulação do *lid-driven cavity* está correta.

6.2 Testes de Desempenho

O gráfico da Figura 6.3 mostra os resultados obtidos durante os testes com um tamanho de domínio de $N_x \times 128 \times 128$ células em cada direção. Nesse teste foi usada memória linear e o ECC das GPUs (Tesla C2075 e Tesla K40m) estava desabilitado, para que pudéssemos comparar os resultados com a GPU GTX 750 Ti. Como descrito anteriormente, a quantidade de células (N_x) na direção x do domínio define a quantidade de *threads* em cada bloco da GPU, com isso variamos N_x de 32 até 512 células com o objetivo de verificar qual a quantidade de *threads* apresenta o melhor desempenho.

Figura 6.1: Comparação entre os resultados obtidos em o *framework* waLBerla e o módulo CUDA. (a) Representa as velocidades na direção y (V_y) sobre a direção x do domínio e (b) representa as velocidades na direção x (V_x) sobre a direção y do domínio.

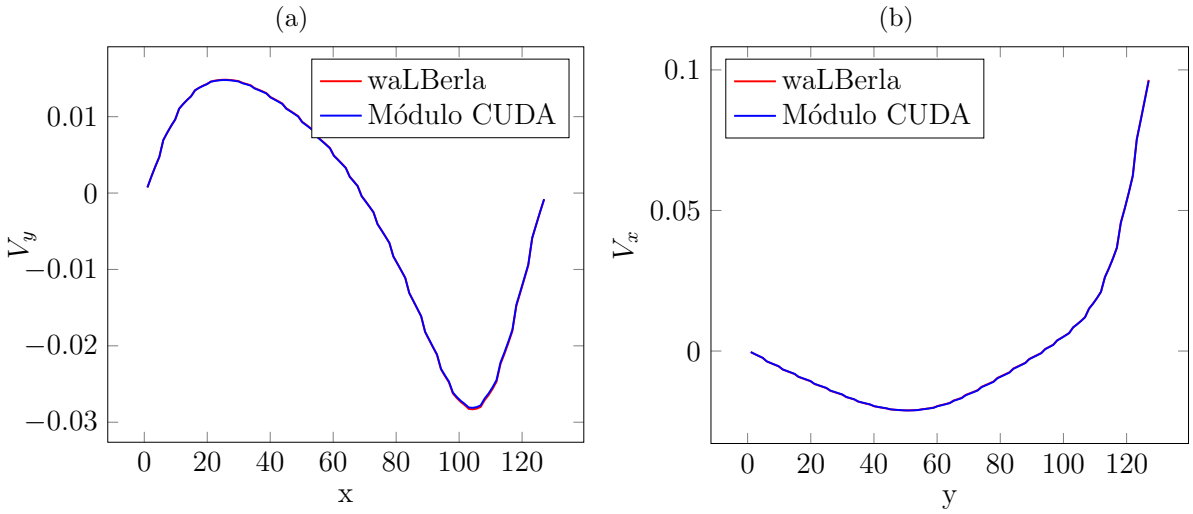
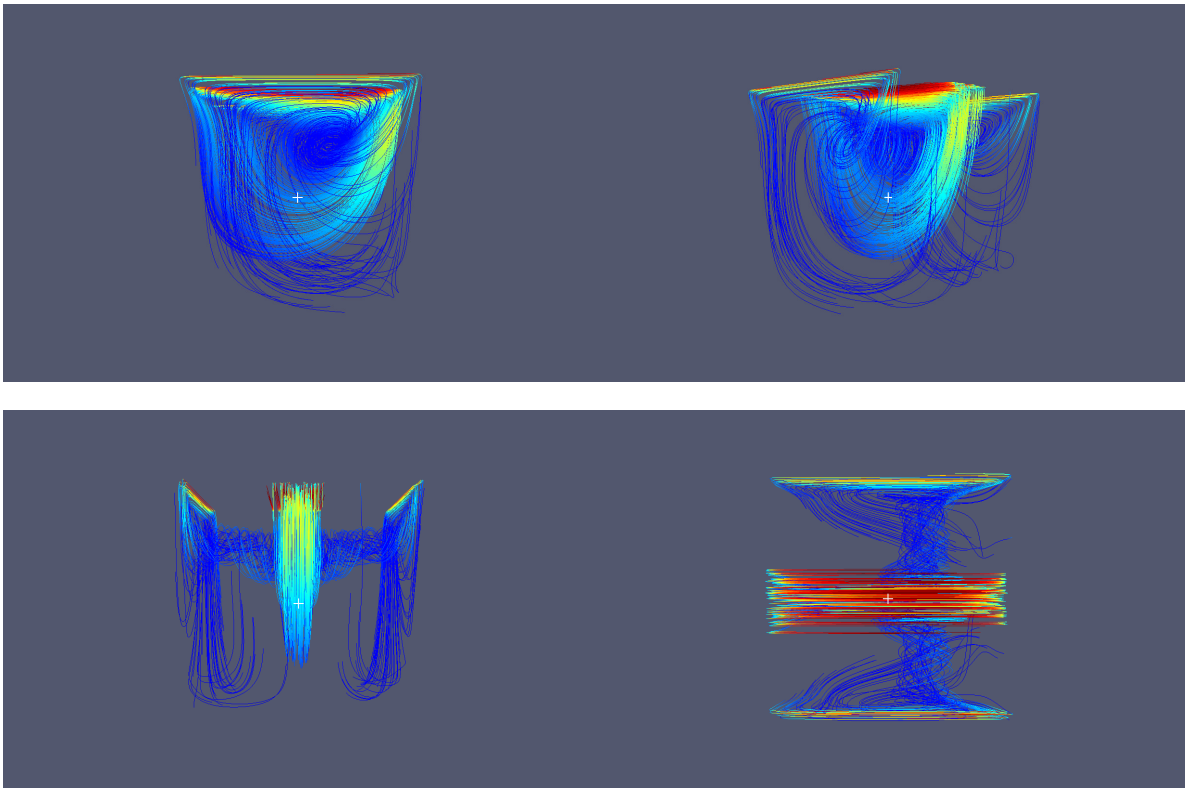


Figura 6.2: Linhas de corrente vórtice principal ao redor do eixo y para o número de Reynolds Re = 100.

