MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE MESTRADO EM SISTEMAS E COMPUTAÇÃO

MARCELO MACHADO COLLARES

UM RESOLVEDOR NUMÉRICO BASEADO NO MÉTODO DE LATTICE-BOLTZMANN APLICADO EM UNIDADES DE PROCESSAMENTO GRÁFICO

Rio de Janeiro
2012

INSTITUTO MILITAR DE ENGENHARIA

MARCELO MACHADO COLLARES

UM RESOLVEDOR NUMÉRICO BASEADO NO MÉTODO DE LATTICE-BOLTZMANN APLICADO EM UNIDADES DE PROCESSAMENTO GRÁFICO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientadores: Prof. Sérgio Kostin - D.Sc.
Prof^{a}. Raquel C. G. Pinto - D.Sc.,

Rio de Janeiro
2012

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

005.1 Collares, M. M.
C683r Um Resolvedor Numérico Baseado no Método de Lattice-Boltzmann Aplicado em Unidades de Processamento Gráfico/ Marcelo Machado Collares; orientado por Sérgio Kostin, Raquel C. G. Pinto – Rio de Janeiro: Instituto Militar de Engenharia, 2012.

109 p.: il.

Dissertação (mestrado) – Instituto Militar de Engenharia, - Rio de Janeiro 2012.

1. Algorítmos paralelos. 2. Programação em GPU. 3. Método de lattice Boltzmann. 4. Dinâmica de fluidos. I. Sérgio Kostin II. Raquel C. G. Pinto III. Título. IV. Instituto Militar de Engenharia.

CDD 005.1

INSTITUTO MILITAR DE ENGENHARIA

MARCELO MACHADO COLLARES

UM RESOLVEDOR NUMÉRICO BASEADO NO MÉTODO DE LATTICE-BOLTZMANN APLICADO EM UNIDADES DE PROCESSAMENTO GRÁFICO

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Sistemas e Computação.

Orientadores: Prof. Sérgio Kostin - D.Sc.
Prof[a]. Raquel C. G. Pinto - D.Sc.,

Aprovada em 01 de junho de 2012 pela seguinte Banca Examinadora:

Prof. Sérgio Kostin - D.Sc. do IME - Presidente

Prof[a]. Raquel C. G. Pinto - D.Sc., IME

Prof. Julio Cesar Duarte - D.Sc., IME

Prof. Itamar Borges Junior - D.Sc., IME

Prof. Claudio Luis de Amorim - Ph.D., UFRJ

Rio de Janeiro
2012

Dedicatória
A minha esposa Déa e à memória de minha amada mãe Ilza.

AGRADECIMENTOS

Primeiramente, agradeço a Deus pela oportunidade de chegar ao presente, almejando o futuro com a experiência adquirida no passado.

Aos meus pais, que com toda dedicação, apoiaram, confiaram, e amaram incondicionalmente, doando-se como exemplo de vida digna e honesta, proporcionando a base necessária para a formação íntegra de meu caráter.

Aos orientadores Sérgio Kostin e Raquel Coelho Gomes Pinto pelo esforço e pela dedicação despendidos em prol da conclusão deste trabalho.

A minha esposa Déa pela paciência e aos amigos e parentes pela compreensão, nos momentos em que estive ausente.

Ao amigo e companheiro de jornada Eduardo Camargo pelos sólidos conhecimentos adquiridos nas discussões de sábados e domingos de estudo no LNCC.

Ao amigo Daniel Golbert que cedeu gentilmente grande parte de seu tempo e conhecimento para contribuir com a conclusão deste trabalho.

Ao Laboratório de Modelagem em Hemodinâmica (HeMoLab) sediado no Laboratório Nacional de Computação Científica (LNCC) no qual trabalhei nos últimos quatro anos, por apoiar e consentir meu ingresso no mestrado e por disponibilizar parte do hardware utilizado nos experimentos sob o contexto das atividades desenvolvidas pelo Instituto Nacional de Ciência e Tecnologia em Medicina Assistida por Computação Científica (INCT-MACC) (INCT-MACC, 2010).

A todas as pessoas que contribuíram com o desenvolvimento desta dissertação de mestrado, tenha sido por meio de críticas, idéias, apoio, incentivo ou qualquer outra forma de auxílio.

A todos os professores e funcionários do Departamento de Engenharia de Sistemas (SE/8) do Instituto Militar de Engenharia.

Epigrafe

"*A mente que se abre a uma nova idéia jamais volta ao seu tamanho original.*"
(Albert Einstein)

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ABREVIATURAS

3D	-	*3 Dimensões*
ALU	-	*Arithmetical Logical Unit*
ULA	-	*Unidade Lógica Aritmética*
API	-	*Application Programming Interface*
BGK	-	*Bhatnagar-Gross-Krook*
CTP	-	*Cálculo de Trajetória de Partículas*
CUBLAS	-	*Basic Linear Algebra Subprograms for CUDA*
CUDA	-	*Compute Unified Device Architecture*
CUFFT	-	*Fast Fourier Transform for CUDA*
DFC	-	*Dinâmica de Fluidos Computacional*
DICOM	-	*Digital Imaging and Communications in Medicine*
ENS	-	*Equação de Navier-Stokes*
FHP	-	*Frisch, Hasslacher e Pomeau*
FLOPS	-	*Floating Point Operation Per Second*
EE	-	*Equação de Euler*
GB	-	*Gigabyte*
GFLOPS	-	*Giga Floating Point Operation Per Second*
GHz	-	*Gigahertz*
GPGPU	-	*General Purpose Graphics Process Unit*
GPU	-	*Graphics Process Unit*
HeMoLab	-	*Hemodynamics Modeling Laboratory*
HPP	-	*Hardy, Pazzis e Pomeau*
IME	-	*Instituto Militar de Engenharia*
LBM	-	*Lattice Boltzmann Method*
LGA	-	*Lattice Gas Automata*
LNCC	-	*Laboratório Nacional de Computação Científica*
MB	-	*Megabyte*
MEF	-	*Método de Elementos Finitos*
MDF	-	*Método de Diferenças Finitas*
ML	-	*Memória Local*
MP	-	*Memória Principal*

MSC	-	*Modelagem e Simulação Computacional*
MVF	-	*Método de Volumes Finitos*
MHz	-	*Megahertz*
NVCC	-	*NVidia Cuda Compiler*
OMS	-	*Organização Mundial de Saúde*
SCH	-	*Sistema Cardiovascular Humano*
SDK	-	*Software Development Kit*
SIMD	-	*Single Instruction Multiple Data*
SIMT	-	*Single Instruction Multiple Thread*
SP	-	*Stream Processor*
UCP	-	*Unidade Central de Processamento*

RESUMO

O método de *Lattice Boltzmann* (*LBM*) é uma técnica numérica alternativa para a modelagem e simulação de fluidos dinâmicos. Ele é indicado para problemas onde se deseja obter eficiência e facilidade de programação, por se tratar de um método discreto especificamente elaborado para o cálculo computacional. Possui como uma de suas principais características o grande potencial de paralelização.

Algoritmos paralelos, em regra, eram destinados ao processamento em agrupamentos de computadores, mas a evolução computacional das atuais placas gráficas permite solucionar em um único computador o que antigamente era resolvido por vários. Nesse contexto, o objetivo da presente dissertação é propor duas versões paralelas de um resolvedor numérico baseado no *LBM* aplicado em unidades de processamento gráfico (GPU - Graphics Processing Unit), utilizando-se da arquitetura de software *Compute Unified Device Architecture* (*CUDA*). A primeira implementação paralela executa um procedimento de decomposição de dados que favorece a coalescência de dados na memória global. A segunda utiliza a coalescência aliada a utilização da memória constante, que possui menor latência na hierarquia de memória da *GPU*. São avaliados ainda os ganhos de desempenho obtidos pelas versões paralelas, em relação a versão sequencial destinada ao processamento em uma única unidade central de processamento (UCP).

ABSTRACT

The Lattice Boltzmann Method (LBM) is an alternative numerical technique for modeling and simulation of fluid dynamics. It is indicate to problems where you want to achieve efficiency and easiness of programming, because it is a method specifically designed for discrete computational. Have as one of its main features the great potential for parallelization.

Parallel algorithms, as a rule, were destined to processing on clusters of computers, but the evolution of current computer graphics can solve on a single computer what once was solved by several. In this context, the objective of this dissertation is to propose two parallel versions of a numerical solver based on LBM applied on graphics processing units (GPU - Graphics Processing Unit), using the software architecture Compute Unified Device Architecture (CUDA). The first parallel implementation performs a data decomposing procedure which allows the coalescence of data in the global memory. The second uses the coalescence coupled with constant memory usage, which has lower latency in the memory hierarchy of GPU. Also are evaluated the performance gains achieved by parallel versions, in relation to sequential version designed to processing on a single central processing unit (CPU).

1 INTRODUÇÃO

1.1 CONTEXTO

Por meio da modelagem e simulação das propriedades físicas de líquidos e gases é possível obter numericamente diferentes estruturas e fenômenos físicos (SCHEPKE, 2007).

O método de *Lattice Boltzmann* (*LBM*) tem-se tornado um esquema numérico promissor para simular a complexa dinâmica de fluidos nas mais diversas aplicações (CHEN, 1992), (CHEN, 1998), (GUO, 2008) (KUZNIK, 2010). Desde a sua introdução nos anos 80 ele passou por uma série de refinamentos e extensões, que possibilitou a simulação precisa de vários fenômenos complexos de escoamentos de fluidos (GOLBERT, 2009b).

O *LBM* é utilizado em diversas aplicações que envolvem a hemodinâmica computacional. Por exemplo, na simulação de escoamentos sanguíneos em artérias cerebrais (HE, 2009), em válvulas cardíacas mecânicas (PELLICCIONI, 2007) e (KRAFCZYK, 1998), em aneurismas (HIRABAYASHI, 2006) e na aorta abdominal (ARTOLI, 2006).

A utilização de modelos matemáticos para representar a hemodinâmica computacional oferece a possibilidade de simular diversas características do comportamento do fluxo sanguíneo, como por exemplo, a velocidade do sangue dentro das artérias, a propagação da onda de pressão na árvore arterial e o estado de tensões nas paredes arteriais, fornecendo ferramentas com grande potencial para auxiliar o estudo da origem e do desenvolvimento de doenças cardiovasculares e contribuindo para o progresso no entendimento dos fenômenos físicos envolvidos no Sistema Cardiovascular Humano (SCH) (ZIEMER, 2008).

A simulação numérica envolvendo a dinâmica de fluidos possui, em regra, um alto custo computacional. Tradicionalmente o processamento deste tipo de simulação é direcionado a UCP e, buscando alto desempenho, utilizam-se ambientes paralelos como grades computacionais e supercomputadores. O grande avanço tecnológico alcançado pelas atuais placas de vídeo no processamento de dados vem atraindo a comunidade científica, que busca atender suas necessidades de computação de alto desempenho.

A evolução das atuais placas de vídeo tem origem na demanda crescente do mercado de jogos de computadores, que busca alto desempenho no processamento das imagens cada vez mais realistas.

Além do avanço tecnológico atingido pelas atuais placas de vídeo, elas possuem custo financeiro e consumo de energia menores do que sistemas dedicados ao processamento par-

alelo como grades computacionais e supercomputadores, mantendo-se com desempenho semelhante e por vezes superior (CHE, 2008).

1.2 MOTIVAÇÃO

O método *Lattice Boltzmann* (*LBM*) é um método numérico de modelagem e simulação de fluidos dinâmicos, que tem como características:

(a) Simplicidade de implementação computacional em comparação com demais metodologias clássicas da Modelagem e Simulação Computacional (MSC) como, por exemplo, o Método de Elementos Finitos (MEF);

(b) Grande potencial para simular fluxos turbulentos, fluxos em múltiplas fases e fluxos com condições de contorno irregulares (SUCCI, 1997).

(c) Possibilidade de implementação de equações aplicadas em estruturas bidimensionais ou tridimensionais denominadas *Lattices*, posicionadas sobre planos cartesianos;

(d) Grande potencial de paralelização, podendo obter ganhos importantes em termos de escalabilidade se for adequadamente combinada ao *hardware* específico;

Embora o *LBM* possua maior simplicidade de implementação computacional, pois implementa equações lineares, em comparação com demais metodologias clássicas da MSC, que implementam equações diferenciais, sabe-se que ele faz uso intenso tanto de processamento quanto de memória (GUO, 2009), (GOLBERT, 2009b) e (KUZNIK, 2010).

Devido às limitações tecnológicas na indústria de processadores relativas ao consumo excessivo de energia e sobreaquecimento, houve estagnação no aumento da frequência de *clock* dos processadores (FLYNN, 2004). Aliada às limitações tecnológicas, a necessidade de maior poder computacional levou a indústria de processadores a buscar outras soluções como a produção de processadores com mais de um núcleo (INTEL, 2008). Esses avanços fizeram com que as quantidades de núcleos nas *UCPs* e nas *GPUs* dobrassem a cada nova geração de processadores (KIRK, 2010).

Impulsionadas pela crescente complexidade do processamento gráfico, especialmente em aplicações como jogos de computadores, no ano de 2007 a *NVIDIA* lançou novos modelos de *GPU* com o intuito de disponibilizar sua arquitetura paralela para a programação de propósito geral, tornando a *GPU* uma ferramenta promissora para diversas aplicações.

1.3 PROBLEMAS CARDIOVASCULARES

Com intuito de pesquisar e desenvolver modelos apurados para simular o escoamento sanguíneo no sistema cardiovascular humano SCH (LARRABIDE, 2006) e disponibilizar ferramentas auxiliares para o diagnóstico e planejamento cirúrgico (FEIJOO), teve início no ano de 2005 o projeto *HeMoLab*. O projeto mostrou que é possível representar de forma confiável e não-invasiva o comportamento da dinâmica do fluxo sanguíneo através de técnicas de modelagem matemática e simulação computacional (ZIEMER, 2008).

Atualmente as doenças relacionadas com o Sistema Cardiovascular Humano (SCH) são as principais causadoras de mortes e responsáveis por grande parte das internações hospitalares no Brasil (WORLD HEALTH ORGANIZATION, 2011). Assim sendo, entender precisamente o comportamento da dinâmica do fluxo sanguíneo é fator importante para, por exemplo, o melhoramento das técnicas de tratamento médico e procedimentos cirúrgicos.

Conforme dados do Ministério da Saúde, ocorrem cerca de 300 mil mortes e mais de um milhão de internações por ano. A hipertensão arterial atinge aproximadamente 35% da população acima dos 40 anos e, na mesma faixa etária, 11% dos brasileiros sofrem de diabetes. O gasto anual com estas doenças em 2007 foi de R$ 684 milhões: R$ 475 milhões com internações e R$ 209 milhões na compra de medicamentos (MINISTERIO DA SAUDE, 2007).

Dentre as várias razões para usar modelos para representar o SCH, destaca-se as seguintes: (i) contribuição para o progresso no entendimento dos fenômenos físicos envolvidos no sistema; (ii) fornecer (potencialmente) ferramentas auxiliares para o estudo da origem e do desenvolvimento das doenças cardiovasculares, permitindo a predição de resultados de intervenções cirúrgicas; (iii) representar um avanço na construção de parcerias entre as grandes áreas da medicina e da modelagem; (iv) auxiliar no treinamento médico; (v) simular e visualizar o transporte de fármacos e substâncias pelo fluxo sanguíneo; (vi) predizer do risco de ruptura de aneurismas, entre outras (LARRABIDE, 2006).

1.4 OBJETIVO

O objetivo desta dissertação é implementar um versão paralelizada de um resolvedor numérico baseado no Método de *Lattice Boltzmann*, para ser executada em unidades de processamento gráfico utilizando a arquitetura *CUDA*.

As contribuições deste trabalho englobam o estudo de diversas implementações do

Método de *Lattice Boltzmann* (*LBM*), sobre as quais será proposta a realização de uma nova abordagem reunindo parte das metodologias estudadas, o desenvolvimento de uma versão sequencial destinada ao processamento em uma única unidade central de processamento (UCP), o desenvolvimento de duas versões paralelas ambas destinadas ao processamento em duas unidades de processamento gráfico *GPUs* distintas e a análise comparativa dos resultados.

Com os resultados obtidos, são avaliadas todas as versões implementadas sob três métricas: (i) quantidade de nós atualizada por segundo, na qual são avaliados desempenho e escalabilidade; (ii) percentagem do processamento de cada passo iterativo, na qual são feitas inferências sobre o desempenho de cada procedimento em relação ao tempo total de processamento; (iii) *Speedup*, no qual são feitas inferências sobre o ganho de desempenho ocorrido entre as versões paralelas e ambas em relação à sequencial.

1.5 ESTRUTURA DO TRABALHO

Este trabalho encontra-se organizado da seguinte forma:

O capítulo 2 aborda conceitos de modelagem computacional juntamente com a breve descrição do sistema HeMoLab.

O capítulo 3 trata da computação de propósito geral utilizando placas gráficas, incluindo a arquitetura das atuais placas gráficas e o modelo de programação empregado.

O capítulo 4 aborda a teoria da dinâmica de fluidos, descrevendo um breve histórico do Método de *Lattice Boltzmann* e apresentando as equações utilizadas neste trabalho.

O capítulo 5 apresenta os trabalhos relacionados ao problema.

O capítulo 6 discute a solução proposta e detalha sua implementação, relacionando as equações do *LBM* e os passos iterativos que a implementam.

No capítulo 7 serão discutidos os resultados obtidos.

Fechando o presente trabalho as considerações finais e trabalhos futuros são apresentados no capítulo 8.

2 MODELAGEM E SIMULAÇÃO COMPUTACIONAL

A modelagem e a simulação numérica vêm sendo utilizadas para estudar o comportamento do fluxo sanguíneo durante anos, utilizando métodos matemáticos clássicos da Modelagem e Simulação Computacional (MSC), tais como, por exemplo, o Método de Elementos Finitos (MEF), o Método de Diferenças Finitas (MDF), e o Método de Volumes Finitos (MVF) (FEIJOO).

Tais métodos utilizam equações diferenciais para modelar matematicamente a representação de fenômenos físicos como a dinâmica de fluidos. Suas equações possuem variáveis cujo valor deve ser determinado resolvendo-se outra equação ou inequação, que aparece sob forma da respectiva derivada. Estes métodos apresentam soluções computacionais sequenciais e paralelas, tradicionalmente direcionadas ao processamento em UCP e possuem características como a alta complexidade das suas equações e elevados tempos de processamento. Casos pequenos, como, por exemplo, na simulação do escoamento sanguíneo em uma artéria cerebral com aneurisma (FEIJOO), o aneurisma com menos de um centímetro de diâmetro em conjunto com parte da artéria foram modelados por uma geometria com 81000 nós, levando 7,5 dias para alcançar o término da simulação, utilizando um computador *SGI Altix 3700 BX2* com 32 processadores *Itanium2 1.5Ghz* e $64GB$ de memória *RAM*. Casos maiores, como, por exemplo, conjuntos de artérias ou o próprio coração podem durar semanas ou até meses (FEIJOO).

O método de *Lattice Boltzmann LBM* é um método numérico alternativo para a modelagem e simulação da física de fluidos (CHEN, 1998). Ele tem como característica a simplicidade de implementação computacional quando comparado com outras metodologias clássicas de *MSC*, sendo indicado para problemas onde se deseja obter eficiência e facilidade de programação, uma vez que o *LBM* se apresenta como um método discreto especificamente elaborado para o cálculo computacional (SCHEPKE, 2007).