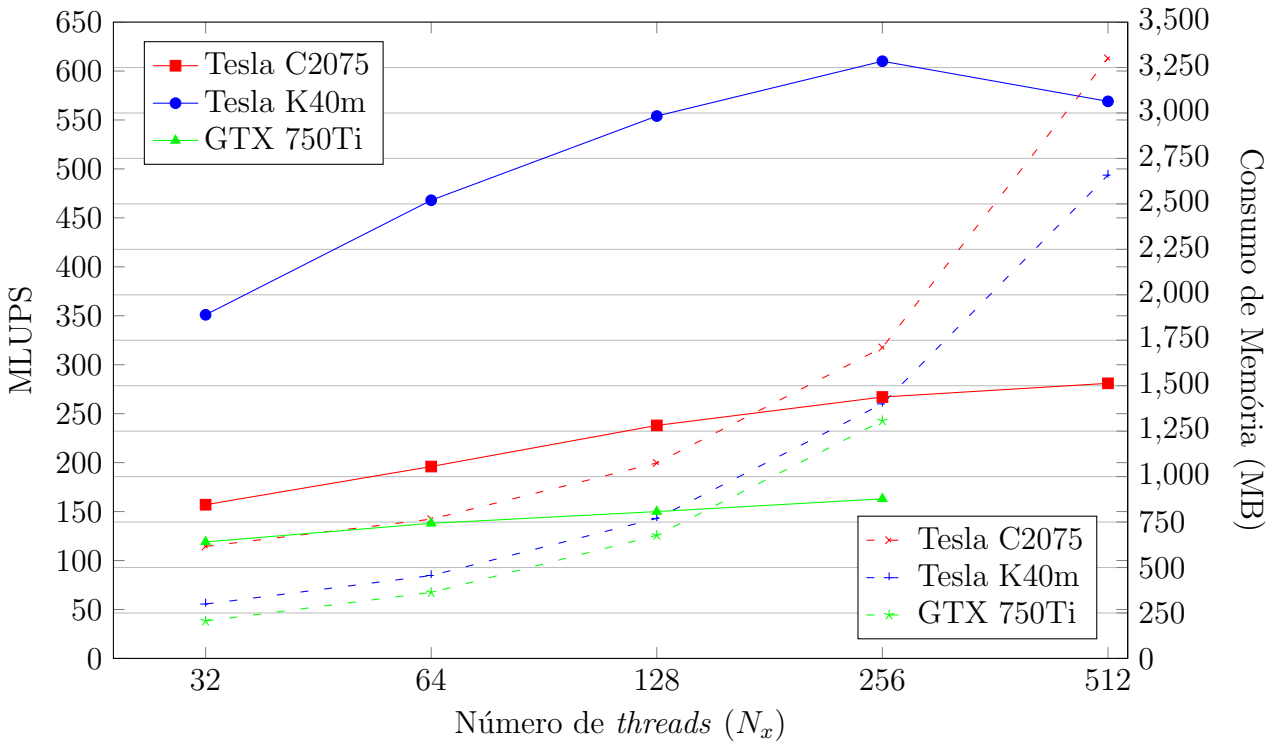


Os resultados mostram que a GPU Tesla K40m teve o melhor desempenho com relação às demais GPUs, alcançando 610 MLUPS com 256 *threads* por bloco. Diferente da GPU Tesla C2075, que obteve o melhor resultado com 512 *threads*, a GPU Tesla K40m não

obteve o melhor resultado para a mesma quantidade de *threads*, pois houve uma redução na taxa de ocupação, como pode ser visto o gráfico 6.11.

Além do desempenho, o gráfico 6.3 também mostra o consumo de memória (linhas tracejadas) para cada simulação. O consumo de memória da GPU Tesla C2075 foi superior as demais GPUs, pois além de ser usada para executar as simulações também estava sendo usada como monitor.

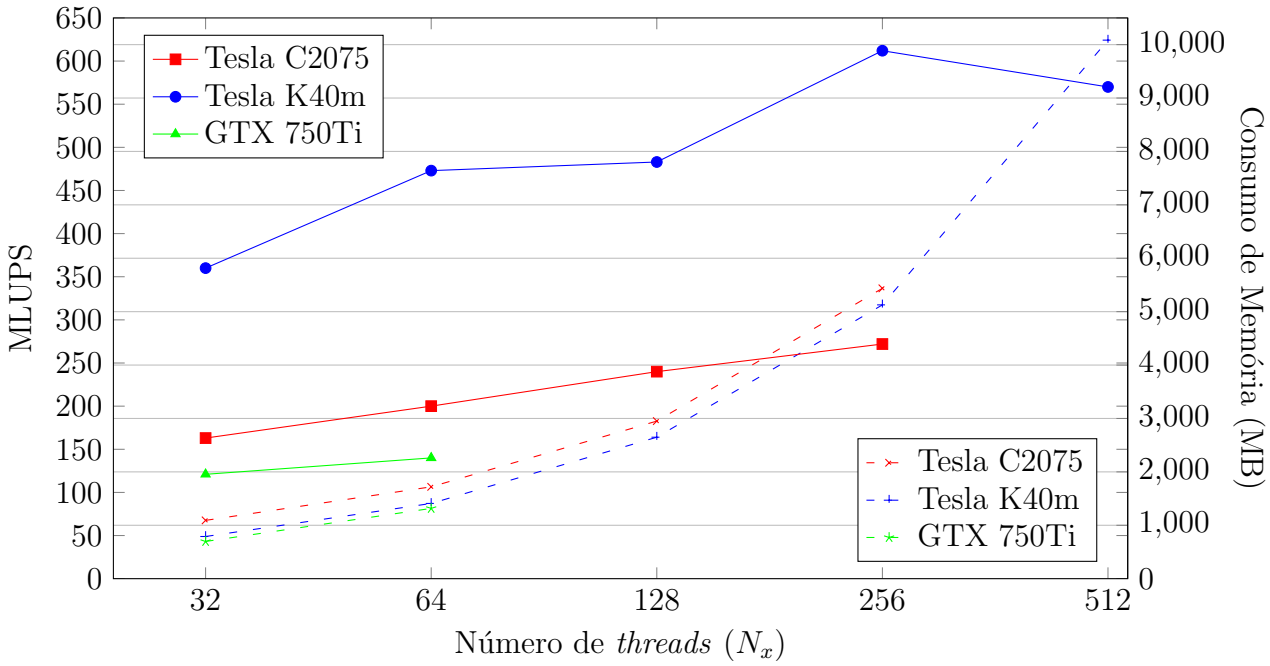
Figura 6.3: Desempenho das simulações usando **memória linear** e com **ECC desabilitado** para um domínio de tamanho $N_x \times 128 \times 128$.



O segundo gráfico (Figura 6.4) mostra os resultados obtidos utilizando um tamanho de domínio maior, com $N_x \times 256 \times 256$ células em cada direção. A única diferença desse teste com relação ao primeiro é o tamanho do domínio. Com um domínio maior podemos observar que a quantidade de memória consumida pelas simulações quase atingiu a quantidade máxima de memória global de cada GPU. Com isso, tivemos uma redução da quantidade de *threads* em cada bloco. Para a GPU GTX 750 Ti, apenas 32 e 64 *threads* por bloco foram usadas. Já a GPU Tesla C2075 utilizou no máximo 256 *threads*, enquanto que a GPU Tesla K40m usou 512 *threads* por bloco, ou seja, não houve alteração da quantidade de *threads* utilizadas, devido a sua grande quantidade de memória global, alcançando o melhor resultado com 256 *threads*.

O terceiro gráfico (Figura 6.5) mostra o desempenho das simulações utilizando memória alinhada. Nesse primeiro teste com memória alinhada o tamanho do domínio foi de $N_x \times 128 \times 128$ células em cada direção. Podemos observar que o desempenho de memória

Figura 6.4: Desempenho das simulações usando **memória linear** e com **ECC desabilitado** para um domínio de tamanho $N_x \times 256 \times 256$.



alinhada é inferior ao desempenho usando memória linear. Essa redução no desempenho ocorre para todas as GPUs analisadas. Como o alinhamento dos dados ocorre na direção x, a quantidade máxima de *threads* utilizada nesse teste foi de apenas 128 *threads*, ou seja, houve uma redução significativa no desempenho com relação a memória linear. Nesse teste a GPU que obteve o melhor desempenho foi a GPU Tesla K40m, com 357 MLUPS com apenas 64 *threads* por bloco.

Os resultados do segundo teste com memória alinhada podem ser vistos no gráfico da Figura 6.6. Eles foram realizados com um domínio de tamanho $N_x \times 256 \times 256$ e o melhor resultado foi de 401 MLUPS para a GPU Tesla K40m com 64 *threads*. Como o domínio utilizado nesse teste é o maior não foi possível executar os testes na GPU GTX 750 Ti, pois a quantidade de memória necessária era superior à sua quantidade máxima de memória global.

Os demais testes foram feitos com ECC das GPUs (Tesla C2075 e Tesla K40m) habilitado, eliminando a GPU GTX 750 Ti que não possui ECC. Com o uso do ECC observamos outra queda no desempenho, pois o ECC verifica e corrige possíveis erros na memória durante a execução de qualquer algoritmo na GPU.

No teste com memória linear e com o tamanho de domínio de $N_x \times 128 \times 128$ células o melhor resultado atingido foi de 488 MLUPS para a GPU Tesla K40m usando 512 *threads*, ou seja, houve um redução de 14,3% se compararmos com ECC desabilitado e o mesmo tamanho de domínio. Porém, como o melhor resultado obtido pela GPU Tesla K40m foi

Figura 6.5: Desempenho das simulações usando **memória alinhada** e com **ECC desabilitado** para um domínio de tamanho $N_x \times 128 \times 128$.

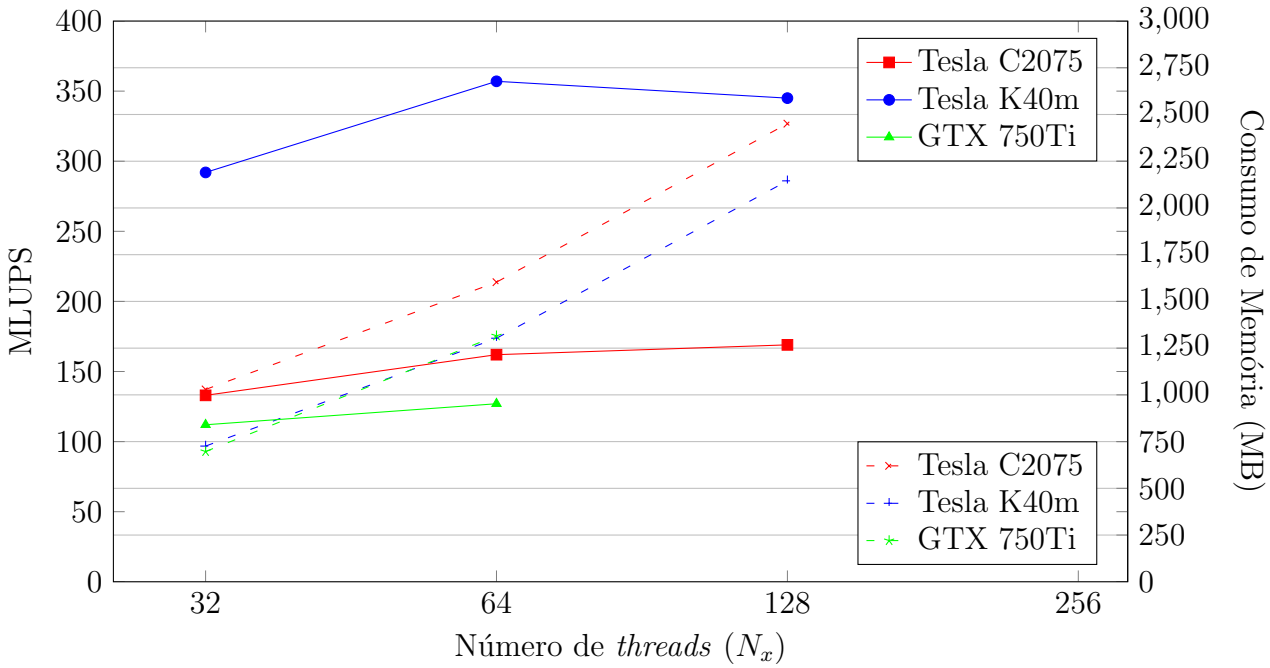
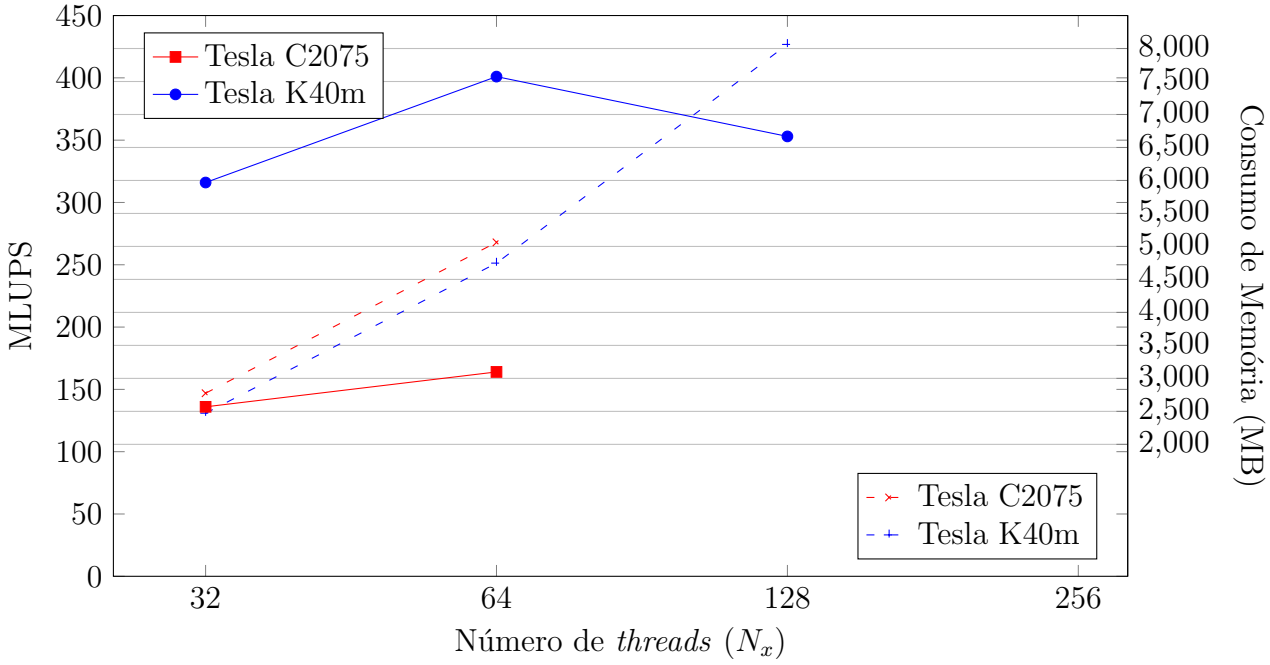
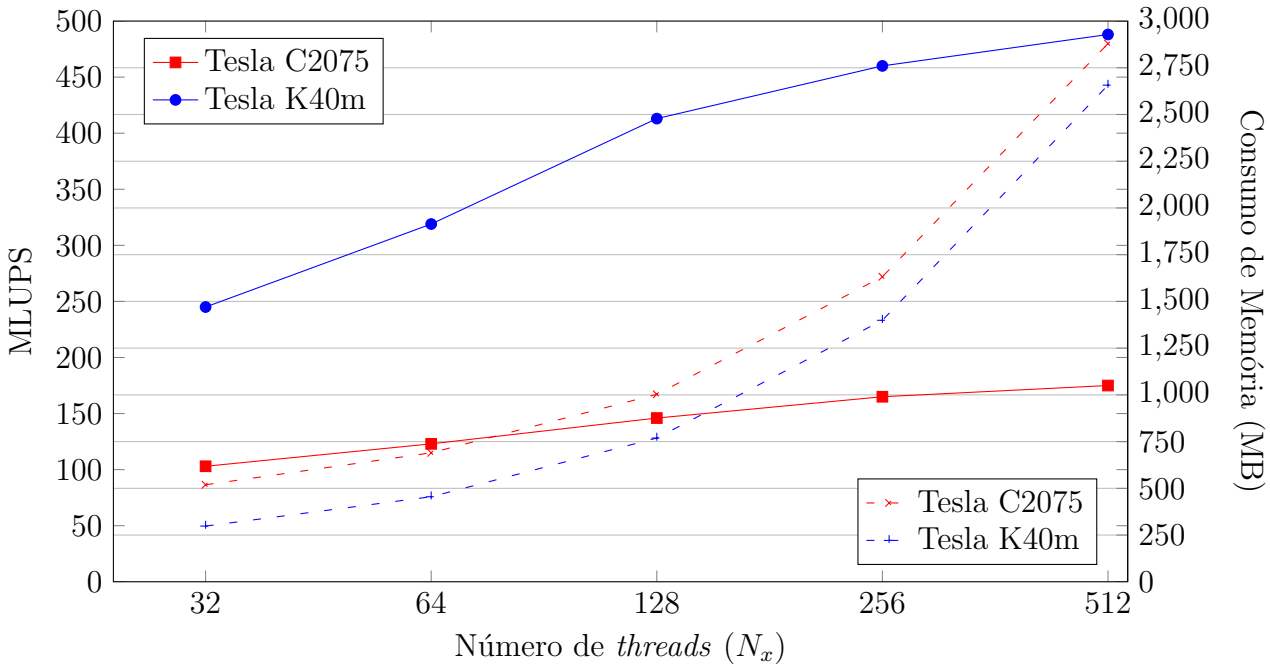


Figura 6.6: Desempenho das simulações usando **memória alinhada** e com **ECC desabilitado** para um domínio de tamanho $N_x \times 256 \times 256$.



com 256 *threads* e com ECC desabilitado, essa diferença no desempenho é ainda maior, ou seja, uma redução de 20% no desempenho da simulação. Esses resultados podem ser observados no gráfico da Figura 6.7.