2.1 HEMOLAB - LABORATÓRIO DE MODELAGEM EM HEMODINÂMICA

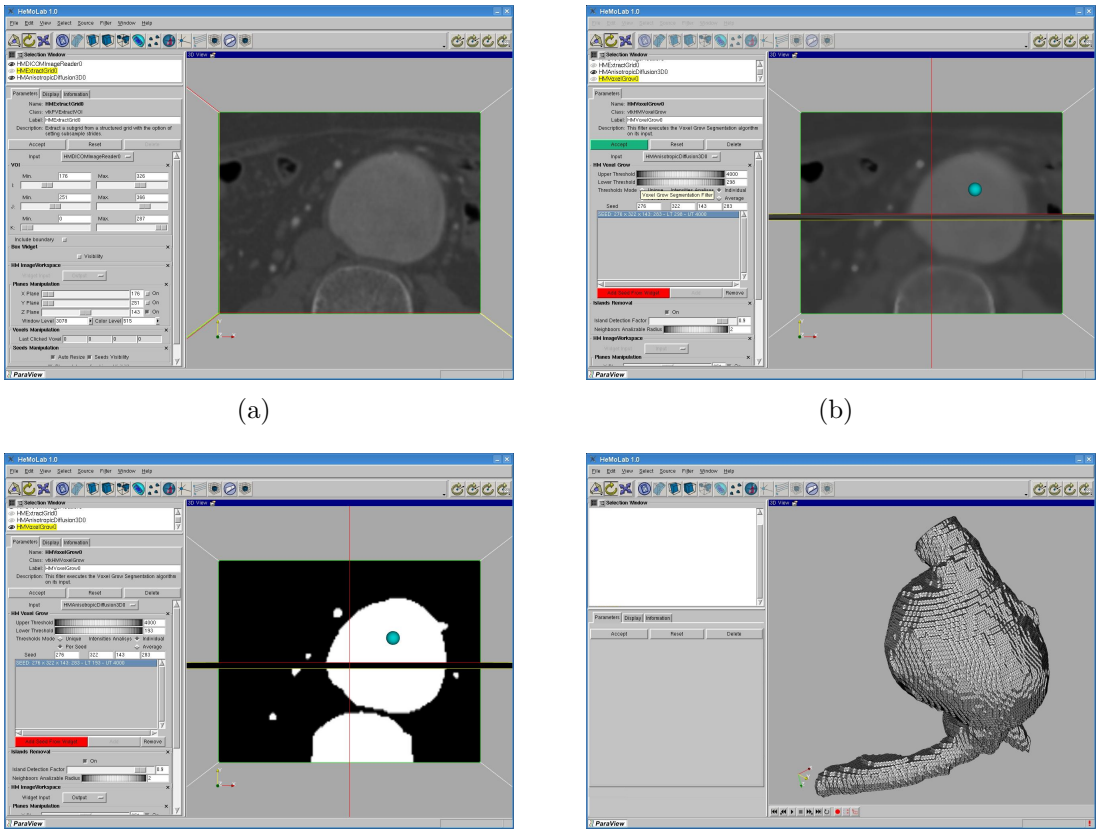
O projeto *HeMoLab* foi iniciado no ano de 2005 com o objetivo de desenvolver modelos e ferramentas computacionais para simular o Sistema Cardiovascular Humano SCH (FEIJOO).

Por meio da pesquisa e do desenvolvimento de modelos mais apurados para o escoa-

mento sanguíneo, aliado ao inicio das atividades de desenvolvimento de software, o projeto ganhou grande força no ano de 2006, visando disponibilizar ferramentas auxiliares para o diagnóstico e planejamento cirúrgico (LARRABIDE, 2006).

O *HeMoLab* atualmente é um aplicativo que agrupa dentro de um mesmo ambiente diferentes funcionalidades que permitem criar modelos do SCH, por meio de ferramentas de processamento e segmentação de imagens (FEIJOO). O *HeMoLab* é um sistema modular e seus principais módulos são: Módulo de Processamento de Imagens, Módulo *1D*, Módulo *3D*, e Módulo de Acoplamento que serão brevemente descritos a seguir:

O *módulo de processamento de imagens* realiza a leitura de imagens médicas no formato *DICOM* (NEMA) e trabalha as imagens com o objetivo de extrair geometrias gerando malhas tridimensionais, ilustrado na figura 2.1.



(a) (b)

(c) (d)

FIG. 2.1: Módulo Processamento de Imagens

(a) Tratamento com filtros de suavização de imagem, com o objetivo de remover os ruídos produzidos pelo equipamento de aquisição de imagens médicas.

(b) Seleção da região de interesse para que filtros de segmentação façam a sua extração (paredes arteriais, tecidos, etc.). O ponto em azul que se encontra no centro da

imagem seleciona a região.

(c) Identificação da região de interesse representada pela redução do conjunto de tons de cinza da imagem, reduzida de muitos tons para apenas dois, preto e branco. A parte branca representa a região de interesse e a parte preta a região que será descartada.

(d) Aplicação de um filtro que cria um contorno sobre a região de interesse, gerando assim uma malha chamada de superfície (conjunto de triângulos) que representa as paredes de um vaso arterial.

O *módulo 1D* oferece a representação do SCH de maneira simplificada através de *segmentos* e *terminais*, formando uma árvore arterial como mostra a figura 2.2. Os *segmentos* representam artérias subdivididas em *elementos* e *nós*. Cada *elemento* pode ser configurado por diversas características mecânicas como, por exemplo, elastina e colágeno. Cada *nó* é um ponto de controle que define as características geométricas do *segmento*, como por exemplo: largura da parede arterial, e raio do segmento. São utilizados *terminais* para representar a influência dos vasos de menor calibre, presentes nas extremidades (LARRABIDE, 2006). O *coração* é um tipo de *terminal* especial com características particulares, as quais não serão abordadas.

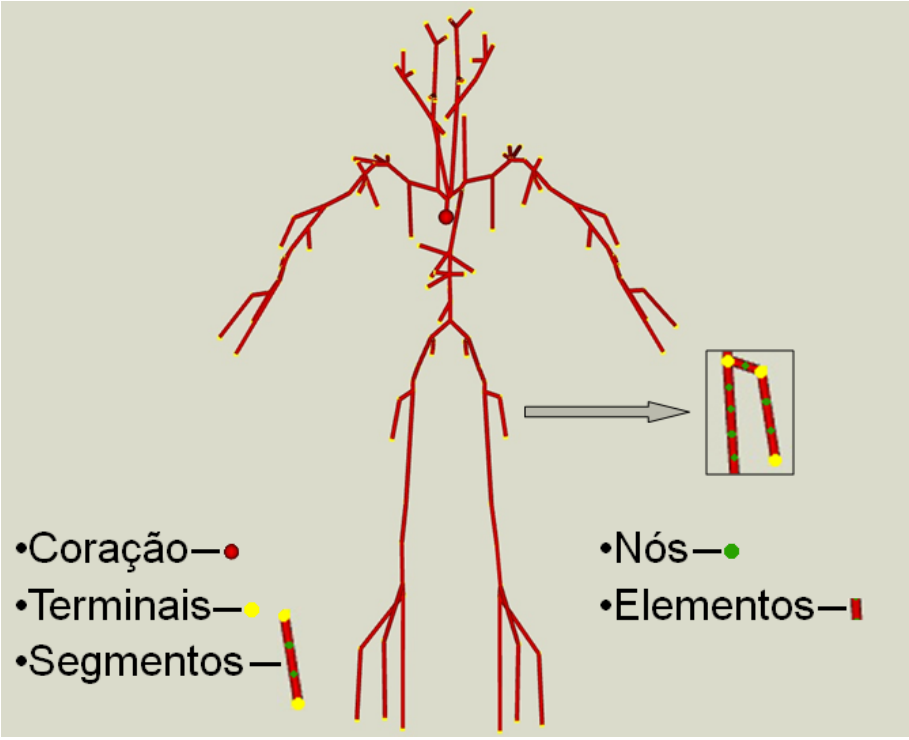


FIG. 2.2: Árvore Arterial do Módulo *1D*

Com as características do modelo *1D*, como por exemplo elastina, colágeno, largura da

parede arterial e raio do segmento, adequadamente configuradas, este módulo utiliza tais configurações para submeter o modelo *1D* à simulação numérica. A simulação numérica é feita por uma ferramenta externa que utiliza um resolvedor de propósito geral denominado *SolverGP* (URQUIZA, 2002). A geometria *1D* é salva em arquivo para posteriormente ser visualizada de forma individual ou utilizada em modelos acoplados pelo *módulo de acoplamento*.

O *módulo 3D* oferece ferramentas que permitem refinar a geometria obtida pelo módulo de processamento de imagens, que é reticulada e pode ser tratada antes de ser submetida à simulação.

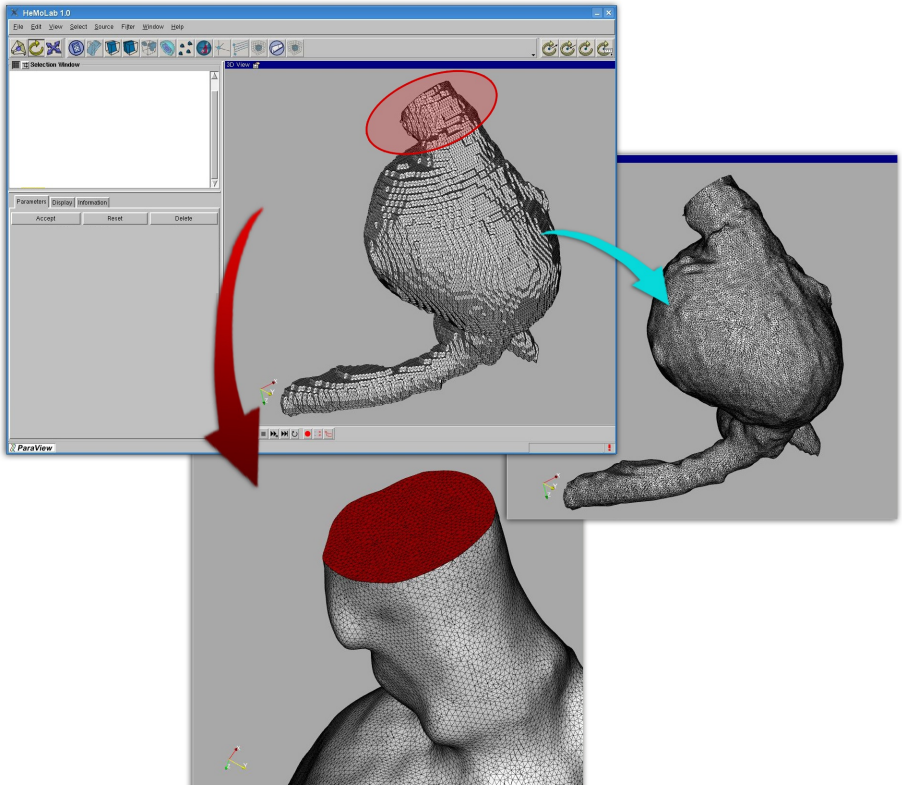


FIG. 2.3: Exemplo tratamento da malha executado pelo Módulo *3D*.

Conforme mostra a figura 2.3, a seta em vermelho indica a aplicação de um filtro que insere novos grupos de triângulos com o intuito de fechar a geometria, a esses grupos dá-se o nome de tampas. Segundo (LARRABIDE, 2006), as superfícies geradas a partir de imagens médicas costumam ser de má qualidade, possuindo triângulos denominados agulhas, que representam triângulos com área próxima de zero sendo o comprimento das arestas maior do que zero, ou muito pequenos onde todas as arestas são da mesma ordem mas a área do elemento é próxima de zero. Neste caso são necessários procedimentos que permitam melhorar a superfície eliminando esse tipo de característica. Neste contexto, indicado pela seta em azul são utilizados filtros que processam os triângulos da geometria para melhorar a sua qualidade, que segundo (LARRABIDE, 2006) uma malha de boa qualidade é aquela em que todos os seus triângulos são praticamente equiláteros.

Uma vez que a condição anterior seja satisfeita, é realizado o preenchimento da malha de superfície com tetraedros, gerando uma malha de volume. Após os refinamentos, a malha *3D* é submetida a simulação numérica com o *SolverGP* (URQUIZA, 2002). A

geometria *3D* é salva em arquivo para posteriormente ser visualizada de forma individual ou utilizada em modelos acoplados pelo *módulo de acoplamento*

O *módulo de acoplamento* utiliza as geometrias geradas pelos modelos *1D* e *3D* para a criação de modelos híbridos, associando uma tampa da geometria *3D* com um *nó* da geometria *1D*. Esse recurso de associação é feito manualmente, permitindo a escolha de qual tampa ficará ligada a qual *nó*. Na figura 2.4 em amarelo está o terminal 3 do modelo *1D* sendo acoplado a tampa 1 do modelo *3D*. Após a configuração do modelo acoplado, utilizando os modelos *1D* e *3D*, esses dados podem ser submetidos a simulação numérica utilizando o *SolverGP* (URQUIZA, 2002).

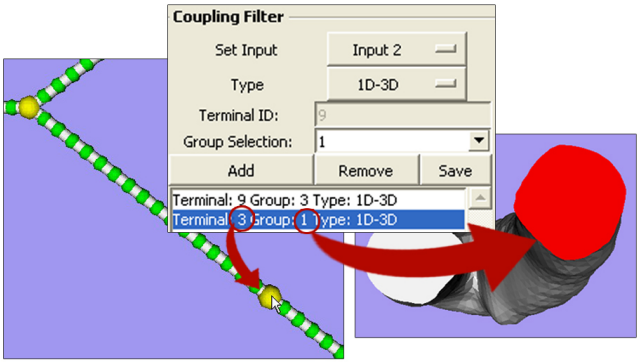


FIG. 2.4: Acoplamendo de uma tampa da geometria *3D* com um *nó* do modelo *1D*

O *SolverGP* é uma ferramenta externa ao *HeMoLab*, que recebe como dados de entrada uma geometria gerada pela saída dos módulos *1D*, *3D* ou um acoplamento em formato específico. Como resultado, o *SolverGP* escreve arquivos de saída contendo vetores de velocidade, deslocamento e valores escalares de pressão associados a geometria resultante. Os resultados então são lidos pelo *HeMoLab*, onde poderão ser tratados com técnicas de visualização e computação gráfica para ter um melhor entendimento sobre o conjunto de dados produzido. A relação do *SolverGP* com o *HeMoLab* é ilustrada na figura 2.5.

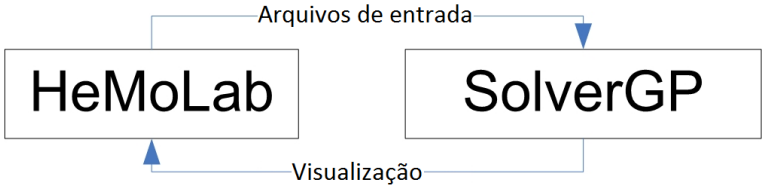


FIG. 2.5: Resolvedor numérico de propósito geral - *SolverGP*

O *SolverGP* é descrito em (LARRABIDE, 2006) como uma arquitetura genérica, a qual permite implementar facilmente resolvedores numéricos para métodos como *MEF*, *MDF*, *MVF*, assim como resolvedores para outros métodos discretos como redes elétricas,

redes neurais, entre outros, tanto para problemas estacionários como transientes. Tais modelos não podem ser resolvidos de forma analítica, e por este motivo são usados métodos numéricos os quais fornecem um resultado o mais aproximado possível.

Como exposto anteriormente, o sistema *HeMoLab* é dividido em módulos que se complementam e interagem. Para produzir os resultados esperados, é necessário o encadeamento dos módulos de forma que a saída de um seja a entrada do seguinte, conforme figura 2.6:

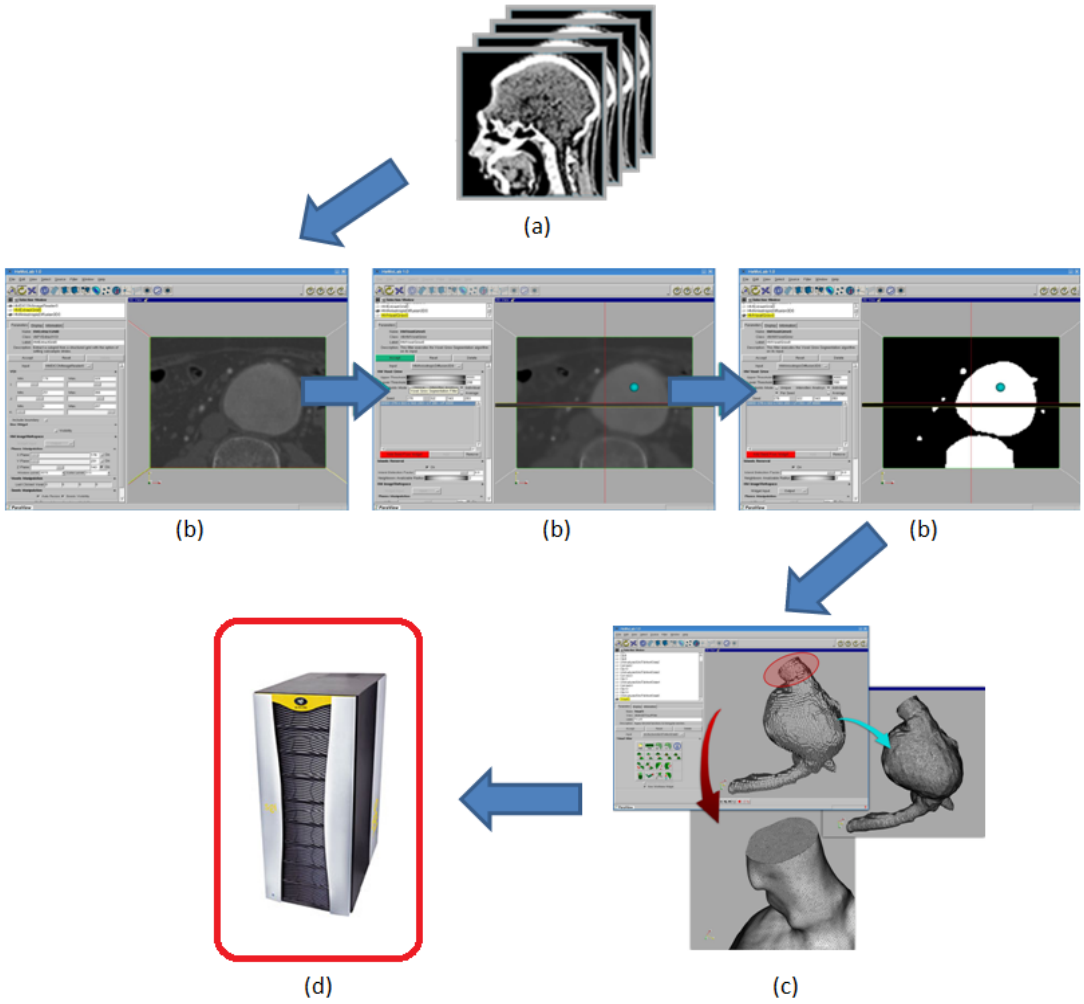


FIG. 2.6: Pipeline do processo de visualização

(a) Leitura de imagens médicas no formato *DICOM* adquiridas de exames de ressonância magnética (RM) ou tomografia computadorizada (TC);

(b) Utilização do módulo de processamento de imagens para tratamento (remoção de ruídos), segmentação e extração de uma geometria (vaso arterial);

(c) Utilização do módulo 3D para criação das tampas e geração de malhas de volume.

(d) Simulação numérica utilizando as malhas geradas, foco desta dissertação.

O *Paraview* possui a capacidade de adição de novos módulos e possui seu código aberto (*Open Source*), sendo amplamente utilizado pela comunidade científica para tarefas de visualização (ZIEMER, 2008). O *HeMoLab* foi desenvolvido como uma extensão do *ParaView* (KITWARE, 2009a), agregando toda a funcionalidade e versatilidade de uma ferramenta originalmente desenvolvida para visualização de dados científicos.