Figura 6.7: Desempenho das simulações usando **memória linear** e com **ECC habilitado** para um domínio de tamanho $N_x \times 128 \times 128$.



O segundo teste com ECC habilitado e memória linear foi realizado com um tamanho de domínio de $N_x \times 256 \times 256$ células e o melhor resultado foi de 489 MLUPS para a GPU Tesla K40m com 512 *threads*. Com ECC habilitado e memória linear o desempenho para ambos os tamanhos de domínio foi semelhante. O consumo de memória, mais uma vez, foi próximo da capacidade máxima das GPUs. Lembrando que o ECC consome cerca de 12,5% da quantidade máxima da memória global da GPU. O gráfico da Figura 6.8 mostra os resultados desse teste.

O gráfico da Figura 6.9 mostra o desempenho da simulação usando memória alinhada e ECC habilitado. Para o tamanho de domínio $N_x \times 128 \times 128$ o melhor desempenho foi da GPU Tesla K40m atingindo 266 MLUPS com apenas 64 *threads*. Porém para a GPU Tesla C2075 seu melhor foi com 128 *threads*.

Para o teste com memória alinhada, ECC habilitado e com o tamanho de domínio de $N_x \times 256 \times 256$ células a quantidade de *threads* em cada bloco da GPU foi de no máximo 128 para a GPU Tesla K40m, porém seu melhor desempenho foi com apenas 64 *threads* atingindo 271 MLUPS, como pode ser visto no gráfico da Figura 6.10. Já a GPU Tesla C2075 obteve melhor resultado com 64 *threads*, pois foi a quantidade máxima de *threads* por bloco para esse teste.

Apesar dos bons resultados obtidos a taxa de ocupação das GPUs foi baixa, como pode ser visto no gráfico da Figura 6.11. Isso porque, a quantidade de *threads* utilizada em cada *streaming multiprocessor* durante nossas simulações é inferior à quantidade de

Figura 6.8: Desempenho das simulações usando **memória linear** e com **ECC habilitado** para um domínio de tamanho $N_x \times 256 \times 256$.

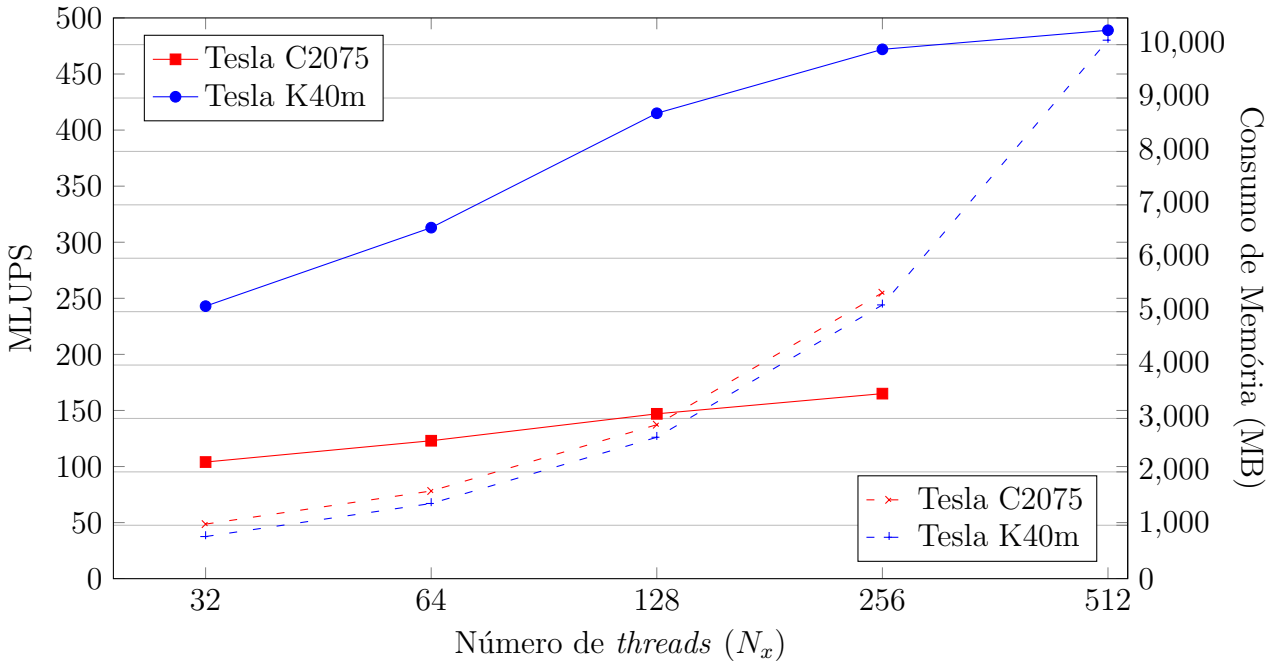
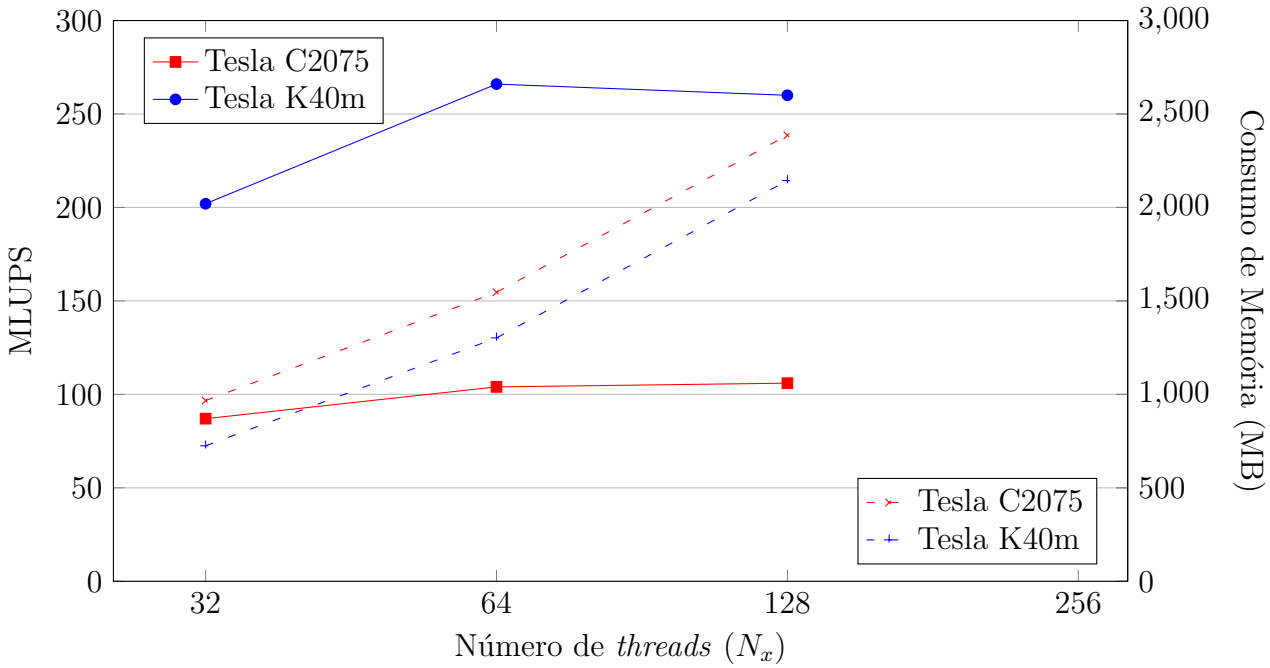
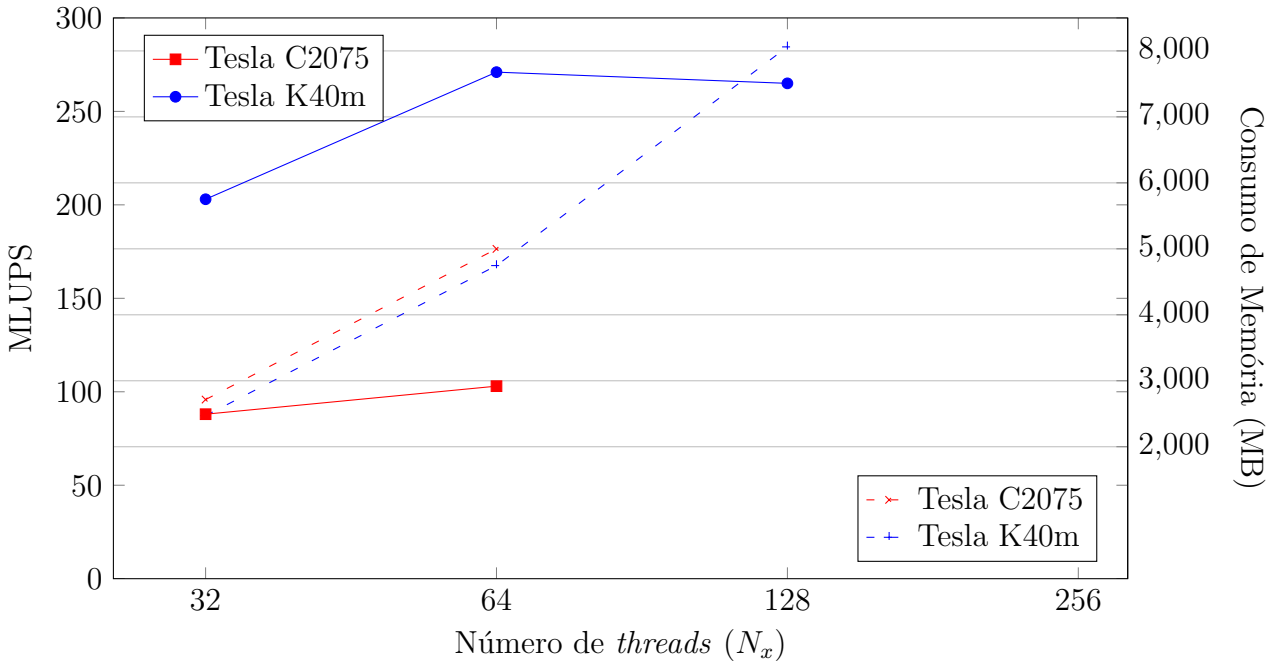


Figura 6.9: Desempenho das simulações usando **memória alinhada** e com **ECC habilitado** para um domínio de tamanho $N_x \times 128 \times 128$.



threads disponível. Outra questão que influencia na taxa de ocupação é a quantidade de registradores utilizados para cada *thread*. Em uma análise feita sobre o *kernel* streamAndCollideD3Q19KernelOptimized pudemos observar que a quantidade de registradores usa-

Figura 6.10: Desempenho das simulações usando **memória alinhada** e com **ECC habilitado** para um domínio de tamanho $N_x \times 256 \times 256$.



dos pelas *threads*, durante a execução desse *kernel*, foi alta, com isso a taxa de ocupação apresentou valores baixos. Para a GPU Tesla K40m com 128 *threads* a quantidade de registradores utilizados em cada *thread* foi de 74. Para a mesma quantidade de *threads* as GPUs Tesla C2075 e GTX 750 Ti utilizaram 63 e 78 registradores, respectivamente.

No entanto, a taxa de ocupação não sofreu nenhuma influencia do tamanho do domínio ou do tipo de memória alocada. Como pode ser visto no gráfico 6.11 a GPU que obteve a maior taxa de ocupação foi a GPU Tesla K40m. Para a GPU GTX 750 Ti não foi possível executar o teste usando 512 *threads*.

Os gráficos da Figura 6.12 apresentam apenas uma comparação entre o desempenho obtido usando memória linear e o desempenho usando memória alinhada. O gráfico da Figura (a) mostra os resultados obtidos para um tamanho de domínio de $64 \times 128 \times 128$ células, tamanho máximo para realizar uma simulação na GPU GTX 750 Ti, com ECC desabilitado e memória alinhada. Já com ECC habilitado o tamanho do domínio utilizado foi de $128 \times 128 \times 128$ células, ou seja, o maior tamanho de domínio suportado pela GPU Tesla C2075 usando memória alinhada, como pode ser visto na Figura (b). Esses tamanhos de domínios não representam os melhores resultados obtidos, como visto nos gráficos anteriores, porém foram utilizados com o objetivo de comparar as GPUs.

Figura 6.11: Taxa de Ocupação com relação a quantidade de *threads* para o *kernel* streamAndCollideD3Q19KernelOptimized.

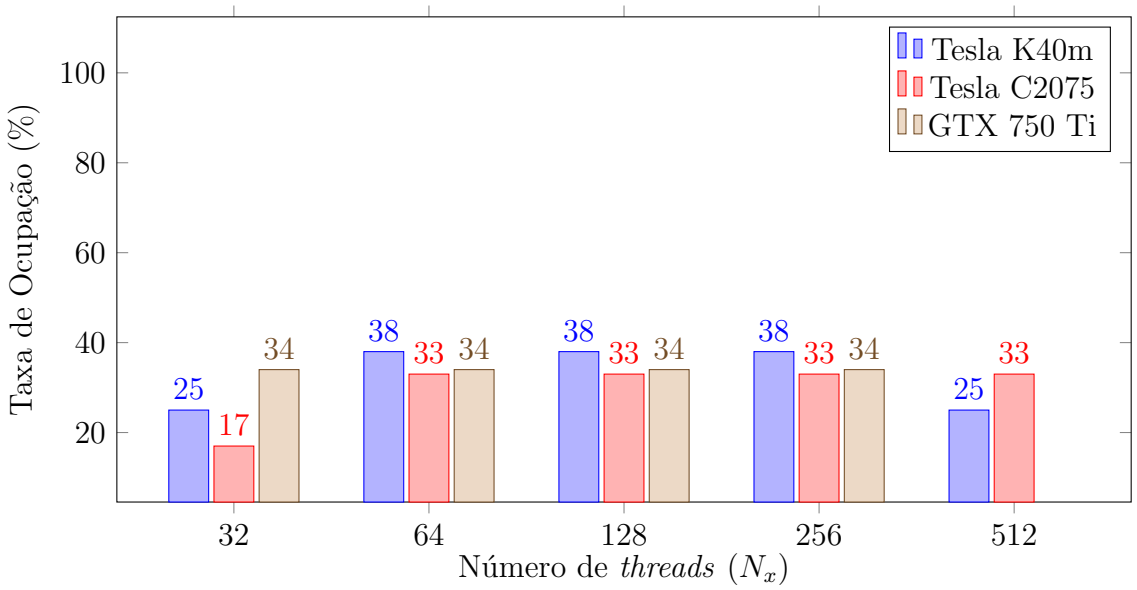
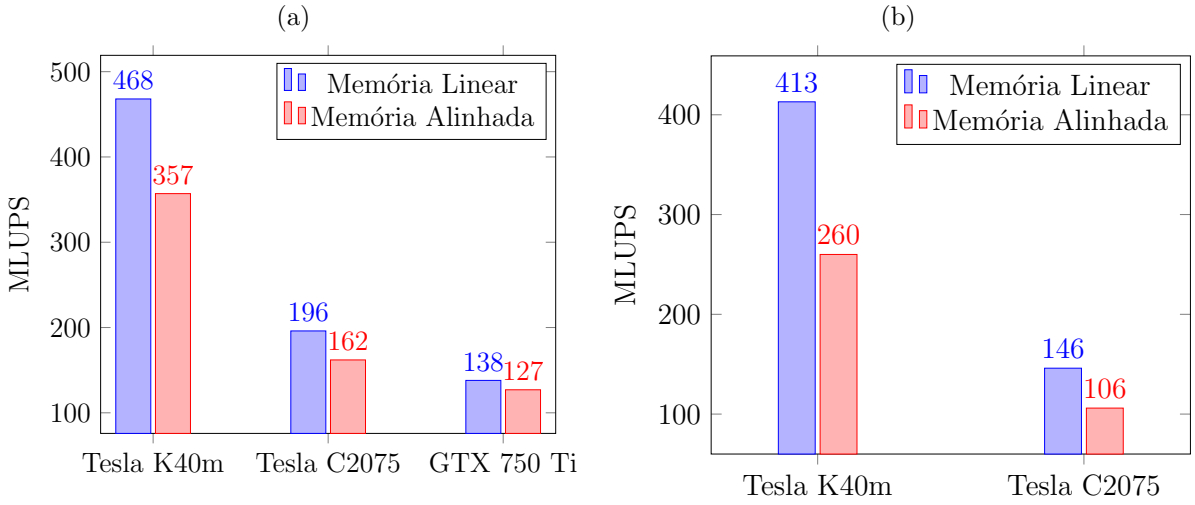