Com interface gráfica simples que pode ser parametrizada pelo próprio usuário, este *software* também incorpora métodos de processamento de imagens, além de proporcionar suporte para computação distribuída e para vários sistemas operacionais.

O *ParaView* é baseado em uma das mais utilizadas bibliotecas para visualização e renderização *3D*, chamada *VTK (Visualization ToolKit)* (KITWARE, 2009b) (SCHROEDER, 2004).

O *VTK* é uma ferramenta de código aberto, implementado em *C++* e usa primitivas[1] *OpenGL* para a renderização. Ele oferece inúmeras estruturas, geometrias, leitores e escritores para os mais variados formatos assim como conversores de estruturas.

O *ParaView*, em conjunto com o *VTK*, forma uma ferramenta que executa aplicações *VTK* com suporte para computação distribuída, oferecendo assim uma interface para leitura dos mais variados tipos de dados, utilização de filtros para processamento e segmentação de imagens e visualização de dados.

[1]Primitivas são comandos que desenham as formas básicas do *OpenGL* (pontos, linhas, polígonos e imagens). As formas são utilizadas como peças para montar a cena desejada.

3 GRAPHICS PROCESS UNIT (GPU) AND COMPUTED UNIFIED DEVICE ARCHITECTURE (*CUDA*)

Neste capítulo será apresentado um breve histórico da programação de propósito geral usando *GPU*. Em seguida, será descrita a arquitetura das *GPUs* (*Graphics Process Units*) atuais e como é o seu funcionamento. Por último, será descrita a utilização da arquitetura *CUDA*.

3.1 HISTÓRICO DE EVOLUÇÃO DA PROGRAMAÇÃO DE PROPÓSITO GERAL EM GPU

A intensa utilização das *GPUs* nos últimos anos resultou em aumento na pesquisa experimental com *hardware* gráfico. Em (TRENDALL, 2000) há um resumo detalhado dos tipos de cálculos implementados em *GPUs*.

As *GPUs* são utilizadas também por pesquisadores em diferentes aplicações não gráficas tais como: utilização de rasterização para planejar a movimentação de um robô (LENGYEL, 1990), utilização de técnicas de z-buffer para calcular diagramas de *Voronoi* (HOFF, 1999), quebra da criptografia de senhas de sistemas *UNIX* (KEDEM, 1999), implementação de métodos de *Ray Tracing* (traçado de raio) (CARR, 2002) (PURCELL, 2002), e também para o cálculo de redes neurais artificiais (BOHN, 1998).

Nas técnicas antigas, uma aplicação candidata à implementação em *hardware* gráfico tinha que ter seus dados passados à *GPU* na forma de vértices ou pixels, sendo processados por *Pixel Shaders* e *Vertex Shaders* (HUANG, 2008).

As limitações de frequência das UCPs e a crescente busca por maior poder de processamento levaram a indústria de processadores a produzir processadores com múltiplos núcleos (INTEL, 2008). Esses avanços fizeram com que o número de núcleos nas *UCPs* e nas *GPUs* dobrassem a cada nova geração de processadores (KIRK, 2010).

Na busca constante de maior rapidez no processamento dos dados, as unidade de processamento gráfico, com arquitetura baseada em *Pixel Shaders* e *Vertex Shaders*, evoluíram para disponibilizar sua arquitetura paralela com maior usabilidade à comunidade científica. Em 2006, a *NVIDIA* (NVIDIA, 2009) lançou na série 8 de suas placas gráficas uma arquitetura denominada de *G80*, unificando os *Pixel Shaders* e *Vertex Shaders* em *Stream Processor* programáveis, permitindo a utilização dinâmica dos anti-

gos *Shaders*. A antiga arquitetura gerava problemas, pois diferentes aplicações podiam requisitar mais de um tipo de *Shader* do que de outro, fazendo com que parte dos *Shaders* ficassem ociosos.

A unificação dos *Shaders* tira do fabricante a responsabilidade de estimar o número ideal de *Pixel Shaders* e *Vertex Shaders*, maximizando o número de *Stream Processor* programáveis. Seu novo modelo de programação é baseado na organização da aplicação em *streams* e *kernels* (KAPASI, 2003). Os dados são enviados em forma de *streams* ao *kernel*, que os processa e os devolve na mesma forma. Essa *stream* resultante pode ser reenviada de forma encadeada a outro *kernel* ou simplesmente retornar como resultado.

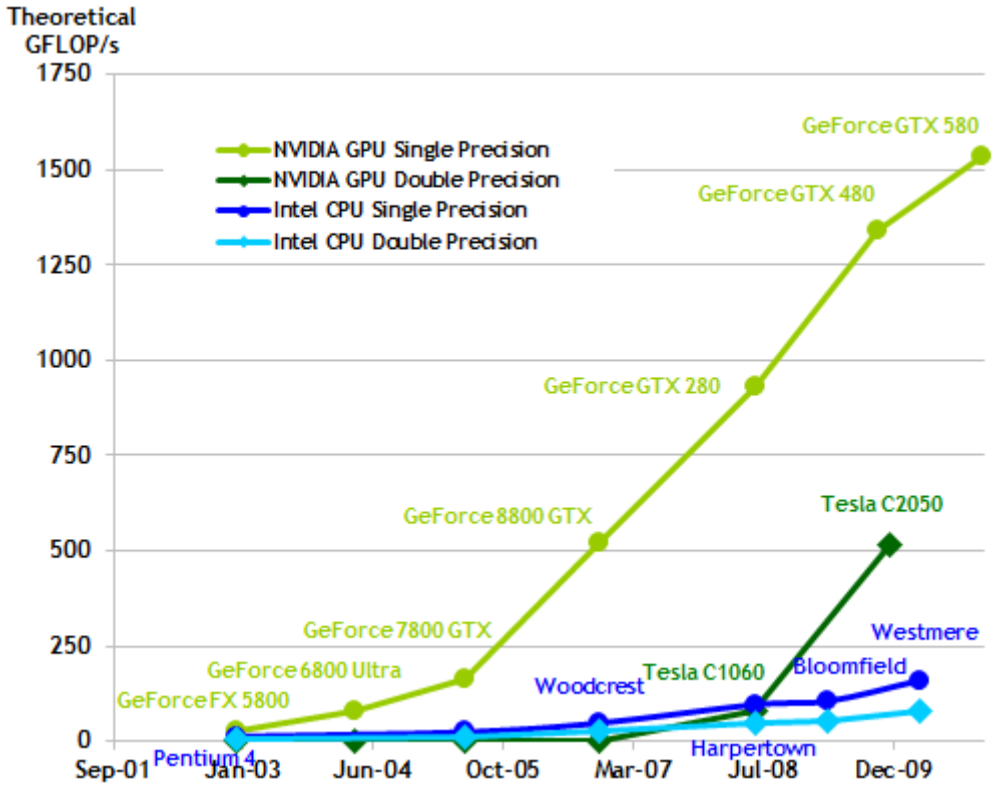


FIG. 3.1: Operações de ponto flutuante por segundo. Fonte (NVIDIA, 2011).

Desta forma, uma *GPU* trabalha com multiprocessadores com grande poder computacional, e como ilustrado na figura 3.1, a partir do lançamento da série 8, as *GPUs* tiveram um maior destaque em relação a UCP em operações de ponto flutuante por segundo (*FLOPS*).

A razão de tal diferença deve-se ao fato da *GPU* possuir uma arquitetura especializada para computação intensiva e paralela, enquanto que a UCP é projetada para tratar de aplicações sequenciais com um controle de fluxo altamente complexo.

A arquitetura das *GPUs* é projetada de tal forma que a maior parte dos transístores são dedicados ao processamento de dados, em vez de cache de dados e controle de fluxo como ocorre na UCP (NVIDIA, 2009). Essa dedicação da *GPU* se deve à característica paralela das aplicações gráficas, que realizam a mesma operação em um grande volume de dados (CASTANO-DIEZ, 2008).

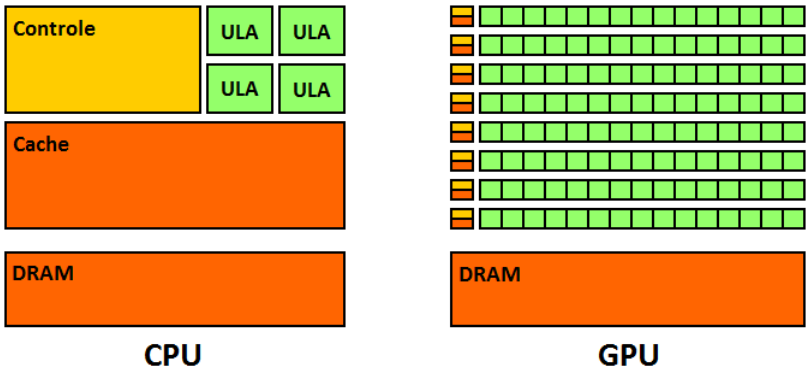


FIG. 3.2: Utilização dos transístores em *GPUs* e *UCPs* (NVIDIA, 2009)

Pode-se ver no esquema de utilização de transístores disposto na figura 3.2 que o espaço destinado ao controle de fluxo e cache de dados na UCP é distribuído na *GPU* entre as ULAs (Unidades Lógicas Aritméticas), buscando esconder a latência do acesso à memória com processamento, ao invés de grandes caches de dados como feito pela UCP.

A arquitetura *G80* impulsionou a utilização da programação de propósito geral para *GPU*, e baseado em uma grande revisão nesta arquitetura, surgiu a segunda geração de *GPUs* da *NVIDIA* com arquitetura denominada *GT200* (NVIDIA, 2009).

A arquitetura *GT200* foi lançada em 2008, e trouxe como uma das principais novidades o aumento do número de *Stream Processors* de 128 para 240, na ocasião chamados de *CUDA cores*. A quantidade de transístores foi dobrada, permitindo um número maior de *threads* executadas simultaneamente. Foi incluído também suporte a processamento de dados com precisão dupla, visando atender a demanda proveniente da comunidade científica.

Com a experiência adquirida na produção das arquiteturas *G80* e *GT200*, incluindo os conhecimentos adquiridos acerca dos aplicativos desenvolvidos e publicados pelos consumidores, principalmente os da comunidade científica (NVIDIA, 2011), em 2011 foi lançada pela *NVIDIA* a mais atual arquitetura das *GPUs*, denominada *FERMI*. Esta arquitetura está descrita na próxima seção.

3.2 ARQUITETURA DAS GPUS ATUAIS

A arquitetura *FERMI* representa o maior avanço na tecnologia das *GPUs* desde a criação da arquitetura *G80*. O comparativo entre a arquitetura *FERMI* e suas antecessoras pode ser visto na tabela 3.1.

TAB. 3.1: Comparação entre os *chips G80*, *GT200* e *FERMI*. Fonte: (NVIDIA, 2011)

GPU	**G80**	**GT200**	**FERMI**
Transístores	681 milhões	1,4 bilhões	3,0 bilhões
CUDA Cores	128	240	512
Operações de ponto flutuante em dupla precisão	Não	30 ops/clock	256 ops/clock
Operações de ponto flutuante em precisão simples	128 ops/clock	240 ops/clock	512 ops/clock
Escalonador de *Warp* por SP	1	1	2
Memória com suporte ECC	Não	Não	Sim
Kernels concorrentes	Não	Não	Até 16
Comprimento de endereços nas operações de leitura/escrita a memória global	32-bits	32-bits	64-bits
Lançamento	Nov/2006	Jun/2008	Mar/2011

Uma *GPU* baseada na arquitetura *FERMI* implementa 3,0 bilhões de transístores, traduzidos em 512 *CUDA cores* organizados em 16 multiprocessadores (MP) com 32 *CUDA cores* cada, 6 partições de memória *GDDR5* com suporte para até 6 *GB* de dados (1 *GB* para cada partição), como pode ser visto na figura 3.3.

Além do incremento na quantidade de *CUDA cores* e no armazenamento de memória, outras características foram implementadas nesta arquitetura que não estão presentes nas anteriores, e são elas:

(a) *NVIDIA* Parallel DataCache - assim denominada pela *NVIDIA* (NVIDIA, 2011), implementa uma hierarquia de cache em dois níveis para os dados da memória global. O primeiro nível chamado L1 possui um tamanho de 64Kb e fica localizado em cada MP, podendo ser utilizado cooperativamente pelos SPs contidos no MP. O segundo nível L2 possui um tamanho de 768Kb e fica disponível a todos os SPs da *GPU*.

(b) *NVIDIA GigaThread*^{tm} - assim denominada pela *NVIDIA* (NVIDIA, 2011), provê a execução paralela de *kernels*, transferência de dados bidirecional e gerenciamento inteligente para dezenas de milhares de *threads*;

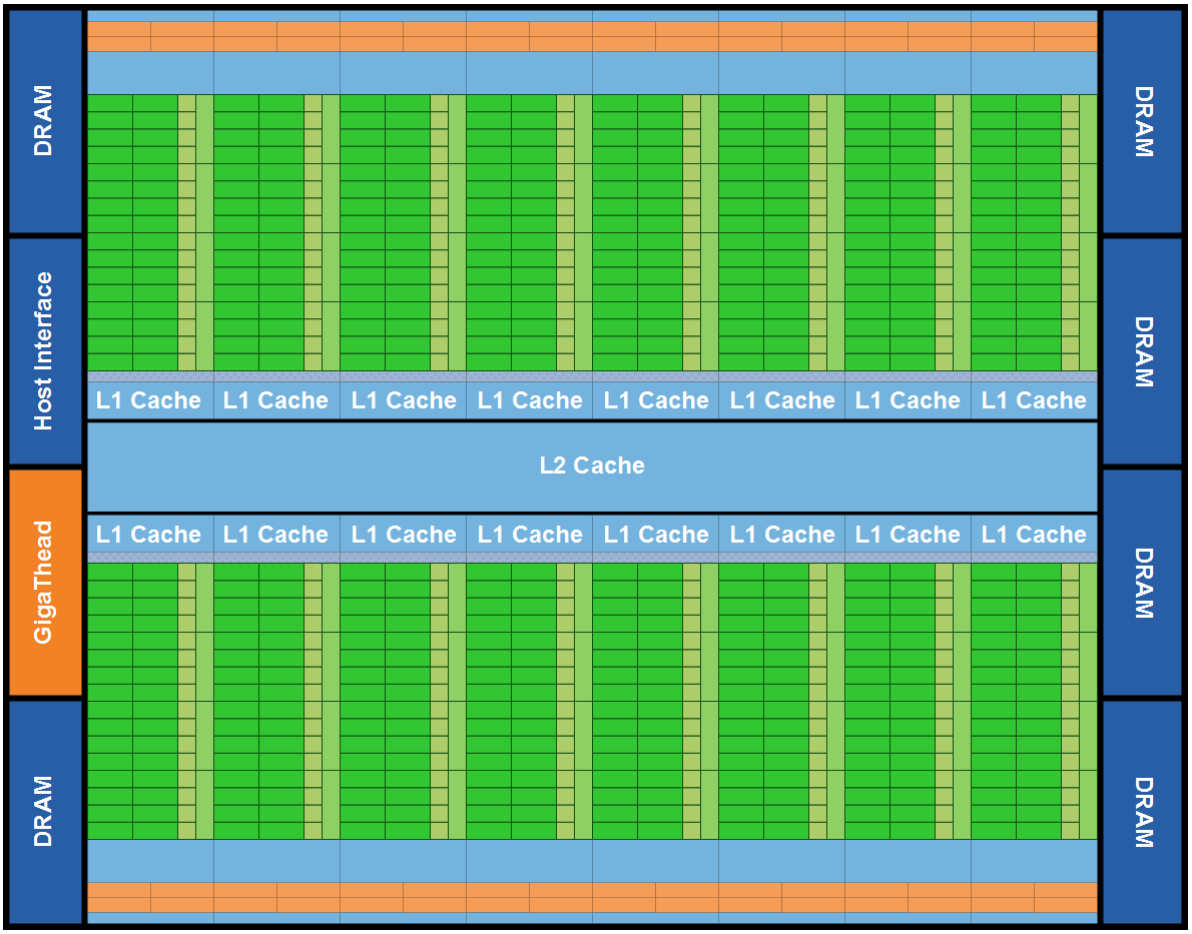


FIG. 3.3: Arquitetura *FERMI*.

(c) Suporte a ECC (*Error Correction Codes*), método usado para identificar e corrigir erros no armazenamento ou transmissão de dados, provendo proteção para arquivos de registro, memórias compartilhadas, caches e memória global.

(d) Suporte a códigos C++ orientados à objetos;

(e) Suporte a endereços de memória de 64-bits expandindo a capacidade de memória que pode ser adicionada a placa gráfica;

(f) Operações de ponto flutuante em dupla precisão até 8 vezes mais rápidas e em conformidade a especificação IEEE 754-2088.

Para manter o pipeline ocupado com vários *threads*, as *GPUs* utilizam uma granularidade fina em sua execução. Em cada ciclo de *clock*, um núcleo termina a execução de uma instrução em um *thread* diferente, executando ciclicamente uma instrução para cada

thread. Isso evita o *forwarding* que ocorre nos pipelines das *UCPs* diminuindo assim a quantidade de circuitos de controle para essa finalidade.

Além de evitar o *forwarding*, serve também para mascarar acessos à memória global, que é uma operação relativamente lenta. Para que essa técnica seja eficiente, deve-se utilizar uma aplicação com alto grau de paralelismo de maneira a manter o pipeline ocupado.

3.3 ARQUITETURA CUDA

O *CUDA* é uma arquitetura de *software* para computação massivamente paralela, que utiliza o poder de processamento das *GPUs* da *NVIDIA* para atingir o alto desempenho (RYOO, 2008). A plataforma foi introduzida formalmente em fevereiro de 2007 (NVIDIA, 2009) e atualmente, encontra-se em sua versão 4.1 (NVIDIA, 2011).

Na arquitetura *CUDA*, a *GPU* é vista como um coprocessador da UCP, no qual o desenvolvedor é responsável por escalonar as atividades entre a *GPU*, chamada de *device*, e a UCP, chamada de *host*. Cabe ao desenvolvedor configurar os acessos aos diversos tipos de memória disponíveis e a quantidade de *threads* (processos leves) que serão utilizados para atingir melhor desempenho.

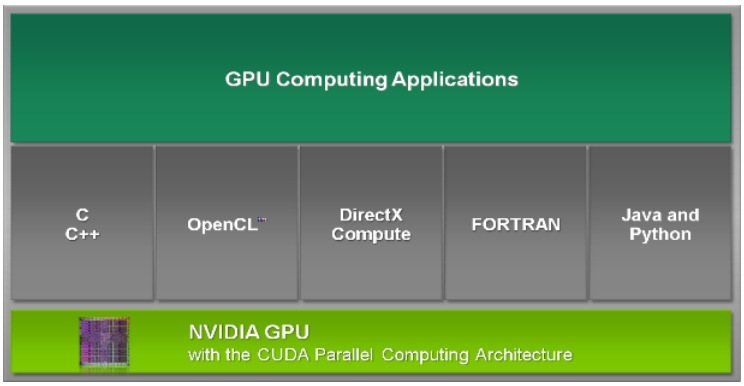


FIG. 3.4: Linguagens e *APIs* suportadas pelo *CUDA* (NVIDIA, 2009)

Como mostra a figura 3.4, a arquitetura fornece suporte a várias linguagens e *APIs* (*Application Programming Interfaces*) para o desenvolvimento de aplicações. Na sua versão 4.1 podem ser utilizadas linguagens de alto nível, como *C*, *C++*, *Fortran*, *Java* e *Python* ou *APIs* do driver, como *OpenCL* e *DirectX Compute*.