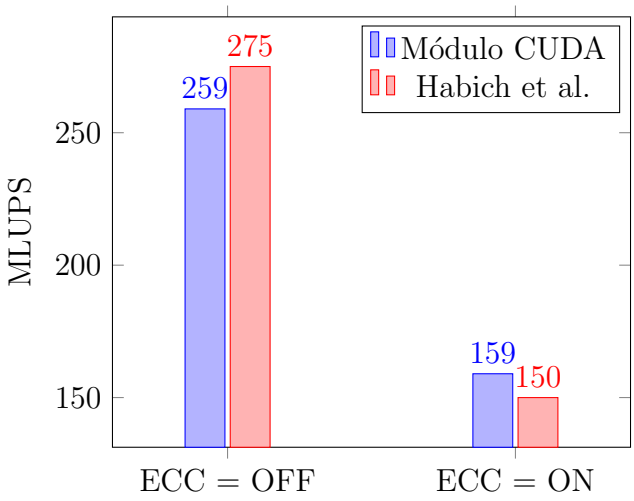
Figura 6.12: (a) Comparação do desempenho entre **memória linear** e **memória alinhada** para um domínio de tamanho 64×128×128 e com **ECC desabilitado**, (b) Comparação do desempenho entre **memória linear** e **memória alinhada** para um domínio de tamanho 128×128×128 e com **ECC habilitado**.



6.3 Teste de Comparação com a Literatura

Esta seção apresenta apenas uma comparação com um dos trabalhos relacionados. O gráfico da Figura 6.13 mostra os resultados obtidos no trabalho de Habich et al. (2011a) e os resultados alcançados pelo módulo CUDA. Com o ECC desabilitado o resultado obtido pelos autores foi superior ao nosso trabalho, porém com o uso de ECC o resultado do módulo CUDA foi um tanto superior. Esse resultado superior, obtido pelos autores, pode ser explicado pelo fato deles terem alcançado uma taxa de ocupação de 50%.

Figura 6.13: Comparação com o trabalho de Habich et al. (2011a) usando **memória linear** para um tamanho de domínio de 200×200×200.



6.4 Teste de Comparação entre o Tempo de Cópia e o Tempo de uma Iteração

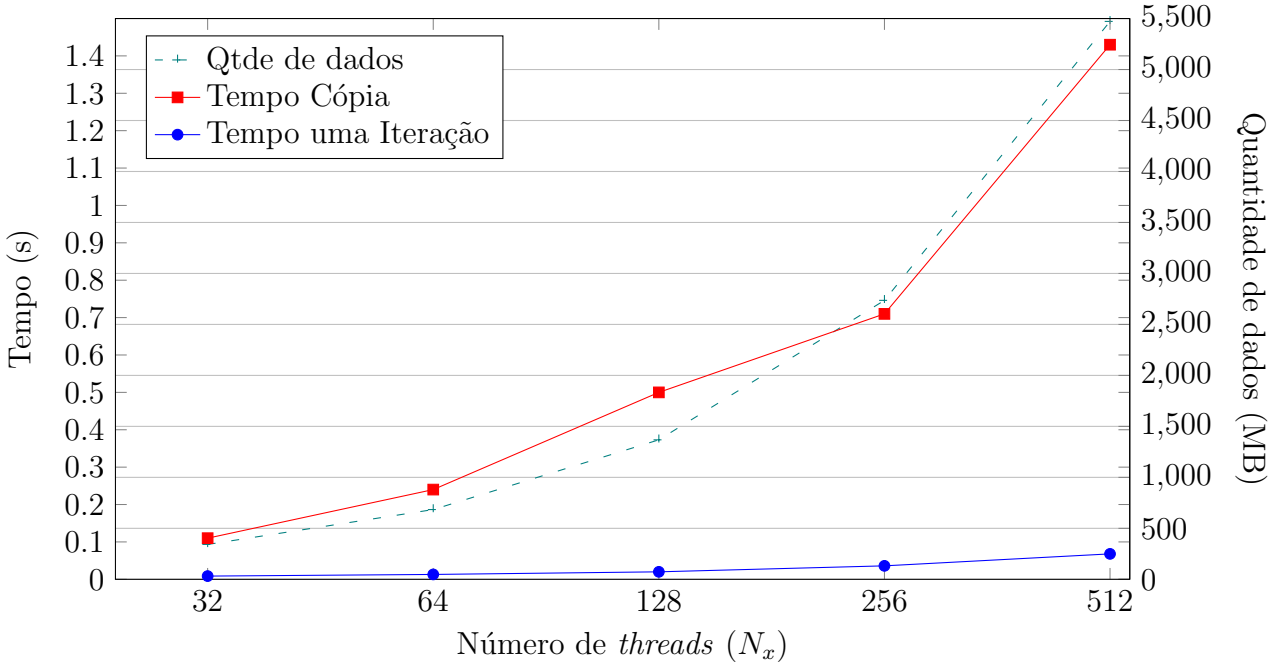
O último teste realizado compara o tempo para executar apenas uma única iteração com o tempo necessário para copiar os dados entre a CPU e a GPU. Como pode ser observado na Figura 6.14 o tempo necessário para copiar os dados é muito superior ao tempo de uma iteração. Para copiar um domínio de tamanho 32×256×256 para a GPU o tempo de cópia é 7,8 vez maior do que o tempo para realizar apenas uma iteração, entretanto essa diferença diminui quando o tamanho do domínio aumenta, pois há mais células a serem processados.

Com isso, se o domínio do problema fosse dividido entre a CPU e a GPU e a cada iteração fosse realizado a comunicação entre eles a simulação se tornaria inviável, pois o tempo total seria muito alto devido à comunicação.

Além dos tempos o gráfico também mostra a quantidade de dados transferidos para a GPU (linha tracejada). Esse teste foi apenas realizado usando a GPU Tesla K40m, pois foi a GPU que apresentou os melhores resultados para o maior tamanho de domínio e o ECC habilitado.

Após realizar todos esses testes podemos observar que a GPU que alcançou os melhores resultados foi a GPU Tesla K40m da arquitetura Kepler, seu melhor resultado foi de 612 MLUPS obtido utilizando 256 *threads* para um tamanho de domínio de 256×256×256 células e com o ECC desabilitado. Entretanto, como o LBM é um automato celular, ou seja, cada célula do *lattice* depende dos valores dos seus vizinhos qualquer valor incorreto durante a simulação pode ser propagado para as demais células invalidando seu resultado, com isso o uso de ECC se faz necessário em simulações de LBM. Portanto, com ECC habilitado o melhor resultado foi de 489 MLUPS obtido com 512 *threads* para um tamanho

Figura 6.14: Comparação entre o tempo para realizar uma única iteração do LBM e o tempo necessário para copiar os dados da CPU para a GPU Tesla K40m com **ECC habilitado** e para um domínio de tamanho $N_x \times 256 \times 256$.



de domínio de 512×256×256 células em cada direção.

Outro ponto relevante a ser observado é que o uso de memória alinhada não trouxe nenhum ganho de desempenho, ao contrário do que havíamos esperado, ou seja, o desempenho das simulações utilizando memória linear foram superiores ao uso de memória alinhada para todos os testes realizados.

A relação entre a quantidade de *threads* e o tamanho do domínio varia de acordo com cada arquitetura. Para a GPU Tesla K40m os melhores resultados foram obtidos usando os domínios de tamanho $N_x \times 256 \times 256$ e com ECC desabilitado. Já com ECC habilitado não houve diferença significativa entre os tamanhos do domínio, porém o melhor resultado foi obtido usando o tamanho de domínio 512×256×256 e com 512 *threads*.

Para a GPU C2075 o melhor resultado alcançado foi com o tamanho de domínio 512×128×128, com 512 *threads* e ECC desabilitado, entretanto não foi possível executar o mesmo teste, ou seja, com 512 *threads* para o domínio 512×256×256 por causa da quantidade de memória. Com ECC habilitado os resultados foram semelhantes, isto é, o tamanho do domínio não teve contribuição relevante.

Semelhante a GPU Tesla C2075, a GPU GTX 750 Ti obteve o melhor resultado usando o tamanho de domínio 256×128×128 e com 256 *threads*. Para o maior tamanho de domínio o melhor resultado foi obtido com 64 *threads* devido a quantidade de memória. De maneira geral o tamanho do domínio não teve influencia direta no desempenho das simulações do LBM em três dimensões nas diferentes arquiteturas de GPUs.

E, por fim, a abordagem proposta, apesar dos bons resultados obtidos se comparados com a literatura, apresentou uma taxa de ocupação muito baixa para o *kernel* que realiza os passos de colisão e propagação das partículas.

CAPÍTULO 7

CONCLUSÃO

Neste trabalho foi apresentada uma proposta de paralelização do LBM em GPU para o *framework* waLBerla. Essa proposta consiste de um novo módulo para o *framework* waLBerla, denominado módulo CUDA. Esse módulo permite que as simulações que já eram realizadas pelo *framework* waLBerla em CPU sejam agora realizadas também em GPUs NVIDIA de forma acessível ao usuário.

Para o módulo CUDA foi desenvolvido um nova estrutura de dados, definida como GPUField, que permite realizar as simulações de LBM em GPU. Além disso, foi desenvolvido um conjunto de funções que permitem copiar os dados entre a CPU e a GPU, e ainda classes para manipular e percorrer os dados na memória da GPU.

Além dessas implementações também foi necessário desenvolver uma classe responsável pela configuração dos *kernels* da GPU, visto que o *framework* waLBerla é todo desenvolvido utilizando a versão C++11 da linguagem de programação C++ e durante o período de desenvolvimento do módulo o compilador da NVIDIA (nvcc) ainda não suportava nenhuma funcionalidade do C++11. E, para demonstrar o correto funcionamento do módulo foi implementado uma versão do caso de testes *lid-driven cavity* em GPU.

Os resultados obtidos pelo módulo CUDA estão de acordo com os resultados encontrado na literatura, apesar de terem atingindo uma baixa taxa de ocupação das GPUs. A GPU que apresentou o melhor desempenho foi a GPU Tesla K40m, alcançando valores bem expressivos se comparado as demais GPUs. As simulações realizadas sobre a arquitetura Fermi (Tesla C2075) tiveram seu desempenho limitado pela quantidade de memória global da GPU. Já a GPU GTX 750 Ti apresentou os piores resultados, como era esperado, pois sua finalidade não é a computação científica, entretanto era a única GPU, desenvolvida sobre a arquitetura Maxwell, que possuíamos.

De maneira geral, a relação entre a quantidade de *threads* e o tamanho do domínio não teve influencia direta no desempenho das simulações do LBM em três dimensões nas diferentes arquiteturas de GPUs.

Na comparação entre memória linear e memória alinhada os resultados obtidos utilizando memória linear foram superiores aos resultados com memória alinhada, ou seja, a maneira como é feito o alinhamento dos dados usando a API *cudaMalloc3D()* do CUDA não trouxe ganho do desempenho das simulações do LBM em três dimensões.

Durante o desenvolvimento desse trabalho foi possível observar que o LBM é um método altamente paralelizável e associado as placas gráficas da NVIDIA produzem excelentes resultados para simulações em dinâmica dos fluidos computacionais. Entretanto,

a cópia dos dados entre a CPU e a GPU é um fator que limita o desempenho das simulações. E, apesar do *framework* waLBerla ser uma ferramenta bastante robusta sua curva de aprendizado é alta.

7.1 Trabalhos Futuros

Durante o desenvolvimento do módulo CUDA foi possível observar diversas possibilidades para trabalhos futuros, as quais serão apresentadas a seguir:

- Permitir que o domínio da simulação na GPU possa ser dividido entre vários blocos, pois a atual versão permite apenas que o domínio seja distribuído em apenas um bloco, dessa maneira cada bloco poderia ser atribuído à uma GPU diferente;
- Desenvolver um mecanismo de comunicação entre GPUs;
- Desenvolver um mecanismo de comunicação eficiente entre CPU e GPU que permita que a simulação seja processada tanto pela CPU quanto pela GPU de forma paralela;
- Otimizar a alocação de memória, ou seja, ao invés de usar o dobro de memória alocar memória suficiente para o domínio e para apenas mais um bloco;
- Para que seja possível simular outros fenômenos da dinâmica de fluidos computacional no módulo CUDA é necessário desenvolver novas condições de contorno;
- Desenvolver os operadores de colisão MRT e TRT, visto na Seção 2.4.5, por serem operadores mais estável e preciso que o SRT;
- Aumentar a taxa de ocupação, para isso é necessário diminuir a quantidade de registradores utilizados pelas *threads* em cada *kernel*;
- Desenvolver um mecanismo que auxilie a divisão do domínio do problema de forma a garantir um balanceamento de carga entre a CPU e a GPU para simulações de LBM;
- E, finalmente, uma possibilidade bastante interessante seria implementar uma técnica conhecida como *grid refinement*.

REFERÊNCIAS BIBLIOGRÁFICAS

Bailey, P., Myre, J., Walsh, S. D., Lilja, D. J., and Saar, M. O. (2009). Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 550–557. IEEE.

C++ (2015a). Class Templates. `http://www.cplusplus.com/doc/tutorial/templates/`. Acessado em 20/06/2015 11:13.

C++ (2015b). Container C++. `http://www.cplusplus.com/reference/vector/vector/`. Acessado em 08/05/2015 10:26.

d'Humières, D. (2002). Multiple–relaxation–time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451.

Donath, S., Mecke, K., Rabha, S., Buwa, V., and Rüde, U. (2011). Verification of surface tension in the parallel free surface lattice Boltzmann method in waLBerla. *Computers & Fluids*, 45(1):177–186.

Farber, R. (2009). CUDA, Supercomuting for the Masses. `http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/218100902?pgno=2`. Acessado em 18/07/2015 20:46.

Feichtiner, C., Götz, J., Donath, S., Iglberger, K., and Rüde, U. (2007). Concepts of waLBerla Prototype 0.1. Technical Report 07-10.

Feichtinger, C. (2012). *Design and Performance Evaluation of a Software Framework for Multi-Physics Simulations on Heterogeneous Supercomputers*. Verlag Dr. Hut.

Feichtinger, C., Donath, S., Köstler, H., Götz, J., and Rüde, U. (2011). WaLBerla: HPC software design for computational engineering simulations. *J. Comput. Science*, 2(2):105–112.