Para que se possa utilizar a arquitetura *CUDA*, é necessário primeiramente satisfazer alguns requisitos de *hardware* e *software*.

Para satisfazer o requisito de *hardware*, é necessário uma placa gráfica com suporte a arquitetura de software *CUDA*. Basicamente, a partir da série 8 da *NVIDIA* (*NVIDIA GeForce* 8***), as placas apresentam esse suporte (RYOO, 2008); contudo, uma extensa lista de *GPUs* pode ser encontrada em (NVIDIA, 2009).

Para satisfazer os requisitos de *software*, são necessários no mínimo 2 pacotes para instalação: o *driver* específico para o modelo do *hardware*, e o *CUDA Toolkit* que contém o compilador juntamente com algumas ferramentas e bibliotecas. Adicionalmente, pode-se instalar um pacote com códigos de exemplo (*code samples*); todos são fornecidos pela *NVIDIA* em (NVIDIA, 2011).

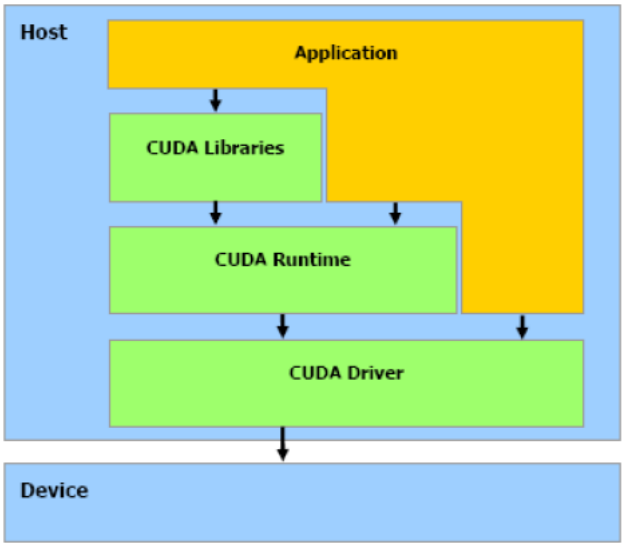


FIG. 3.5: A pilha de *software* da plataforma *CUDA* (NVIDIA, 2009)

A figura 3.5 ilustra a pilha de *software* da plataforma *CUDA*, composta pelo *driver* de acesso ao *hardware*, um componente de tempo de execução e as bibliotecas prontas para uso. No topo dessa pilha encontra-se a *API Application Programming Interface* da plataforma *CUDA*.

Segundo (CHE, 2008), para melhor aproveitar o processamento em placas gráficas é importante conhecer bem a arquitetura de hardware da *GPU* e a arquitetura de software do *CUDA*, para assim adequar o algoritmo ao processamento neste contexto.

3.3.1 MODELO DE PROGRAMAÇÃO

O modelo de programação *CUDA* possui um conceito de execução ordenada e paralela de *threads*. Os *threads* são agrupados em blocos e os blocos agrupados em grades. A configuração da quantidade de blocos na grade e a quantidade de *threads* no bloco são

definidas antes da chamada do *kernel*, por duas variáveis do tipo *dim3*. Esse tipo se assemelha a um *struct* em *C* com 3 componentes do tipo inteiro. No caso de omissão de valores é atribuído o valor 1 ao componente.

A grade é a estrutura básica para alocação de blocos na qual é determinado o número máximo de blocos por grade. As grades podem ter uma ou duas dimensões.

O bloco é a estrutura básica para alocação de *threads* no qual é determinado o número máximo de *threads* por bloco. Um bloco pode ter uma, duas ou até três dimensões.

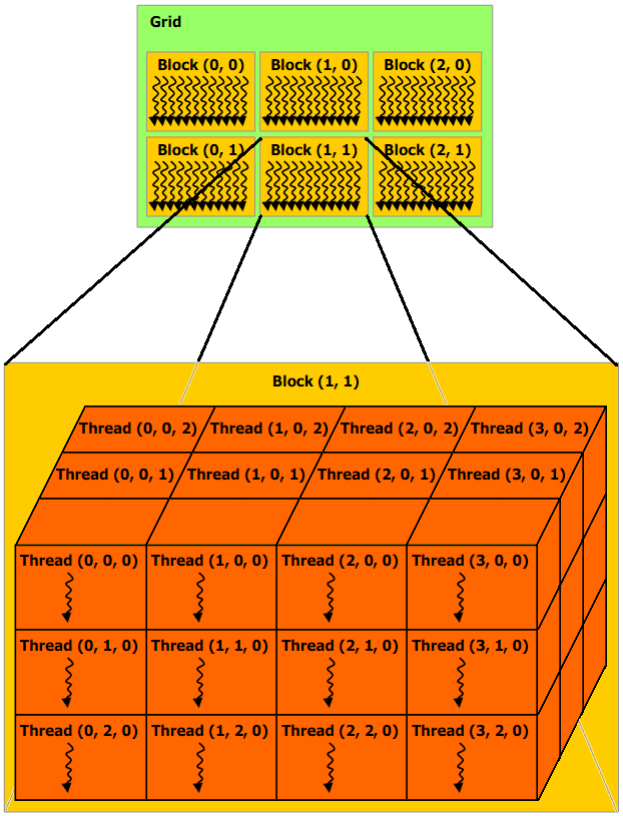


FIG. 3.6: Exemplo de um *grid* com seus blocos e *threads* (NVIDIA, 2009)

Para se executar uma chamada de *kernel*, o mesmo precisa ser informado com quantos blocos e *threads* irá trabalhar. Deste modo antes de efetuar uma chamada de *kernel*, duas variáveis do tipo *dim3* precisam ser configuradas. Uma para a quantidade de blocos e outra para a quantidade de *threads*.

Pseudo-Código 3.1: Chamada do *kernel*

```
1 dim3 dimGrade(2, 3, 1);
2 dim3 dimBloco(3, 4, 3);
3 ChamadaDeKernel <<< dimGrade, dimBloco >>> (Parametro);
```

Para exemplificar a configuração e chamada de um *kernel* toma-se como exemplo o pseudo-código 3.1. A linha 1 mostra a configuração da variável *dimGrade* do tipo *dim3*, com a quantidade de blocos contidos na grade *2x3x1*. Como grades não podem ser tridimensionais, identifica-se o terceiro componente com o valor 1. A linha 2 mostra a configuração da variável *dimBloco* do tipo *dim3*, com a quantidade de *threads* contidos em cada bloco. Na linha 3 é feita a chamada da função de *kernel*, no qual o mesmo é configurado com o número de blocos e *threads* entre <<< >>>. Adicionalmente podem ser passados parâmetros ao *kernel*, neste caso a variável *Parametros* entre (). O esquema de configuração de blocos e *threads* pode ser visto na figura 3.6, onde neste exemplo seriam executados 216 *threads*, em 6 blocos com 36 *threads* cada.

Variáveis Built-in	Significado	Dimensão	Acesso
threadIdx	Índice do thread no bloco	X	threadIdx.x
		Y	threadIdx.x
		Z	threadIdx.z
blockIdx	Índice do bloco na grade	X	blockIdx.x
		Y	blockIdx.x
		Z	blockIdx.z
blockDim	Dimensão do bloco	X	blockDim.x
		Y	blockDim.x
		Z	blockDim.z
gridDim	Dimensão da grade	X	gridDim.x
		Y	gridDim.y

TAB. 3.2: Variáveis globais disponíveis na arquitetura *CUDA*.

Os *threads* são indexados conforme hierarquia de dois níveis, representados por duas variáveis do *runtime CUDA* (*blockIdx* e *threadIdx*), que podem ser acessadas de qualquer parte do *kernel*. A tabela 3.2 mostra tais variáveis, onde *blockIdx* indexa os blocos da grade e *threadIdx* indexa os *threads* de cada bloco. A variável *gridDim* armazena a configuração da quantidade de blocos em cada dimensão da grade, e a variável *blockDim* armazena a configuração da quantidade de *threads* em cada dimensão dos blocos. Todas as variáveis *Built-in* são do tipo *dim3*.

Para gerenciar a sincronização dos *threads*, o *CUDA* possui funções de barreira. Quando um *thread* atinge tal barreira fica aguardando até que os demais *threads* alcancem o mesmo ponto, quando neste caso é dada sequência ao processamento. O *CUDA* provê dois tipos de sincronização: entre os *threads* do mesmo bloco e entre todos os *threads* de um *kernel*.

O fluxo básico de processamento de uma aplicação desenvolvida em *CUDA* inclui, a cópia dos dados da memória principal da UCP para a memória global da *GPU*, o

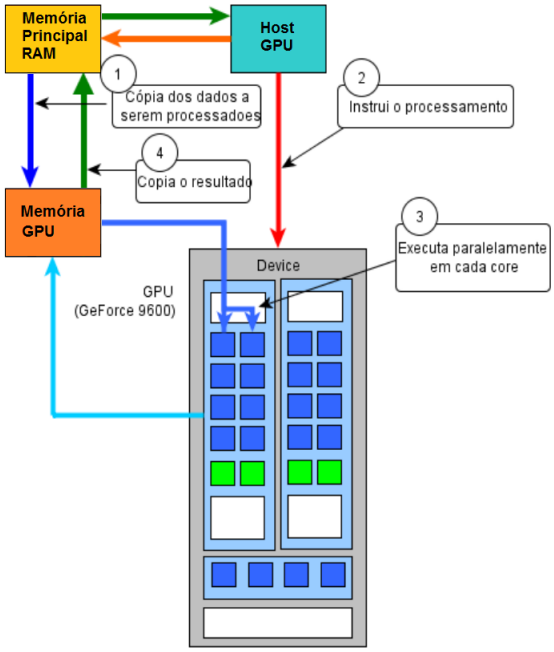


FIG. 3.7: Fluxo de execução de um programa *CUDA*

processamento propriamente dito, e o retorno dos resultados da memória global da *GPU* para a memória principal do computador. A figura 3.7 mostra esse fluxo de processamento.

3.3.2 HIERARQUIA DE MEMÓRIA

A *GPU* possui uma hierarquia de memória que pode ser utilizada com o objetivo de melhorar o desempenho das aplicações. Os tipos de memória podem ser vistos na tabela 3.3 e na figura 3.8.

TAB. 3.3: Qualificadores de variáveis em CUDA

Declaração da variável	**Memória**	**Escopo**	**Tempo de vida**
Variável automática	Registrador	*Thread*	*Kernel*
__device__ __shared__ int var	Compartilhada	Bloco	*Kernel*
__device__ int var	Global	Grade	Aplicação
__device__ __constant__ int var	Constante	Grade	Aplicação

O tempo de vida de uma variável está ligado ao tipo de memória onde está armazenada, podendo ser em memória global, constante, compartilhada, ou registradores. A visibilidade de uma variável está ligada ao escopo onde ela é criada, definindo assim quais *threads* poderão acessá-la. Caso um *thread* utilize uma variável declarada dentro da função *kernel*, essa variável será armazenada em um registrador, e será criada uma cópia privada para cada *thread* que utilizar este *kernel*, ficando o seu tempo de vida restrito ao *thread*.

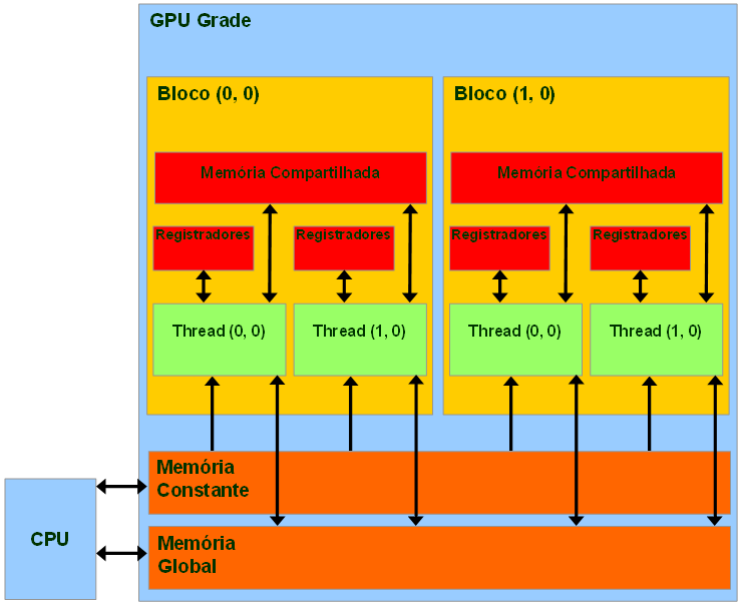


FIG. 3.8: Hierarquia de memória.

Cada bloco possui uma memória compartilhada e cada *thread* possui seus próprios registradores. Os registradores e a memória compartilhada oferecem os menores tempos de acesso na hierarquia de memória, suportando acessos paralelos a altas velocidades.

Os registradores são utilizados pelos *threads* para armazenar suas próprias variáveis, chamadas de variáveis automáticas. Elas não são visíveis a outros *threads* e só existem enquanto o *thread* que a criou existir.

A memória compartilhada tem a serventia de promover a cooperação entre os *threads* de um mesmo bloco, proporcionando maior velocidade no acesso. Os dados armazenados em memória compartilhada tem o tempo de vida associado ao bloco, sendo visível por todos os *threads* nele contido.

As variáveis que precisam estar acessíveis para todos os *threads* de todos os *kernels* devem ser armazenadas em memória global ou em memória constante. A memória global possui a maior latência se comparada aos demais tipos de memória disponibilizadas pela arquitetura, sendo úteis para trafegar dados entre *kernels* no decorrer da aplicação. É também a memória com a maior capacidade de armazenamento, podendo chegar a 6 GB como é o caso na arquitetura *FERMI* já apresentada na figura 3.3.

Os dados armazenados na memória constante se localizam fisicamente na memória global sendo mantidos em cache pelo *runtime CUDA*, o que garante a baixa latência na leitura de seus dados. A memória constante só pode ser escrita de fora do *kernel*, podendo ser acessada de dentro dos *kernels* por qualquer *thread* de forma simultânea, sendo esse

acesso interno restrito à somente leitura.

4 DINÂMICA DE FLUIDOS E O LBM

4.1 TEORIA DA DINÂMICA DE FLUIDOS

Um fluido pode ser descrito fisicamente de forma macroscópica, tratando as propriedades físicas dos materiais (sólidos ou fluidos) sem caracterizar a microestrutura dos seus componentes. Nesse contexto, utilizam-se equações diferenciais para representar as leis primordiais da física, como por exemplo a conservação de massa e momento e para caracterizar especificamente o material a ser analisado (LANDAU, 1982).

As equações mais importantes para descrever a dinâmica de fluidos computacional são a Equação de Euler (EE) e a Equação de *Navier-Stokes* (ENS) (LANDAU, 1982), pois relacionam as propriedades envolvidas na física dos materiais da macroestrutura.

Embora seja possível a simulação macroscópica, somente em alguns casos ela pode gerar resultados precisos (BUICK, 1997). Isso deve-se ao fato da macroestrutura não possuir a sensibilidade pertinente na microestrutura, fazendo com que propriedades físicas existentes na microestrutura permaneçam constantes por conta da grande diferença entre as escalas de tempo e espaço (WOLF, 2006), gerando soluções apenas de forma aproximada.

Baseado na relevância da microescala, é possível simular a dinâmica de fluidos de forma computacional como ilustra a figura 4.1, utilizando um modelo molecular capaz de fornecer bons resultados, no qual neste caso as moléculas interagem de forma aproximada às leis de Newton (WOLF, 2006) podendo ser definidos tanto o posicionamento quanto a velocidade das moléculas (BHATNAGAR, 1954).

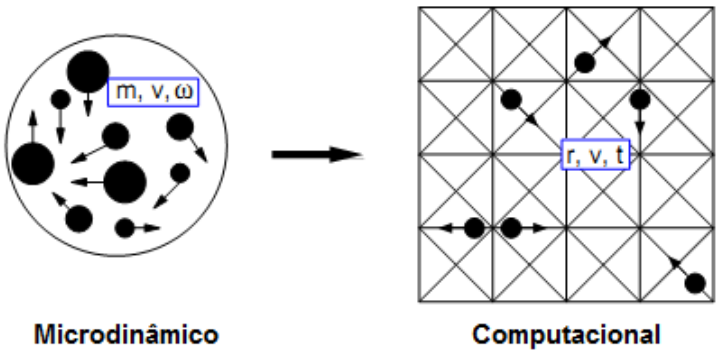


FIG. 4.1: Tradução do modelo molecular para o modelo computacional (SCHEPKE, 2007)

Quanto às propriedades físicas moleculares, estas são obtidas pela análise de resultados estatísticos sobre o comportamento obtido com a iteração entre as moléculas. Computacionalmente esse cenário possui limitações devido às pequenas escalas de espaço e tempo, distantes daquelas encontradas em aplicações reais (WOLF, 2006).

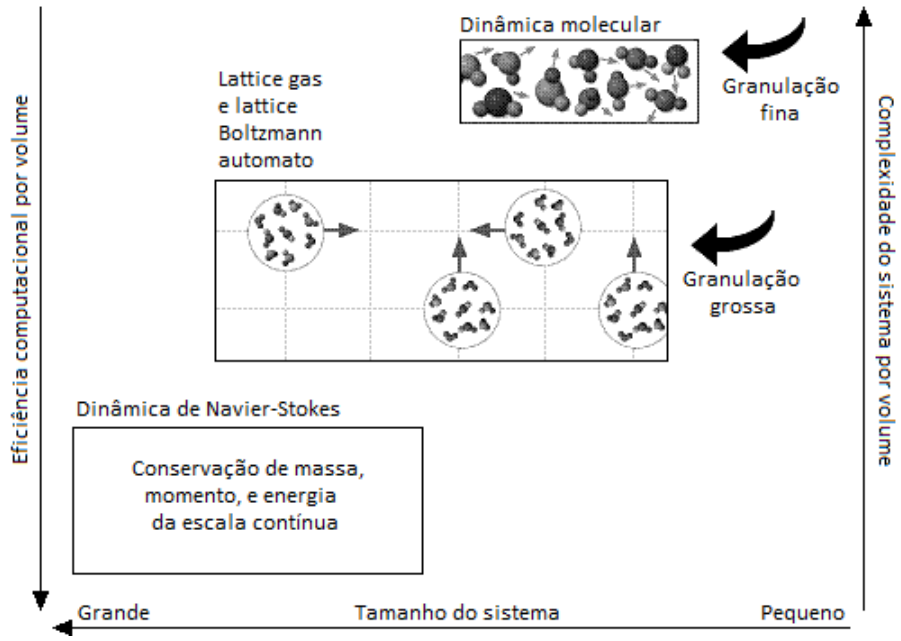


FIG. 4.2: Relação entre abordagens microscópicas, mesoscópicas e macroscópicas (RAABE, 2006)

A figura 4.2 ilustra a relação existente entre as abordagens microscópica, mesoscópica e macroscópica, no que diz respeito a complexidade por volume e eficiência computacional. A pertinência de granulação fina e baixas escalas existente em abordagens microscópicas, a torna menos eficiente computacionalmente e com maior complexidade por volume[2]. Embora as abordagens macroscópicas apresentem uma menor complexidade por volume, a negligência das iterações ocorridas entre as moléculas, decorrentes da microescala, pode afetar a modelagem de fenômenos complexos. Abordagens mesoscópicas se mostram em uma boa posição devido ao fato de se manterem entre as abordagens microscópicas e macroscópicas pois podem aproveitar características das duas.

Abordagens mesoscópicas são assim chamadas, pois se mostram em posição intermediária entre as abordagens microscópicas e macroscópicas, podendo aproveitar características das duas.

[2]Quanto maior o volume maior a complexidade

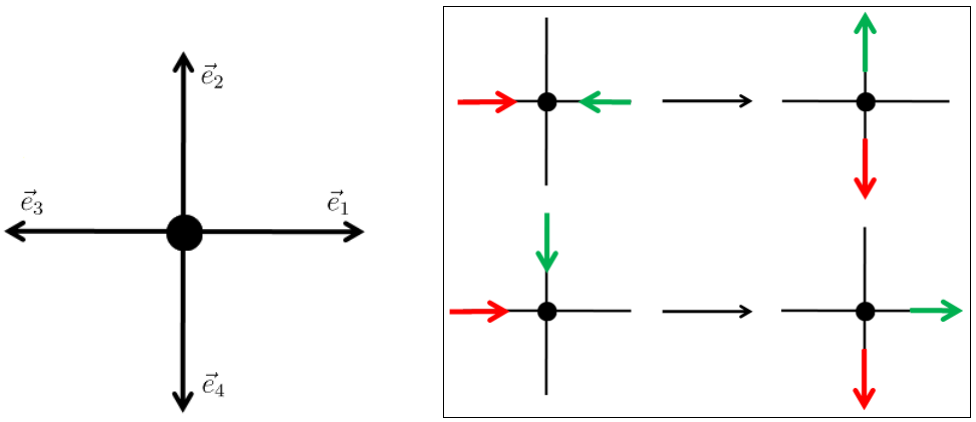
4.2 TEORIA DO MÉTODO DE LATTICE BOLTZMANN

O Método de *Lattice Boltzmann* (*LBM*) tem sua origem histórica no método de *Lattice Gas Celular Automata* (*LGCA*) mas com uma abordagem mesoscópica, passando a modelar distribuições de micro-partículas ao invés de cada micro-partícula. De acordo com (CHOPARD, 2002), o *LBM* é também uma forma simplificada da equação cinética de *Boltzmann* no qual os detalhes de movimentação molecular são descartados, com exceção daqueles necessários para a recuperação do comportamento macroscópico correto. As origens do *LBM* serão expostas nas seções 4.2.1 e 4.2.2 a seguir.

4.2.1 TEORIA DO MÉTODO DE LATTICE GAS CELULAR AUTOMATA LGCA

A utilização de equações cinéticas para simular a dinâmica de fluidos pode ser encontrada na história no trabalho de (BROADWELL, 1964), no qual o modelo apresenta um único módulo de velocidade para simular o escoamento de fluidos. O modelo de Broadwell pode ser visto como uma equação de Lattice Boltzmann unidimensional, com velocidade discreta das partículas sendo o tempo e o espaço contínuos.

Modelos totalmente discretos, mais conhecidos como *Lattice Gas* ou *Lattice Gas Celular Automata* (*LGCA*) foram propostos por (HARDY, 1976) nos quais o autor apresenta um método que aplica a discretização de tempo e espaço sobre um *Lattice* quadrado, com o propósito de entender as propriedades de transporte dos fluidos. Tal método é denominado *HPP* proveniente dos nomes dos seus autores (*Hardy, Pazzis e Pomeau*). O modelo do *Lattice* do método *HPP* denomina-se *D2Q5*, onde *D2* expressa 2 dimensões e *Q5* expressa 4 direções de movimento mais 1 ponto estático. Tal modelo é ilustrado na figura 4.3(a), e sua regra de colisão na figura 4.3(b).

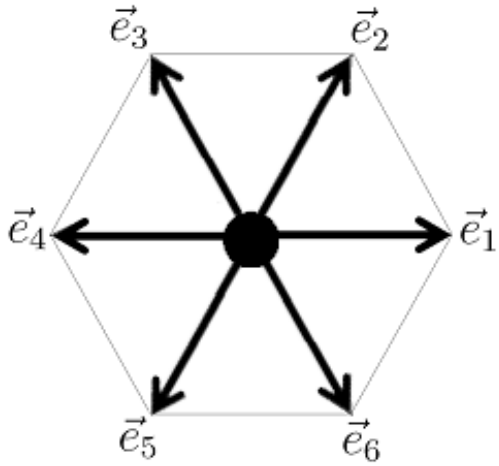


(a) Modelo *HPP* (b) Regras de colisão do modelo *HPP*

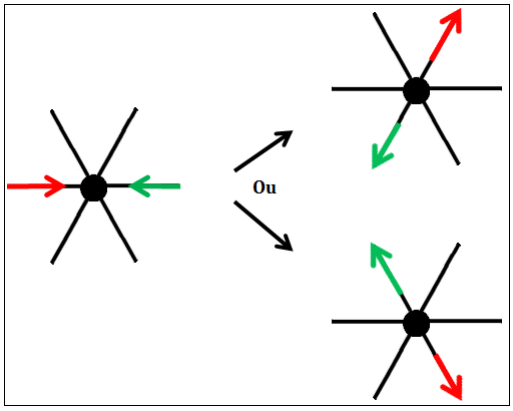
FIG. 4.3: Modelo *HPP* (Hardy, Pazzis e Pomeau) e regra de colisão.

Utilizando a relevância da microestrutura e abstraindo suas limitações, dez anos após a concepção do método HPP que buscava o entendimento das propriedades de transporte na hidrodinâmica bidimensional, *Frisch, Hasslacher e Pomeau* (*FHP*) (FRISCH, 1986) reconhecendo a relevância da simetria do *Lattice* para a recuperação das ENS, obtiveram pela primeira vez as *ENS* corretas a partir do método *LGCA* para estruturas bidimensionais, desta vez aplicando em *Lattices* hexagonais, sendo o método denominado *FHP*; *Pomeau* um dos autores do método *HPP* anterior.

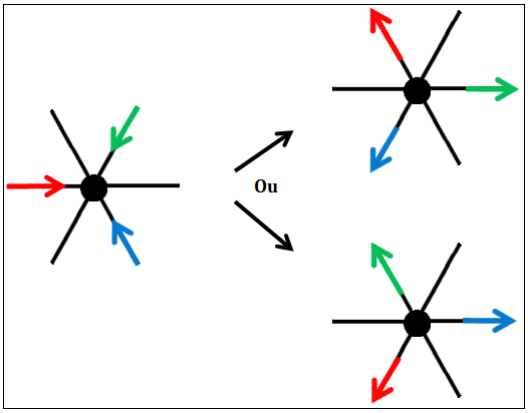
O modelo do *Lattice* do método *FHP* denomina-se *D2Q7*, onde *D2* expressa 2 dimensões e *Q7* expressa 6 direções de movimento mais 1 ponto estático. Tal modelo é ilustrado na figura 4.4(a), e possui regra de colisão que utiliza uma variável booleana para definir se a colisão ocorrerá no sentido horário ou anti-horário, e tal regra se aplica para 2, 3 ou 4 partículas, ilustradas nas figuras 4.4(b), 4.4(c), e 4.4(d) respectivamente.



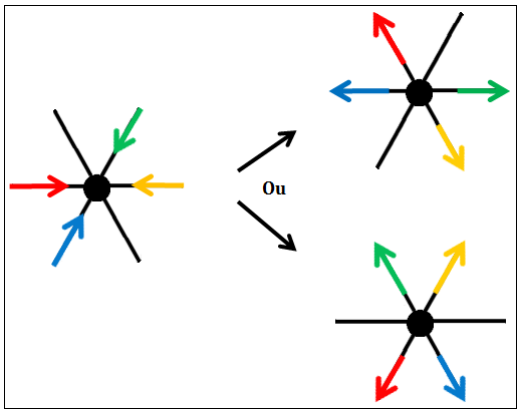
(a) Modelo *FHP*



(b) Regra de colisão para 2 partículas



(c) Regra de colisão para 3 partículas



(d) Regra de colisão para 4 partículas

FIG. 4.4: Modelo *FHP* (Frisch, Hasslacher e Pomeau) e regras de colisão.

Os autores de (FRISCH, 1986) descobriram que o movimento molecular dentro de um

fluido, não necessitava de um alto nível de detalhamento para simular a dinâmica dos fluidos de maneira mais realista, revelando a possibilidade de representar um fluido com partículas fictícias com massas e velocidades iguais, limitadas a um conjunto finito de elementos. Em cada elemento ocorrem iterações entre as partículas tomando como base a vizinhança do elemento.

A movimentação dessas partículas fictícias tem sua origem em um *Lattice* simétrico. Primeiramente, o *Lattice* é preparado para que não haja mais de uma partícula deslocando-se na mesma direção, em uma determinada célula. As partículas então se deslocam a cada passo de tempo, em uma unidade de *Lattice*, movendo-se até uma célula vizinha, tendo a sua direção impressa por sua velocidade.

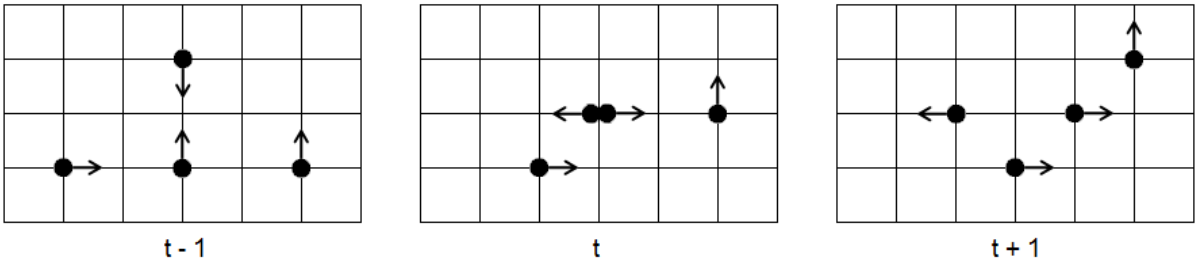


FIG. 4.5: Modelo de propagação e colisão do $LGCA$

Em algum passo de tempo durante a movimentação, as partículas que se movem na figura 4.5 no tempo *t - 1* se encontram na mesma célula de destino provocando uma colisão como visto no tempo t, ocasionando que suas velocidades sejam alteradas. Com base nas velocidades das partículas envolvidas na colisão, é seguida uma regra para tratamento dessa colisão na qual são preservados tanto o número de partículas quanto o vetor soma de suas velocidades, sendo considerada a preservação tanto da massa quanto da quantidade de movimento como mostra o tempo *t + 1*.

O $LGCA$ utiliza também variáveis booleanas $n_i(\vec{x}, t), i = 1, ..., \ell$, para descrever a ocupação das partículas nas células. Essas variáveis indicam se existe ($n_i = 1$) ou não existe ($n_i = 0$) uma partícula na posição $\vec{x}$, no instante de tempo t deslocando-se na direção de sua velocidade $\vec{e_i}$ (estas direções são definidas conforme o *Lattice*), e ℓ é o número de direções de movimento do *Lattice*. Sendo assim, tal equação de movimentação de partículas é definida conforme equação 4.1:

$$n_i(\vec{x} + \vec{e_i}, t + 1) = n_i(\vec{x}, t) + \Omega_i(\vec{n}(\vec{x}, t)), \qquad i = 1, ..., \ell \tag{4.1}$$

onde $\Omega = (\Omega_1, ..., \Omega_\ell)$ é chamado de operador de colisão que relaciona mais de uma partícula, sendo aplicado ao conjunto de partículas $\vec{n}(\vec{x}, t)$.

O *LGCA* trabalha apenas com valores discretos e por isso pode gerar resultados indesejáveis como por exemplo ruídos na simulação. O *LBM* apresenta modificações que se propõem a transpor esses defeitos.

Para migrar do *LGCA* para o *LBM*, ou seja, da microescala[3], para a mesoescala[4], deve-se considerar as médias adquiridas em cada distribuição de partículas, fazendo com que as distribuições variem suavemente no espaço e no tempo.

4.2.2 TEORIA DA EQUAÇÃO DE BOLTZMANN

A Teoria Cinética é uma área da Física Estatística que trata da dinâmica de processos fora do equilíbrio e sua convergência para o equilíbrio termodinâmico (CERCIGNANI, 1988). Nesse contexto, uma importante equação é a de *Boltzmann* que trata da evolução de distribuições de partículas.

Ainda que o *LBM* tenha sua origem histórica derivada do modelo discreto *LGCA* apresentado na seção anterior, ele pode também ser obtido através de uma derivação do modelo físico representado pela equação de transporte de *Boltzmann* baseada em uma expansão que utiliza número de Mach pequeno.

O número de Mach M é uma unidade de medida admensional, obtida através da razão entre o módulo de velocidade do escoamento do fluido e a velocidade do som $M = |\vec{u}|/c_s$, definindo a relação entre a velocidade do fluido e a velocidade do som, devendo a velocidade do fluido ser pequena em relação a do som.

A equação de *Boltzmann* pode ser descrita, sem levar em conta forças externas, da seguinte forma:

$$\frac{\partial g}{\partial t} + \vec{v} \cdot \nabla g = Q(g', g), \tag{4.2}$$

onde $g = g(\vec{x}, \vec{v}, t)$ é a função de distribuição das partículas no espaço de fase[5] contínuo $(\vec{x}, \vec{v})$, $\vec{x}$ é a posição espacial, $\vec{v}$ representa a velocidade das micro partículas, t é um

[3]Micro-partículas com valores discretos

[4]Distribuições de partículas com valores reais

[5]Espaço de fase é uma construção matemática a partir do espaço de todas as possíveis posições instantâneas de um sistema mecânico.

instante de tempo e $Q(g', g)$ é um termo integral de colisão. Nessa equação pode-se observar do lado esquerdo da igualdade a derivada material (ou derivada total em relação ao tempo) da quantidade g e do lado direito o termo de colisão entre distribuições de partículas com velocidades arbitrárias.

O termo $Q(g', g)$ é um termo não linear e torna a equação implícita. Em (GOLBERT, 2009b) o autor torna o termo $Q(g', g)$ da equação de *Boltzmann* 4.2 explícito, com a introdução da aproximação denominada *BGK* (BHATNAGAR, 1954), onde o operador integral de colisão foi linearizado em torno de uma distribuição de partículas, cumprindo o teorema H de *Boltzmann* e a conservação local de massa e momento através da função de distribuição de equilíbrio de *Maxwell-Boltzmann*, resultando em:

$$\frac{\partial g}{\partial t} + \vec{v} \cdot \nabla g = \varpi(g^M - g), \tag{4.3}$$

onde g^M é a função de distribuição de equilíbrio de *Maxwell-Boltzmann* (GOLBERT, 2009b).

No processo de derivação da equação de *Lattice Boltzmann* a partir da equação 4.2 (GOLBERT, 2009b), é feita a discretização dos domínios espacial (x), temporal (t) e de velocidades (v). Mais detalhadamente, o dominio espacial é discretizado de forma regular e uniforme com espaçamento dx, o espaçamento temporal é definido por dt e as velocidades passam a ser representados por um conjunto finito de l direções. Seguindo o processo detalhado em (GOLBERT, 2009b), se chega à equação de *Lattice Boltzmann*, a qual será estudada em detalhe na seção 4.4.1.

4.3 MODELOS DE LATTICE

Nesta seção serão apresentados diversos modelos de *Lattice*, dentre os quais um deles será utilizado para a implementação deste trabalho.

Um *Lattice*, também chamado de malha ou rede, pode ser definido como um reticulado, em que cada nó representa uma possível direção de movimento que pode ser atingida por uma distribuições de micro-partículas, como ilustra a figura 4.6. As distribuições de micro-partículas se movimentam pelo reticulado de um nó para outro, seguindo as direções de movimento que interligam os vizinhos. A quantidade de direções de movimento e a dimensão do reticulado são configuradas de acordo com alguns modelos e todos os nós do reticulado têm a mesma configuração.

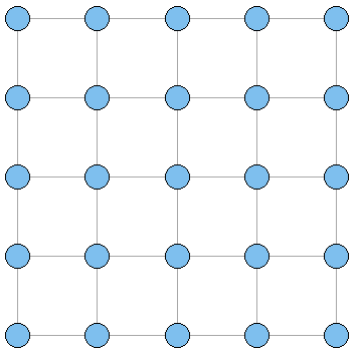


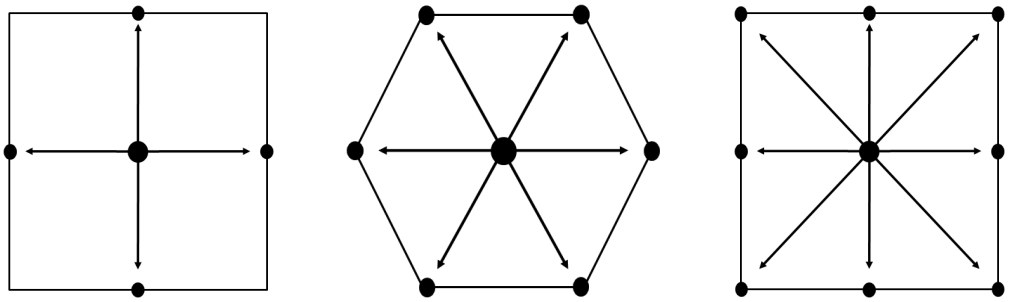
FIG. 4.6: Modelo de reticulado.

O nome de um *Lattice* é composto pelo número de dimensões que ele possui, e o número de direções de movimento que ele pode atingir. Deste modo o nome segue o modelo $DXQY$, no qual X indica o número de dimensões e Y indica as direções de movimento, incluindo o ponto estático.

O ponto estático representa a possibilidade de não movimentação de uma distribuição de partículas, ou seja quando a velocidade é nula.

4.3.1 *LATTICES* COM 2 DIMENSÕES

Estes modelos apresentam apenas movimentações em duas dimensões do *Lattice*, tomando como exemplo (X,Y). São três os modelos mais utilizados: *D2Q5* que possui 4 direções de movimento além do ponto estático, ilustrado na figura 4.7(a), *D2Q7* com 6 direções de movimento compreendendo também o ponto estático como ilustrado na figura 4.7(b), e *D2Q9* com 8 direções de movimento incluindo também a possibilidade do ponto estático, conforme ilustrado na figura 4.7(c).

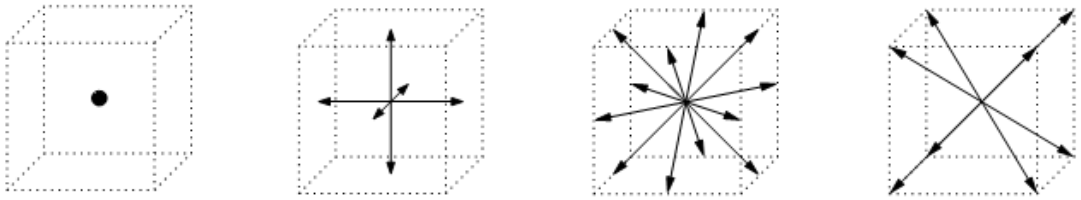


(a) Modelo *D2Q5* (b) Modelo *D2Q7* (c) Modelo *D2Q9*

FIG. 4.7: Modelos de *Lattices* de duas dimensões.

4.3.2 *LATTICES* COM 3 DIMENSÕES

Por conter muitas direções de movimento, *Lattices* tridimensionais podem ser melhor interpretados quando separados em sub-grupos de acordo com suas características. Tomando como exemplo, na figura 4.8(a) é ilustrado o sub-grupo que representa o ponto estático, na figura 4.8(b) o sub-grupo contendo as direções de movimento que tem suas direções apontadas para as faces do cubo, na figura 4.8(c) o sub-grupo com as direções apontadas para as arestas e na figura 4.8(d) o sub-grupo com as direções apontadas para os vértices.