Fietz, J., Krause, M. J., 0003, C. S., Sanders, P., and Heuveline, V. (2012). Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In Kaklamanis, C., Papatheodorou, T. S., and Spirakis, P. G., editors, *Euro-Par*, volume 7484 of *Lecture Notes in Computer Science*, pages 818–829. Springer.

Fortuna, A. (2012). *Técnicas computacionais para dinâminca dos fluidos: conceitos básicos e aplicações*. EDUSP.

Frisch, U., dHumieres, D., Hasslacher, B., Lallemand, P., Pomeau, Y., and Rivet, J.-P. (1987). Lattice gas hydrodynamics in two and three dimensions. *Complex systems*, 1(4):649–707.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Ghia, U., Ghia, K. N., and Shin, C. (1982). High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of computational physics*, 48(3):387–411.

Godenschwager, C., Schornbaum, F., Bauer, M., Köstler, H., and Rüde, U. (2013). A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In Gropp, W. and Matsuoka, S., editors, *SC*, page 35. ACM.

Golbert, D. R., Blanco, P. J., and Feijóo, R. A. (2009). LATTICE–BOLTZMANN SIMULATIONS IN COMPUTATIONAL HEMODYNAMICS.

Götz, J., Donath, S., Feichtinger, C., Iglberger, K., and Rüde, U. (2007). Concepts of walberla prototype 0.0. Technical report, Citeseer.

Habich, J., Feichtinger, C., Köstler, H., Hager, G., and Wellein, G. (2011a). Performance engineering for the Lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. *CoRR*, abs/1112.0850.

Habich, J., Zeiser, T., Hager, G., and Wellein, G. (2011b). Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software*, 42(5):266–272.

He, X. and Luo, L.-S. (1997). Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review E*, 56(6):6811.

Ho, C.-F., Chang, C., Lin, K.-H., and Lin, C.-A. (2009). Consistent boundary conditions for 2D and 3D lattice Boltzmann simulations. *Computer Modeling in Engineering and Sciences (CMES)*, 44(2):137.

Körner, C., Pohl, T., Rüde, U., Thürey, N., and Zeiser, T. (2005). Parallel Lattice Boltzmann Methods for CFD Applications. In Bruaset, A. and Tveito, A., editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, chapter 5, pages 439–465.

Kremer, G. and Medeiros, K. (2005). *Introdução à Equação de Boltzmann, Uma*. EDUSP.

Kuznik, F., Obrecht, C., Rusaouen, G., and Roux, J.-J. (2010). LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications*, 59(7):2380–2392.

Mattila, K., Hyväluoma, J., Rossi, T., Aspnäs, M., and Westerholm, J. (2007). An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications*, 176(3):200–210.

McNamara, G. R. and Zanetti, G. (1988). Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20):2332.

Mohamad, A. A. (2011). *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. Springer.

Nan-Zhong, H., Neng-Chao, W., Bao-Chang, S., and Zhao-Li, G. (2004). A unified incompressible lattice BGK model and its application to three-dimensional lid-driven cavity flow. *Chinese Physics*, 13(1):40.

NVIDIA (2009). Whitepaper Fermi.

NVIDIA (2014a). Fermi GF100. `http://www.nvidia.com.br/object/fermi-architecture-br.html`. Acessado em 12/05/2014.

NVIDIA (2014b). Kepler GK110. `http://www.nvidia.com.br/object/nvidia-kepler-br.html`. Acessado em 17/01/2014.

NVIDIA (2014c). Whitepaper NVIDIA GeForce GTX 750 Ti.

NVIDIA (2015a). CUDA. `www.nvidia.com.br/object/cuda\_home\_new\_br.html`. Acessado em 17/06/2015 09:47.

NVIDIA (2015b). cudaDeviceSynchronize(). `http://developer.download.nvidia.com/compute/cuda/4\_1/rel/toolkit/docs/online/group\_\_CUDART\_\_DEVICE\_gb76422145b5425829597ebd1003303fe.html`. Acessado em 17/05/2015 17:57.

NVIDIA (2015c). cudaMalloc3D. `https:http://developer.download.nvidia.com/compute/cuda/4\_1/rel/toolkit/docs/online/group\_\_CUDART\_\_MEMORY\_g04a7553c90322aef32f8544d5c356a10.html\#g04a7553c90322aef32f8544d5c356a10`. Acessado em 22/04/2015.

NVIDIA (2015d). Maxwell GM107. `https://developer.nvidia.com/maxwell-compute-architecture`. Acessado em 05/05/2015 20:33.

Obrecht, C., Kuznik, F., Tourancheau, B., and Roux, J.-J. (2011). A new approach to the lattice Boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628–3638.

Obrecht, C., Kuznik, F., Tourancheau, B., and Roux, J.-J. (2013). Scalable lattice Boltzmann solvers for CUDA GPU clusters. *Parallel Computing*, 39(6):259–270.

ParaView (2015). ParaView. `http://www.paraview.org/`. Acessado em 21/07/2015.

Pohl, T., Kowarschik, M., Wilke, J., Iglberger, K., and Rüde, U. (2003). Optimization And Profiling Of The Cache Performance Of Parallel Lattice Boltzmann Codes. *Parallel Processing Letters*, 13(4):549–560.

Qian, Y. H., D'Humiéres, D., and Lallemand, P. (1992). Lattice BGK Models for Navier-Stokes Equation. *Europhysics Letters*, 17(6):479–484.

Raabe, D. (2004). Overview of the lattice Boltzmann method for nano-and microscale fluid dynamics in materials science and engineering. *Modelling and Simulation in Materials Science and Engineering*, 12(6):R13.

Rinaldi, P., Dari, E., Vénere, M., and Clausse, A. (2012). A Lattice-Boltzmann solver for 3D fluid simulation on GPU. *Simulation Modelling Practice and Theory*, 25:163–171.

Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General Purpose GPU Programming.* Addison-Wesley.

Schönherr, M., Geier, M., and Krafczyk, M. (2011). 3D GPGPU LBM Implementation on Non-Uniform Grids. *International Conference on Parallel Computational Fluid Dynamics 2011 (Parallel CFD 2001).*

Succi, S. (2001). *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Clarendon Press, Oxford.

Tölke, J. (2008). Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39.

Tölke, J. and Krafczyk, M. (2008). TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456.

TOP500 (2015a). JUQUEEN. `http://www.top500.org/system/177722`. Acessado em 29/09/2015 11:07.

TOP500 (2015b). SuperMUC. `http://www.top500.org/system/177719`. Acessado em 29/09/2015 11:06.

waLBerla (2014). waLBerla. `https://www10.informatik.uni-erlangen.de/~buildscript/walberla/`. Acessado em 12/05/2014.

Wang, Y., Wang, Y., An, Y., and Chen, Y. (2008). Aerodynamic simulation of high-speed trains based on the Lattice Boltzmann Method (LBM). *Science in China Series E: Technological Sciences*, 51(6):773–783.

Wilt, N. (2013). *The CUDA Handbook. A Comprehensive Guide to GPU Programming*. Addison-Wesley.

Wittmann, M., Zeiser, T., Hager, G., and Wellein, G. (2013). Comparison of different propagation steps for lattice Boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924–935.

Wolf-Gladrow, D. A. (2000). *Lattice-gas cellular automata and lattice Boltzmann models: An Introduction*. Number 1725. Springer Science & Business Media.

Xiong, Q., Li, B., Xu, J., Fang, X., Wang, X., Wang, L., He, X., and Ge, W. (2012). Efficient parallel implementation of the lattice Boltzmann method on large clusters of graphic processing units. *Chinese Science Bulletin*, 57(7):707–715.

Yu, D., Mei, R., Luo, L. S., and Shyy, W. (2003). Viscous flow computations with the method of lattice Boltzmann equation. *Progress in Aerospace Sciences*, 39(5):329–367.

JOSÉ AURIMAR SEPKA JÚNIOR

EXTENSÃO DO FRAMEWORK WALBERLA PARA USO DE GPU EM SIMULAÇÕES DO MÉTODO DE LATTICE BOLTZMANN

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Daniel Weingaertner

CURITIBA

2015