(a) Ponto estático. (b) Faces (c) Arestas (d) Vértices

FIG. 4.8: Direções de movimento separadas em conjuntos. (SCHEPKE, 2007)

Segundo a figura 4.8, combinações entre as sub-figuras (a), (b), (c) e (d) revelam 3 modelos de *Lattices* tridimensionais. A união de (a), (b), e (d) dá procedência ao modelo *D3Q15* que contém 1 ponto estático, 6 faces e 8 vértices. A composição de (a), (b), e (c) dá forma ao modelo *D3Q19* com um ponto estático, 6 faces e 12 arestas. A associação entre (a), (b), (c), e (d) dá origem ao modelo de *D3Q27*, que detém o maior número de direções de movimento possíveis, 1 ponto estático, 6 faces, 12 arestas e 8 vértices.

O modelo tridimensional mais utilizado pela literatura é o *D3Q19*, por possuir maior quantidade de movimentos do que o modelo *D3Q15* e menor custo computacional do que o modelo *D3Q27* (GOLBERT, 2009b). Pelo seu custo benefício costuma ser adotado em trabalhos envolvendo a hemodinâmica computacional (GOLBERT, 2009b), (ARTOLI, 2006), (HE, 2009), (PELLICCIONI, 2007), (KRAFCZYK, 1998), e (HIRABAYASHI, 2006). Tal modelo pode ser visto na figura 4.9.

4.4 MÉTODO DE LATTICE BOLTZMANN

O LBM pode ser descrito como um método numérico baseado em equações cinéticas formuladas em uma escala mesoscópica simulando a dinâmica de fluidos em uma escala macroscópica (CHEN, 1998), localizando-se entre os mundos microscópico e macroscópico, absorvendo parte dos dois (GOLBERT, 2009b). A abstração do modelo molecular para

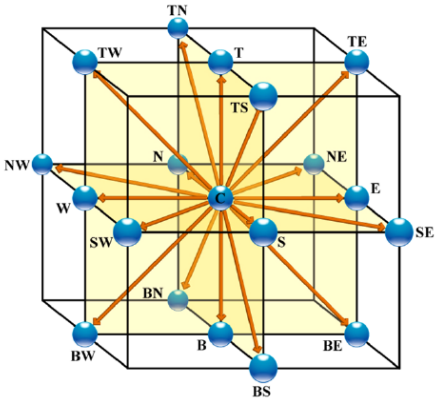


FIG. 4.9: Modelo de *Lattice D3Q19* com as direções de movimento.

o modelo mesoscópico com propriedades macroscópicas pode ser vista na figura 4.10.

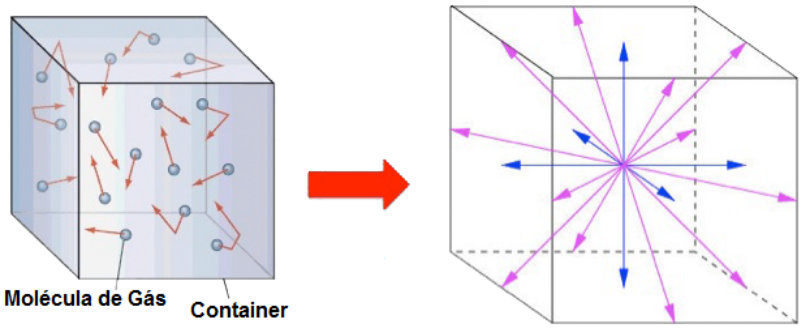


FIG. 4.10: Discretização do modelo molecular para o *LBM* (SUCCI, 2010)

Devido ao seu grande potencial para simular fluxos turbulentos, fluxos em múltiplas fases e fluxos com condições de contorno irregulares, o *LBM* é frequentemente utilizado como um modelo alternativo para a modelagem e simulação computacional da dinâmica de fluidos (SUCCI, 1997).

Criado a partir da derivação da equação de *Boltzmann* e baseado no método de *LGCA*, o *LBM* utiliza estruturas denominadas *Lattice* ou malha, assim como o *LGCA*, que podem ser bidimensionais ou tridimensionais.

4.4.1 EQUAÇÃO DE LATTICE BOLTZMANN

A equação de *Lattice Boltzmann* é descrita por:

$$f_i(\vec{x} + \Delta x\vec{e_i}, t + \Delta t) = f_i(\vec{x}, t) + \Omega_i(\vec{f}(\vec{x}, t)), \qquad i = 1, ..., \ell \tag{4.4}$$

onde, f_i representa a função de distribuição de partículas e $\Omega_i(\vec{f}(\vec{x},t))$ é o termo de colisão (que relaciona um conjunto de distribuições de partículas f_i, denotado por $\vec{f}(\vec{x},t)$), analisados na direção $\vec{e_i}$, o espaçamento do *Lattice* é representado por Δx, o passo no tempo por Δt e o número de direções de movimento do *Lattice* por ℓ.

Na equação 4.4 pode-se perceber que o *LBM* substitui as variáveis booleanas de ocupação de partículas n_i , da equação 4.1 de movimentação de partículas do *LGCA*, por funções de distribuição de partículas reais $f_i = \langle n_i \rangle$, onde $\langle \cdot \rangle$ leva em conta as médias das distribuições, ignorando a movimentação de partículas individuais. Com isso, o *LBM* elimina resultados indesejáveis como por exemplo os ruídos presentes no *LGCA*, assumindo as médias dos valores de n_i fazendo com que as distribuições variem suavemente no espaço e no tempo.

A conservação de massa e do momento é obtida a partir do somatório das variáveis mesoscópicas do fluido. Considera-se ρ (massa específica) e a quantidade de movimento do fluido $\rho\vec{u}$ (momento) através das equações:

$$\rho = \sum_{i=0}^{l} f_i \tag{4.5}$$

$$\rho\vec{u} = \sum_{i=1}^{l} f_i \vec{e_i} v, \tag{4.6}$$

onde $v = \frac{\Delta x}{\Delta t}$.

4.4.2 APROXIMAÇÃO BGK

Para fazer a substituição do termo de colisão da equação de *Lattice Boltzmann* 4.4, uma aproximação utilizada é a *BGK* (BHATNAGAR, 1954), que utiliza um termo de relaxação cumprindo a conservação local de massa e momento.

$$f_i(\vec{x} + \Delta x\vec{e_i}, t + \Delta t) = f_i(\vec{x}, t) + \frac{1}{\tau}[f_i^{eq}(\vec{x}, t) - f_i(\vec{x}, t)], \qquad i = 0, ..., \ell \tag{4.7}$$

onde é importante ressaltar que o termo $f_i^e q(\vec{x}, t)$, é uma distribuição de equilíbrio dependente da massa específica ρ e do vetor de velocidades $\vec{u}$ das distribuições envolvidas, controlando a taxa de aproximação do equilíbrio para simulações que envolvem a dinâmica de fluidos incompressíveis. O termo de relaxamento τ é definido com base nos parâmetros

físicos do fluido e de parâmetros específicos do próprio *Lattice*. Segundo (GOLBERT, 2009b) o valor mínimo de τ é de 0,5 sendo valores abaixo deste limiar considerados negativos, o que estaria em desacordo com as leis da termodinâmica.

Na equação 4.8, a viscosidade cinemática ν do fluido modelado depende tanto dos parâmetros que definem a discretização espaço-temporal como do parâmetro de relaxação que caracteriza o operador de colisão.

$$\nu = \frac{(2\tau - 1)}{6}\frac{\Delta x^2}{\Delta t} \tag{4.8}$$

A equação 4.7 pode ser dividida em duas etapas: colisão e propagação, como proposto pela implementação de (GOLBERT, 2009b) sem perder a característica da equação, de maneira a facilitar a implementação. A colisão segue a forma da equação 4.9 e a equação de propagação a forma de 4.10.

$$f_i(\vec{x}, t)^* = f_i(\vec{x}, t) + \frac{1}{\tau}[f_i^{eq}(\vec{x}, t) - f_i(\vec{x}, t)], \qquad i = 0, ..., \ell \tag{4.9}$$

$$f_i(\vec{x} + \Delta x\vec{e_i}, t + \Delta t) = f_i(\vec{x}, t)^*, \qquad i = 0, ..., \ell \tag{4.10}$$

Nesse modelo denominado *Lattice BGK* (*LBGK*), a distribuição de equilíbrio local foi escolhida de forma que se possam recuperar as *ENS* (QIAN, 1992) e (CHEN, 1992). Ao se desenvolver uma equação cinética simplificada, evita-se resolver equações cinéticas complexas como a equação de *Boltzmann* na sua forma mais completa (GOLBERT, 2009b), além de não ser preciso acompanhar cada uma das partículas como acontece em simulações que envolvem a dinâmica molecular (CHEN, 1998).

4.4.3 DISTRIBUIÇÃO DE EQUILÍBRIO INCOMPRESSÍVEL

Uma das versões da distribuição de equilíbrio (f_i^{eq}) descrita em (HE, 1997), controla a taxa de aproximação do equilíbrio para simulações que envolvem a dinâmica de fluidos incompressíveis. Como pretende-se simular tal dinâmica, é vantajoso escolher uma distribuição de equilíbrio que consiga reduzir possíveis erros de compressibilidade ligados ao

LBM. A relaxação da aproximação *BGK* cumpre ainda a conservação local de massa e momento e possui a forma:

$$f_i^{eq} = \omega_i \left\{ \rho + \rho_0 \left[3\frac{(v\vec{e_i}.\vec{u})}{v^2} + \frac{9}{2}\frac{(v\vec{e_i}.\vec{u})^2}{v^4} - \frac{3}{2}\frac{(\vec{u}.\vec{u})}{v^2} \right] \right\} \tag{4.11}$$

onde $i = 0, ..., l$ e $v = \frac{\Delta x}{\Delta t}$ é a velocidade da partícula, os pesos w_i são dependentes do *Lattice* e para o modelo de *Lattice* utilizado nesse trabalho (*D3Q19*), os pesos podem ser vistos na equação 4.12; ρ_0 é a massa específica média, ρ é a massa específica, u a velocidade do fluido e ν a viscosidade cinemática descrita pela equação 4.8.

$$\omega_0 = \frac{1}{9} \qquad \omega_{1-6} = \frac{1}{18} \qquad \omega_{7-18} = \frac{1}{36} \tag{4.12}$$

Com o uso desta distribuição de equilíbrio pode-se recuperar as *ENS* incompressíveis, com ordens de aproximação em termos do número de Mach (GOLBERT, 2009b).

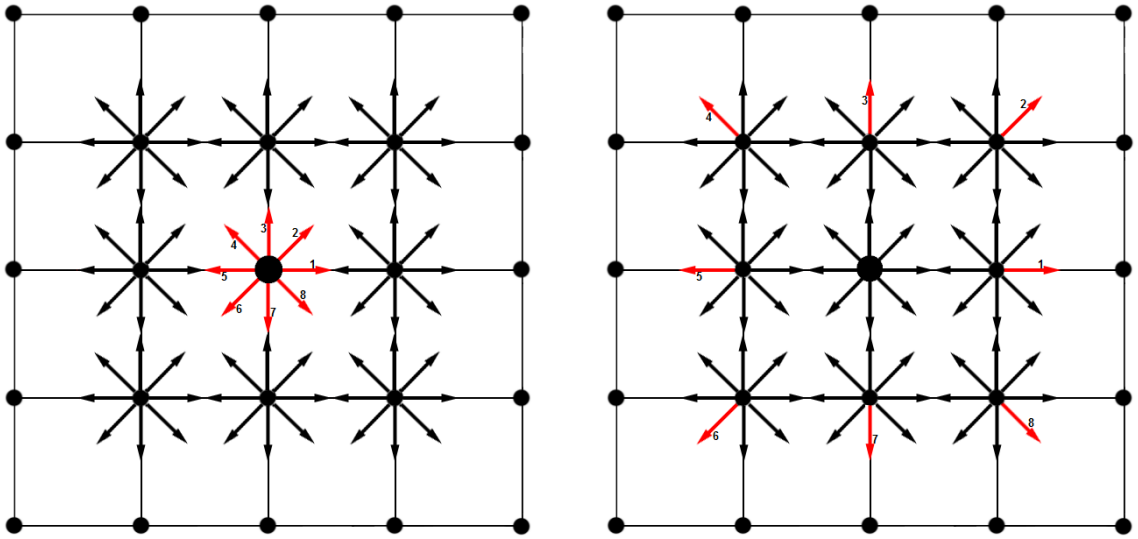
4.4.4 PROPAGAÇÃO

A etapa de propagação representa a movimentação das distribuições de partículas até os nós vizinhos, nas direções indicadas $\vec{e_i}$ conforme a equação 4.10. A propagação consiste em mover cada distribuição de partículas para o vizinho no qual a direção de movimento da distribuição está apontando.

Para facilitar o entendimento, a figura 4.11 ilustra colisão e propagação apenas de um nó localizado no centro do *Lattice*, mas ambos procedimentos ocorrem de forma análoga em todos os nós.

A figura 4.11(a) ilustra a colisão ocorrida entre as distribuições de partículas, nas quais cada distribuição destacada em vermelho tem sua direção de movimento apontada para um nó vizinho. A figura 4.11(b) ilustra o destino das distribuições que colidiram-se em 4.11(a), quando forem propagadas para os nós vizinhos. É possível notar que este procedimento pode sobrescrever a distribuição de um vizinho, caso a mesma não tenha sido propagada. Em contrapartida, se as distribuições dos vizinhos tiverem sido propagadas, as distribuições do nó em questão foram sobrescritas. A sobrescrita é um problema em ambos os casos pois acabam por gerar inconsistências na simulação.

Embora diversos autores dupliquem a malha para evitar a sobrescrita, isto implica em utilizar o dobro de memória e aumentar o tempo de processamento. De forma mais



(a) Colisão (b) Propagação

FIG. 4.11: Problema da sobrescrita causado pela propagação.

otimizada, uma técnica proposta por (MATTILA, 2007) executa simples trocas entre posições de memória e utiliza uma única malha. Essa técnica divide a propagação em duas partes, aumentando o processamento em contrapartida a duplicação do uso de memória.

A primeira parte efetua trocas internas entre as distribuições e a segunda parte realiza trocas externas com os vizinhos, sendo esta dependente da completa execução da primeira parte.

Antes de efetuar as trocas internas, as distribuições estão dispostas como ilustrado na figura 4.12. As distribuições opostas foram destacadas em cores diferentes (vermelho e verde), visando facilitar a identificação das trocas que serão realizadas no decorrer da execução das duas partes envolvidas na propagação.

Para descrever tal técnica, foi utilizada apenas uma face do *Lattice*, omitindo as demais distribuições do modelo *D3Q19* para facilitar a compreensão sem perda de generalidade.

Na primeira parte da propagação conforme mostra a figura 4.13, são feitas trocas entre distribuições opostas internas ao nó, trocando todas as distribuições em verde pelas distribuições opostas em vermelho. Como está sendo feita a troca entre posições opostas, apenas metade das distribuições precisa ser acessada para que todas as trocas internas sejam feitas. Tal procedimento é repetido para todos os nós do *Lattice*, e deve ser completado antes da execução da segunda parte da propagação. Isso garante consistência na execução da segunda parte, que irá utilizar os nós vizinhos para executar as trocas externas, assegurando que os vizinhos já tenham feito as suas trocas internas.

Na segunda parte da propagação, cada nó efetua metade das trocas de suas dis-

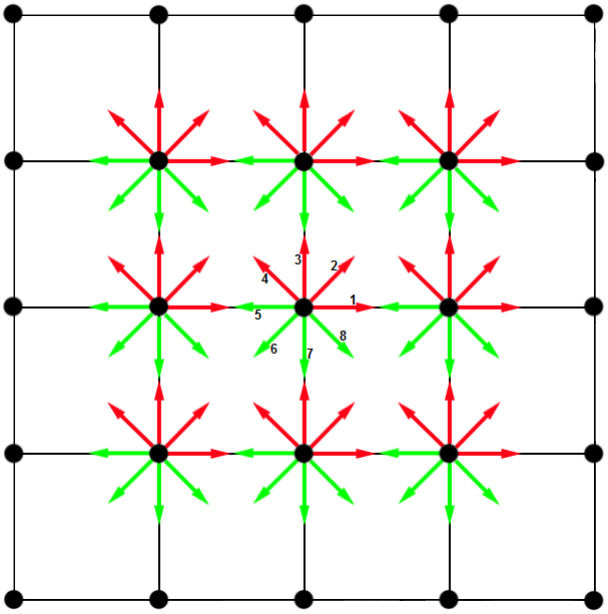


FIG. 4.12: Distribuições antes do passo da propagação.

tribuições com metade dos seus vizinhos, deixando que a outra metade das distribuições sejam trocadas pela outra metade dos vizinhos. Desta maneira não haverá sobrescrita de dados nem dependência entre os nós para efetuar as trocas, o que torna a técnica interessante para execução paralela.

Os nós encontravam-se com todos as suas distribuições invertidas na figura 4.13. A troca é feita de forma análoga a da primeira parte da propagação, mas desta vez a distribuição oposta se localiza no vizinho oposto, e ambos são acessados pelo mesmo índice. A figura 4.14 ilustra as trocas que foram executadas com os vizinhos pelo nó central.

Utilizando como comparação a figura 4.13 e a figura 4.14, o nó central executa as trocas de suas distribuições 1, 2, 3 e 4 com as distribuições 5, 6, 7 e 8 dos nós vizinhos. Utiliza-se o deslocamento de forma similar a utilizada na primeira parte da propagação, sendo que o deslocamento é também utilizado para acessar o nó vizinho correspondente.

Para que o nó central tenha todas as suas trocas finalizadas, os nós 1, 2, 3 e 4 precisam executar as suas trocas externas. Cada um desses nós utiliza o mesmo procedimento executado pelo nó central, e quando tais nós fazem suas trocas utilizam o nó central como vizinho. Quando terminarem as suas trocas externas, o nó central terá todas as novas distribuições posicionadas, como pode ser visto na figura 4.15.

Após as trocas serem finalizadas em todos os nós, nenhum dado foi sobre-escrito e o *Lattice* fica exatamente como antes da propagação na figura 4.12 já exposta, só que com todas as distribuições propagadas.

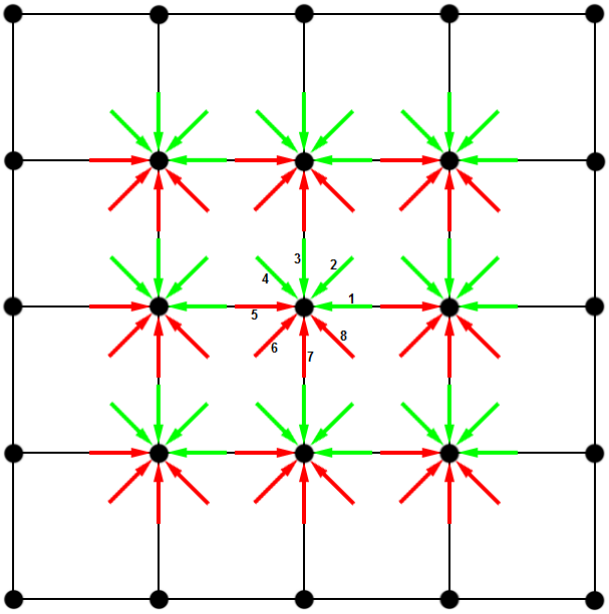


FIG. 4.13: Distribuições depois das trocas internas.

Após o término do passo da propagação, os valores macroscópicos de velocidade e massa específica são atualizados em todos os nós e o *Lattice* está pronto para retornar ao início do loop do *LBM*.

4.4.5 CONDIÇÃO DE CONTORNO DE VELOCIDADE

A aplicação das condições de contorno é um fator importante para o *LBM*, pois elas podem exercer influência nos erros ao realizar as simulações computacionais (GOLBERT, 2009b).

Mesmo as condições de contorno do *LBM* não possuindo relação macroscópica, ao se definir regras microscópicas adequadas pode-se atingir a macroestrutura. Existem diversas implementações para soluções de condição de contorno de velocidade. Em (GOLBERT, 2009b) o autor detalha as principais implementações encontradas na literatura, fazendo uma comparação entre elas, mas somente o modelo escolhido será exposto no trabalho.

O modelo de esquema de interpolação escolhido foi proposto por (GUO, 2002). Traz vantagens sobre os demais modelos, pois pode-se empregar as condições de contorno fora do centro do nó do *Lattice*, atingindo maior precisão. Ele ainda pode ser empregado em linhas curvas mais precisamente do que os modelos anteriores, pois as curvas não necessitam estar posicionadas sobre o centro dos nós do *Lattice*.

A implementação desse modelo é mais complexa devido à necessidade de identificação dos pontos em que as linhas de contorno se cruzam com as conexões entre os nós do

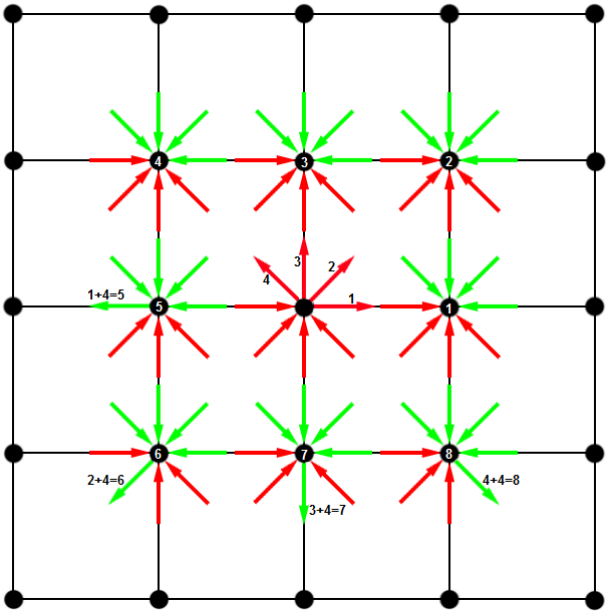


FIG. 4.14: Distribuições após trocas com os vizinhos 5, 6, 7 e 8 executadas pelo nó central.

Lattice. Porém em um *Lattice* regular, como é o caso do *LBM*, essa identificação pode ser considerada de pouca relevância devido à regularidade no posicionamento dos nós e da quantidade finita e definida de vizinhos. Como forma de simplificar sua implementação, considera-se que as linhas de contorno passam exatamente sobre os nós do *Lattice*.

O modelo se baseia em estimar as f_i's dos nós de contorno também chamados de nós de parede, separando em partes de equilíbrio (f_i^{eq}) e de não-equilíbrio ($f_i - f_i^{eq}$). A aplicação das estimativas são feitas para cada direção i de um nó de parede em que haja um nó de fluido vizinho.

A parte de equilíbrio obtida pela equação 4.11 já exposta é estimada com base na velocidade e na massa específica na célula de parede, que são calculadas através da interpolação das velocidades $\vec{u}$ e massas específicas ρ dos nós de fluido próximos conforme equação 4.13:

$$f_i^{eqP}(\rho^F(x_{f_i}), \vec{u}) \qquad i = 0, ..., \ell \tag{4.13}$$

onde é calculada a distribuição de equilíbrio do nó de parede (f_i^{eqP}) através da massa específica do nó de fluido vizinho ρ^F na posição x_{f_i} e velocidade a ser imposta $\vec{u}$.

A parte de não-equilíbrio é copiada do nó de fluido mais próximo, conforme equação 4.14.

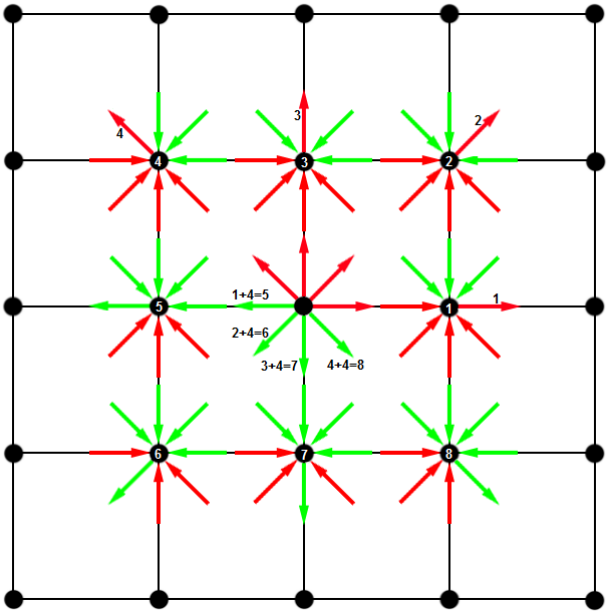


FIG. 4.15: Distribuições após trocas executadas pelos nós 1, 2, 3 e 4 com o nó central.

$$f_i^{NeqP} = f_i^{NeqF} = f_i^F(x_{f_i}) - f_i^{eqF}(x_{f_i}) \qquad i = 0, ..., \ell \tag{4.14}$$

onde a distribuição de não equilíbrio do nó de fluido próximo (f_i^{NeqF}) é obtida pela diferença entre a sua distribuição das partículas $f_i^F(x_{f_i})$ e sua distribuição de equilíbrio $f_i^{eqF}(x_{f_i})$.

Com isso pode-se calcular a nova distribuição de partículas que saem do nó de parede, conforme mostra a equação 4.15:

$$f_i^P = f_i^{eqP} + f_i^{NeqP} \qquad i = 0, ..., \ell \tag{4.15}$$

onde a nova distribuição de partículas f_i^P recebe a imposição da condição de contorno de velocidade.

5 TRABALHOS RELACIONADOS

A simulação computacional da dinâmica de fluidos é frequentemente descrita pelas equações de *Navier Stokes* (*ENS*) (SCHEPKE, 2007) (LI, 2003). Tais equações utilizam apenas as propriedades físicas macroscópicas do fluido para descrevê-lo. Deste modo, as *ENS* se tornam insensíveis à dinâmica microscópica, uma vez que os valores resultantes dos processos microscópicos permanecem constantes durante a simulação.

Por outro lado, abordagens microscópicas se tornam por vezes inviáveis em razão das escalas de espaço e tempo muito pequenas, bem distantes das encontradas em aplicações práticas (WOLF, 2006).

O método de *Lattice Boltzmann* (*LBM*) é uma técnica numérica, que trabalha em uma escala mesoscópica, podendo resolver as limitações do modelo macroscópico e a alta definição do modelo microscópico (WOLF, 2006). Suas equações descrevem fisicamente as propriedades em função de distribuições de partículas, ao invés de cada partícula individualmente.

O *LBM* tem sido empregado nas duas últimas décadas como meio alternativo para simular a dinâmica de fluidos computacional (CHEN, 1992), (CHEN, 1998), (GUO, 2008) e (KUZNIK, 2010). Ele tem como características positivas uma menor complexidade de implementação computacional, alto potencial de paralelização e escalabilidade em comparação com demais métodos clássicos da simulação de fluidos computacional (GOLBERT, 2009b). Entretanto suas características negativas apontam diretamente para o alto consumo de processamento e utilização de memória, reconhecidas literariamente em (GUO, 2009), (GOLBERT, 2009b) e (KUZNIK, 2010).

Um método computacional trabalhado sobre medida para um *hardware* específico pode ter grandes benefícios em termos de escalabilidade e ganho de desempenho segundo estudo feito por (KUZNIK, 2010).

O *LBM* pode ser utilizado em diversas aplicações envolvendo a dinâmica dos fluidos, e tem sido empregado de forma sistemática em diversas aplicações envolvendo a hemodinâmica computacional como, por exemplo: na simulação de escoamentos sanguíneos em artérias cerebrais (HE, 2009), em válvulas cardíacas mecânicas (PELLICCIONI, 2007) e (KRAFCZYK, 1998), em aneurismas cerebrais (HIRABAYASHI, 2006), e na aorta abdominal (ARTOLI, 2006).

Foram utilizadas técnicas de simulação numérica e de visualização científica no trabalho de (COLLARES, 2011). A simulação numérica apresentada pelos autores é parte desta dissertação, onde foram implementadas três versões do *LBM*, sendo uma versão sequencial e duas versões paralelas sem otimização de memória. A visualização científica exposta em tal trabalho apresentou uma técnica denominada de traçado de partículas, que permitiu a visualização das trajetórias de pequenas massas de partículas. Os autores fizeram comparações entre implementações seriais e paralelas, expondo o ganho de desempenho obtido pelas implementações paralelas.

O cálculo de trajetória de partículas *CTP* também foi alvo do trabalho apresentado em (CAMARGO, 2011), no qual o autor discutiu uma ferramenta que utiliza *GPU* para o processamento do *CTP*. Ainda foram executadas comparações com diferentes metodologias de alocação de blocos e *threads* na *GPU*.

No trabalho (GOLBERT, 2009a) o autor implementa o *LBM* com objetivo de simular o fluxo sanguíneo nas principais artérias do corpo humano simulando artificialmente o fluxo sanguíneo em um aneurisma cerebral.

No trabalho (GUO, 2008) os autores concluem que para atingir o processamento de alto desempenho ou *High Performance Computing* (*HPC*) é importante buscar a aceleração do algoritmo por meio de técnicas de otimização e paralelização. Neste artigo, os autores analisam diversas implementações do *LBM* e definem algumas métricas que, segundo eles, são importantes para alcançar um melhor desempenho. Utilizando tais métricas, organizaram as implementações em quatro diferentes grupos: (i) refinamento eficiente do *Lattice*, (ii) paralelização do algoritmo, (iii) otimização de cache e (iv) implementação baseada em *GPU*.

O primeiro grupo, refinamento eficiente do *Lattice*, é descrito pelo autor como uma técnica que divide o domínio computacional em sub-domínios, ajustando cada um deles com espaçamento e passo de tempo de acordo com a precisão exigida em cada região. *Lattices* mais refinados em regiões onde a variação da movimentação ocorre de forma mais dinâmica, e *Lattices* menos refinados nas demais regiões onde a variação do movimento é mais suave.

Uma implementação baseada em refinamento local do *Lattice* foi implementado por (ZHAO, 2007), no qual o *Lattice* é dividido em partes e o tamanho de cada parte pode variar dependendo da necessidade de refinamento de cada região. A utilização de múltiplos *Lattices* ocasiona procedimentos computacionais adicionais para garantir a sincronia no processamento, como por exemplo na interpolação temporal e espacial. Adicionalmente,

é necessário uma multiplicidade entre a quantidade de elementos dos *Lattices* (refinados e sem refinamento) para também garantir a correta sincronização das suas execuções.

Referente ao segundo grupo, paralelização do algoritmo, segundo os autores, os esforços são comumente direcionados à estratégias apropriadas de balanceamento de carga, visando minimizar custos de sobrecarga de comunicação oriundas da latência da rede e de largura de banda, e sobrecarga de processamento oriunda de má divisão dos dados para processamento. Dentre algumas técnicas de decomposição de domínio, em (DUPUIS, 2002) o autor propôs representar um *Lattice* de 2 dimensões na forma de um vetor de células no qual cada célula conhece os índices dos seus vizinhos no *Lattice 2D*. Em (WANG, 2005) o autor decompõe o domínio em subdomínios e divide as células entre os subdomínios para executar o balanceamento de carga. Já em (YU, 2006), o autor divide o domínio em blocos e cada bloco é designado a um processador em paralelo, sendo necessário somente que os dados de borda sejam compartilhados entre os processadores ou no caso de um *cluster*, transportados pela rede.

O terceiro grupo, segundo os autores, consiste em aplicar otimizações de *cache* no sentido de evitar acessos demasiados à memória principal (que são mais lentos do que acessos a memória cache) durante o processamento. Uma técnica que se encaixa nesse contexto é a decomposição do domínio em sub-domínios até que as partes caibam na memória *cache* para, deste modo, serem utilizados respectivamente.

Um algoritmo que melhora ainda mais o desempenho do *LBM* foi introduzido por (MATTILA, 2007), uma que vez os dados estejam carregados em *cache* o algoritmo baseia-se em simples trocas de posições de memória entre as distribuições de partículas.

No trabalho (WELLEIN, 2006), o autor concluiu que a escolha correta da configuração dos dados da simulação, em conjunto com a escolha correta dos parâmetros a serem carregados em cache, são fundamentais para alcançar o alto desempenho do *LBM*.

No quarto e último grupo, implementação baseada em *GPU*, os algoritmos precisam tirar proveito da arquitetura *SIMD* das placas gráficas para executar as inúmeras computações exigidas pelo *LBM*, afirmam os autores.

A importância de se utilizar as memórias de menor latência para a implementação do *LBM* baseado em *GPU* é ressaltada em (KUZNIK, 2010). O autor destaca também a relevância da utilização de acessos coalescentes que, segundo ele, podem ser alcançados separando-se os tipos de nós do *Lattice* em grupos diferentes, conforme suas densidades. O autor confronta duas implementações do *LBM* com números em ponto flutuante, uma com precisão simples, e outra com precisão dupla, inferindo que o tempo computacional

do algoritmo com precisão simples é 3.8 vezes menor. Finalmente, o autor aponta a implementação em *GPU* como uma convincente alternativa para implementação do algoritmo do *LBM*, por possuir maior facilidade de manutenção e menor custo se comparada com um *cluster* de *UCPs*.

Pode-se ver como as atuais técnicas de computação de alto desempenho, são designadas para utilização em aplicações médicas em (MITTMANN, 2009). Segundo o autor, aplicações que demandam uma grande quantidade de cálculos matemáticos de natureza vetorial podem fazer uso das modernas unidades de processamento gráfico (*GPU*) para melhorar seu desempenho.

São comparados o desempenho de três aplicações naturalmente paralelas e que demandam grande poder computacional em (CHE, 2008), no qual o autor utiliza implementações em UCP e *GPU* como, simulação de tráfego, simulação térmica e algoritmo de k-médias. Os autores demonstram que muitas outras aplicações paralelizáveis podem utilizar o poder computacional das *GPUs*, podendo atingir ganhos de desempenho dezenas de vezes superiores às suas equivalentes implementações em UCP.

No trabalho desenvolvido por (GASPAROTO, 2009), o autor indica que nem todos os procedimentos de um processamento são relevantes o suficiente que justifique o processamento em *GPU*. Conforme simulação apresentada em seu trabalho, apenas os procedimentos que necessitam de processamento mais intenso devem ser levados a *GPU*.

No que tange ao volume de dados, nos experimentos feitos em (LABAKI, 2009) são comparadas implementações em UCP e *GPU*, o autor apresenta gráficos que justificam a utilização de *GPU* apenas a partir de um determinado volume de dados. Isso acontece porque implementações em *GPU* exigem alguns passos extras para seu processamento, que não são necessários para a UCP. O autor executou experimentos aumentando o volume de dados e mostrando que após um ponto de intersecção entre os tempos de processamento das simulações, a *GPU* atingiu um ganho de 71,3 vezes sobre a UCP.

O *LBM* é implementado com divisão em *multi-lattice* em (ZHAO, 2007), combinando duas das métricas apontadas em (GUO, 2008): refinamento eficiente do *Lattice* e implementação baseada em *GPU*. O autor implementa *kernels* diferentes para lidar com os diferentes refinamentos utilizados no *Lattice*, e segundo ele o algoritmo pode facilmente trabalhar com tantos *Lattices* refinados quantos forem necessários para o detalhamento das regiões de interesse.

Em trabalhos mais recentes como (OBRECHT, 2011) e (XIAN, 2011), o processamento do *LBM* foi direcionado a múltiplas *GPUs*.

Em (OBRECHT, 2011) os autores utilizaram seis *GPUs Tesla C1060*, dividiram o domínio em seis partes e designaram cada parte a uma GPU, nas quais a comunicação foi feita utilizando *MPI*. Dentro dos sub-domínios foi lançado um *kernel* para cada passo de tempo e a sincronização foi feita utilizando barreiras no padrão *POSIX*. Os autores concluíram que o desempenho da aplicação é quase ideal em comparação com implementações em supercomputadores ou grades de computadores, sugerindo estudos adicionais para melhorar a compreensão sobre a comunicação entre as *GPUs* e projetar uma decomposição de domínio mais compatível para a paralelização utilizando *MPI*.

Em (XIAN, 2011) os autores utilizaram *MPI* para comunicação entre os nós de um *cluster*, concluindo que a desvantagem para a utilização de múltiplas GPUs é que as GPUs não podem controlar o tráfego de dados entre elas, a comunicação deve ser feita através das UCPs o que aumenta o tempo da simulação.

6 PROPOSTA DE IMPLEMENTAÇÃO DO LBM

A simulação do *LBM* implementada neste trabalho requer como entrada uma malha de nós. Essa malha é processada utilizando as equações apresentadas na seção 4.4.1 e uma outra malha de saída é retornada como resultado do processamento.

A construção de uma malha consiste em transformar um domínio físico, contínuo, num domínio computacional discreto de pontos uniformemente distribuídos em um espaço cartesiano.

Foi utilizado o módulo de processamento de imagens do *HeMoLab*, apresentado na seção 2.1, para a geração da malha de entrada, na qual é salva em um arquivo no formato *VTK*. A malha não sofre nenhum refinamento descrito pelo módulo *3D* do *HeMoLab* na seção 2.1, mantendo a regularidade no espaçamento entre os nós.[6]

A medida que os passos do *LBM* são executados, os vetores de velocidade dos nós são atualizados. Ao final de uma iteração tem-se o estado atual da movimentação ocorrida até a presente iteração. Ao final da simulação, tem-se o estado final de toda movimentação.

Para a escrita da malha de saída são agregados valores escalares aos nós, provenientes de seus vetores de velocidade. A utilização de valores escalares permite que softwares especializados, como o *HeMoLab/ParaView*, seção 2.1, façam uma visualização da malha, apresentando de forma apropriada o estado final da movimentação de todas as porções de fluido.

Tanto a malha de entrada quanto a de saída são armazenadas em arquivos no formato *VTK*, apresentado no final da seção 2.1. A malha de entrada contém informações de coordenadas e espaçamento entre os nós e a malha de saída possui, adicionalmente, valores escalares de velocidade agregados para os nós.

Dentre os diversos modelos de *Lattice* apresentados na seção 4.3, o modelo escolhido foi o *D3Q19*, utilizado em diversos trabalhos envolvendo a hemodinâmica computacional, como exposto ao final da seção 4.3.

6.1 CONFIGURAÇÃO INICIAL PARA EXECUÇÃO DO LBM

Nesta seção, será descrito o passo inicial do *LBM*, representado por (A) na figura 6.1.

[6]No processo de refinamento são adicionados pontos na malha visando diminuir o serrilhado, que desfigura o espaçamento regular entre os pontos.

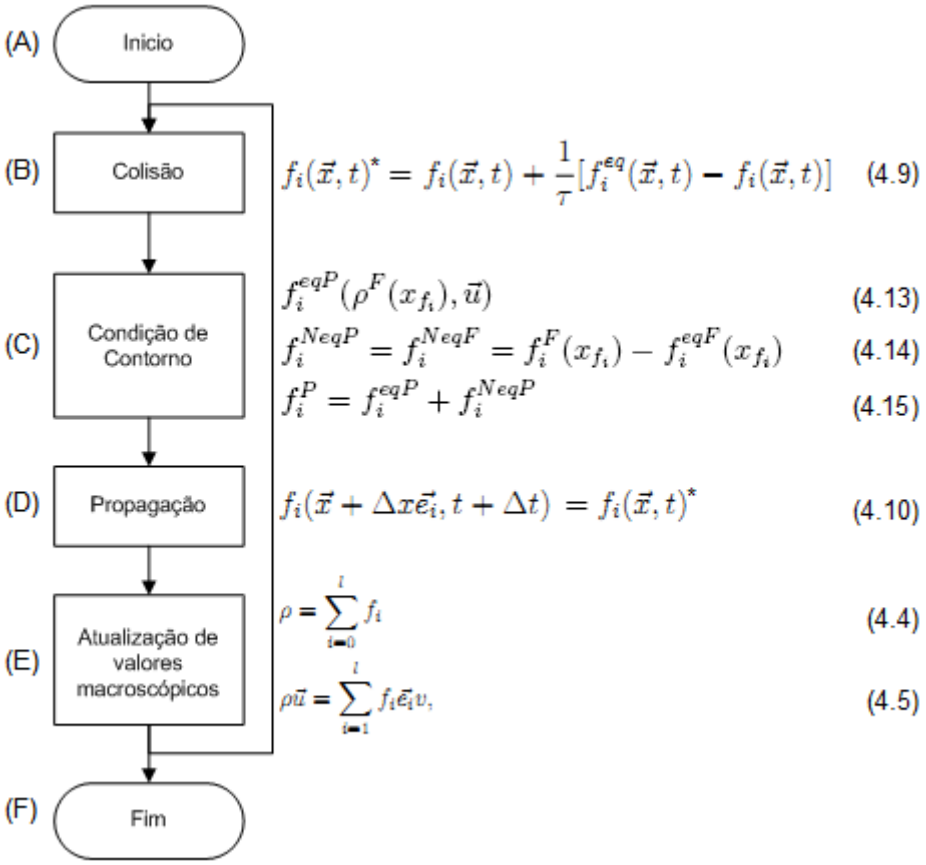


FIG. 6.1: Estrutura do algoritmo do *LBM*, com associação entre os passos da implementação e as equações do *LBM*.

A configuração inicial é formada por quatro procedimentos: a) leitura da malha; b) preenchimento dos nós com dados discretos e espaciais; c) configuração do esquema de vizinhança adotado pelo modelo de *Lattice* escolhido; d) identificação dos nós de acordo com as suas densidades. Esses procedimentos são todos executados em UCP e são comuns tanto para implementação sequencial quanto para a implementação paralela.

a) A leitura da malha de entrada é feita a partir de um arquivo VTK, onde estão armazenadas as coordenadas de todos os nós. Cada nó irá armazenar uma estrutura computacional chamada de *Node*, que é alocada e posicionada sobre o nó. As estruturas são alocadas em uma matriz tridimensional, mantendo de forma abstrata a mesma forma tridimensional da malha lida.

Os nós[7] da malha são utilizados como um molde, para que as estruturas computacionais do tipo *Node*[8] sejam posicionadas; portanto, daqui por diante os nós da malha

[7]Nó representa um ponto fixo no *Lattice*.

[8]Node é uma estrutura computacional que representa a distribuição de partículas posicionada sobre um nó do *Lattice*, e que se move pelos demais nós conforme a execução da simulação.

serão chamados de *Node*. O conteúdo da estrutura *Node* pode ser visto no pseudo-código 6.1.

Pseudo-Código 6.1: Estrutura *Node*

```
struct Node{
Posicao[3]: float      // Coordenada espacial no plano cartesiano.
OwnIndex[4]: int       // Índices do nó na matriz e no vetor alocados.
Tipo: int              // Tipo do ponto. -1(externo) 0(parede) 1(fluido).
U[3]: float            // Vetor velocidade.
Rho: float             // Massa específica
F[19]: float           // Direcões de deslocamento.
NearIndex[19][2]: int // Índice e tipo dos vizinho na matriz alocada.
}
```

b) Cada *Node* armazena atributos que representam as características reais de uma distribuição de partículas, como velocidade e massa específica. De maneira computacional, referem também sua localização espacial e também a direção dos seus vizinhos.

c) A configuração do modelo de *Lattice* se dá pelo preenchimento dos atributos dos *Nodes*, que identificam as direções que podem levá-lo a seus vizinhos. Como os *Nodes* estão dispostos em uma matriz, para identificar os vizinhos, incrementa-se ou decrementa-se o índice na dimensão a ser identificada. Para melhor entender, na figura 6.2 foi utilizado um *Lattice* com 2 dimensões onde será feita a configuração do *Node* com índice (2,2). Os números na cor preta representam os índices (X,Y) nos *Nodes*. As anotações em vermelho são os incrementos ou decrementos necessários para alcançar os índices dos vizinhos.

d) Para demonstrar como é feita a identificação dos *Nodes*, observe a figura 6.3. Todos os *Nodes* possuem um atributo do tipo inteiro, que serve para diferenciá-los. Tal identificação se faz necessária devido a diferença de densidade entre os tipos de *Nodes*. Fisicamente um *Node* de parede é mais denso do que um *Node* de fluido, e um *Node* externo não possui densidade. Aos *Nodes* externos identificados na cor cinza é atribuído o valor -1, aos de parede na cor verde o valor 0, e aos de fluido na cor azul o valor 1. Tal identificação tem ainda outra finalidade, pois os *Nodes* de parede e de fluido são computados em passos diferentes, sendo submetidos a equações mutuamente excludentes, como será abordado na seção 6.3.

Os passos que implementam o *LBM* são representados na figura 6.1 pelas letras (B), (C), (D) e (E). Esses passos são feitos de forma iterativa, até que o número de iterações atinja um máximo estimado quando, neste caso, a simulação termina. O passo (F) finaliza a simulação, colhe os resultados e escreve em arquivos no formato *VTK*.

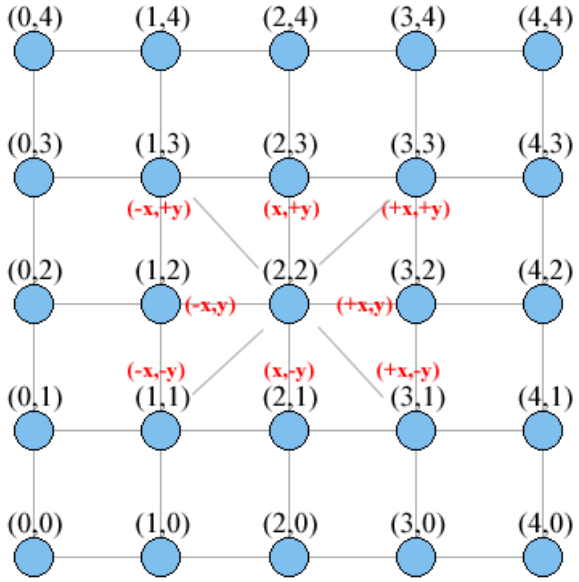


FIG. 6.2: Lattice 2D com *Nodes* indexados.

6.2 DECOMPOSIÇÃO DO DOMÍNIO DE DADOS

Em trabalho executado por (KUZNIK, 2010), o autor propôs a divisão dos *Nodes* em grupos, conforme suas densidades, fazendo a decomposição da matriz tridimensional em dois ou mais vetores unidimensionais distintos.

São utilizados nesse trabalho dois vetores, um para os *Nodes* de fluido e outro para os *Nodes* de parede, sendo os *Nodes* externos descartados, como ilustra a figura 6.4.

Essa decomposição favorece a implementação do *LBM* em dois aspectos: execução dos passos e a utilização *coalescente* de memória.

Com relação ao primeiro aspecto, considere que cada passo iterativo é executado sobre um diferente grupo de *Nodes*. Decompondo-se a matriz em grupos, cada passo recebe como entrada um vetor que contém um subconjunto de *Nodes*. Desta forma não há a necessidade de percorrer toda a matriz tridimensional a cada passo.

Com relação ao segundo aspecto, formando-se subconjuntos de *Nodes*, garantindo a proximidade dos *Nodes* a serem computados por um passo, garante-se também proximidade na alocação desses *Nodes* na memória da *GPU*, favorecendo o acesso coalescente à memória, como observado por (KUZNIK, 2010) em seu trabalho.

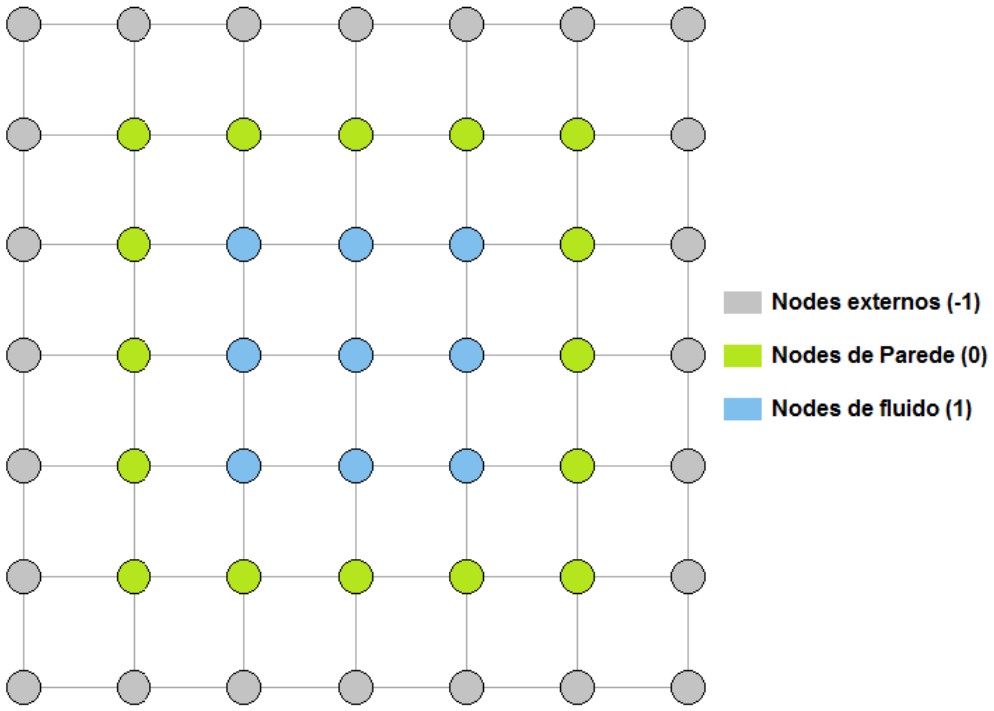


FIG. 6.3: *Nodes* configurados conforme suas densidades em um *Lattice* 2D.

6.3 EXECUÇÃO DAS ITERAÇÕES DO LBM

Em regra, o *LBM* possui dois passos: colisão (B) e propagação(D). O passo de condição de contorno (C) é uma ponderação utilizada para simular a existência de um contorno sobre a dinâmica do fluido, que respeita os mesmos princípios de conservação de massa e momento, visando não desequilibrar a execução do método.

O passo (E) é o último passo da iteração, sendo responsável por atualizar os valores macroscópicos de velocidade e massa específica de todas as distribuições, de modo que a próxima iteração ocorra sobre as distribuições equilibradas e já movimentadas, dando continuidade ao movimento.

Cada passo distinto implementa diferentes equações, que são aplicadas a subconjuntos de *Nodes*. Tais passos não podem ser executados em paralelo, pois existem dependências entre eles, entretanto as computações dentro de cada passo podem ser paralelizadas.

Na implementação sequencial tal dependência não fica explícita devido à sequencialidade das iterações. Nas implementações paralelas esse é um fator que deve ser levado em conta.

O passo (B) implementa a equação 4.9, apresentada na seção 4.4.2, sendo aplicado em todos os *Nodes* de fluido. O procedimento trata as colisões internas ocorridas em cada

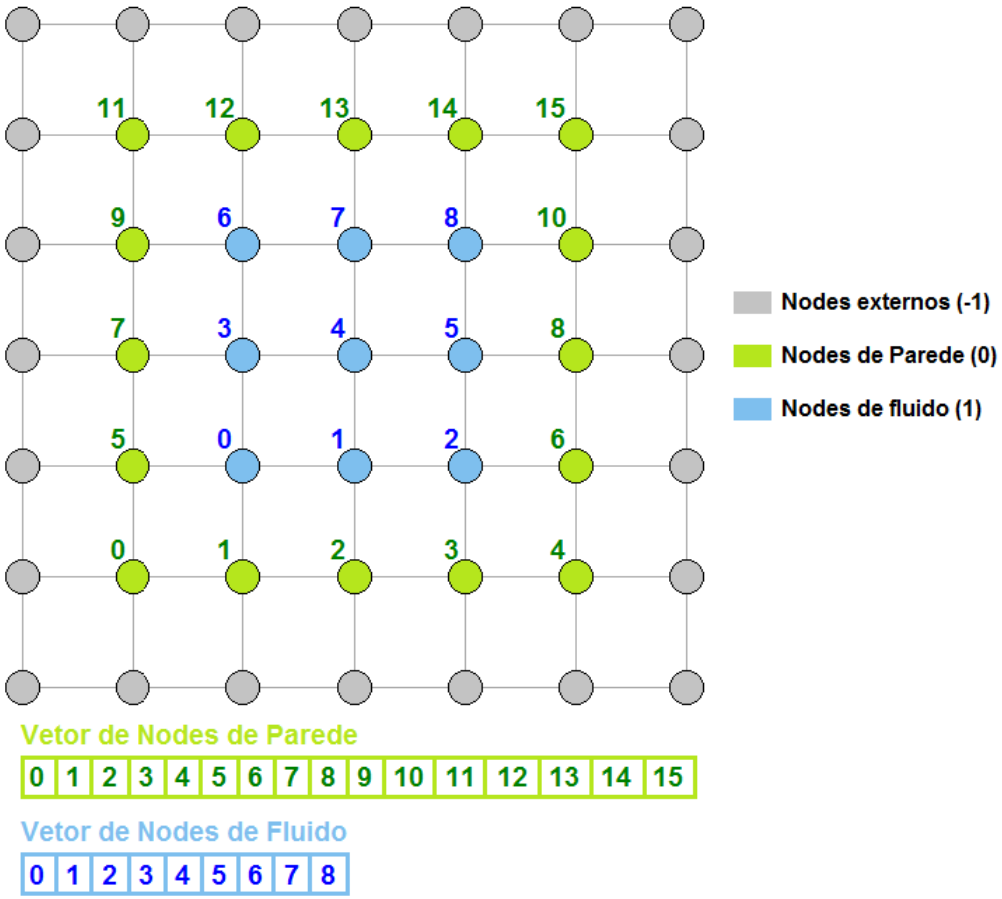


FIG. 6.4: Decomposição da matriz tridimensional.

Node pelas distribuições de partículas, definindo para qual direção de movimento cada distribuição irá se mover.

Na execução deste passo, cada um dos *Nodes* utiliza seus atributos internos de velocidade e massa específica, juntamente com o operador de colisão, definindo as direções de movimento resultantes. Por este motivo, o passo de colisão pode ter as computações de todos os *Nodes* executadas em paralelo sem qualquer dependência. Observa-se o pseudo-código 6.2 relacionado à equação 4.9.

Pseudo-Código 6.2: Resumo do passo de colisão

```
1 void Colisao()
2 {
3   for(int j=0; j < TODOS_NODES_DE_FLUIDO; j++)
4     {
5     for(int i=0; i<19; i++)
6       {
7       float Fi, Fi_eq, Colisao;
```

```
8         Fi = Nodes_Fluido[j].F[i];              //Pego a Fi
9         Fi_eq = DistribEquilibrio(RHO0, V);     //Calculo da Fi de equilibrio
10        Colisao = ( (1.0/TAU) * Fi_eq - Fi);    //Calculo efetivo de colisao
11        Nodes_Fluido[j].F[i] = Fi + Colisao;    //Escrita da Fi da distribuição
12        }
13    }
14 }
```

O primeiro termo da equação 4.9, $f_i(\vec{x}, t)$, representa uma distribuição $F[i]$ do pseudo-código 6.2, onde i varia de 0 a 18 distribuições. Este termo é obtido na linha 8.

Sobre o segundo termo da equação 4.9 composto por $\frac{1}{\tau}[f_i^{eq}(\vec{x}, t) - f_i(\vec{x}, t)]$, calcula-se o termo $f_i^{eq}(\vec{x}, t)$ na linha 9 através da função *DistribEquilibrio()*, que implementa a equação 4.11 apresentada na seção 4.4.3 e utiliza como base para o cálculo as constantes $\rho 0$ (massa específica média) e v (viscosidade cinemática), representadas no pseudo-código por *RHO0* e V respectivamente, juntamente com as variáveis $\vec{u}$ (vetor de velocidade) e ρ (massa específica). Este procedimento visa manter o equilíbrio entre as distribuições contidas nos *Nodes* para que não ocorra nem perda e nem ganho de massa. O cálculo do segundo termo da equação é feito na linha 10.

Por fim, somando o primeiro termo ao segundo, conforme a equação de colisão 4.9, a nova distribuição é atribuída a variável $F[i]$ na linha 11.

Visando mostrar de forma clara o resultado final do passo de colisão, toma-se como exemplo a figura 6.5. Embora este passo seja aplicado em todos os *Nodes* de fluido, será exibida apenas a colisão executada no *Node* central.

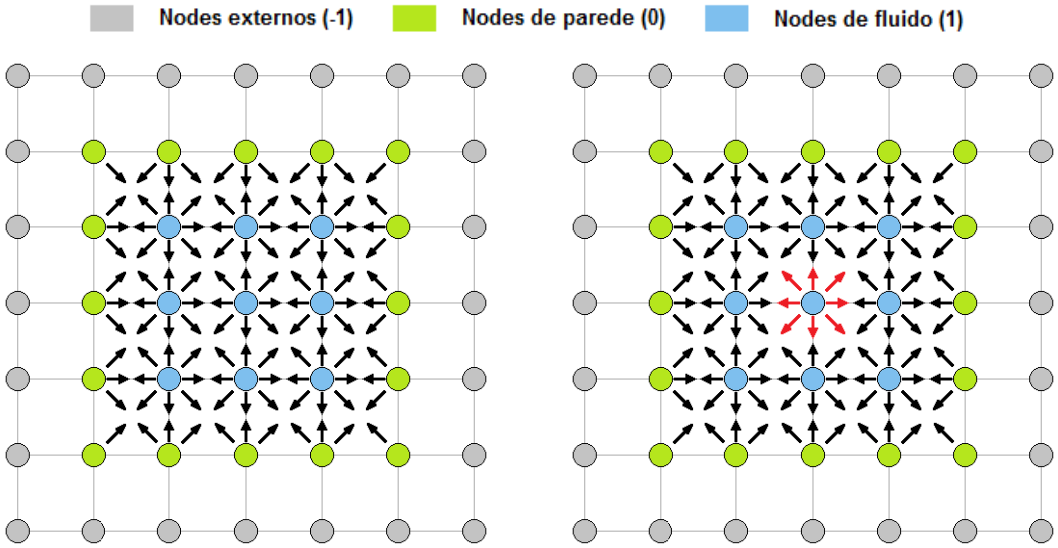
A figura 6.5(a) mostra a configuração das distribuições de partículas antes do cálculo de colisão. As direções de movimento existem mas ainda não foram calculadas as distribuições que irão segui-las. Após o cálculo de colisão, a figura 6.5(b) mostra todas as distribuições com as direções definidas.

O passo (C), implementa as equações 4.13, 4.14 e 4.15 apresentadas na seção 4.4.5, aplicadas a todos os *Nodes* de parede.

Para exemplificar o funcionamento desse passo, toma-se como exemplo o esquema da figura 6.6, que exibe o emprego do passo em questão em apenas um *Node* denominado *P*.

Ilustrado na figura 6.6(a) o *Node* de parede *P* verifica quais vizinhos de fluido possuem porções de suas distribuições apontadas para ele, no caso, *F1* e *F2*, e as redirecionam de acordo com a regra imposta pelas equações de condição de contorno, denotado pela figura 6.6(b).

O passo (C) é dependente de (B), pois todas as direções de movimento dos *Nodes* de



(a) Início de cada iteração (b) Iteração após o passo de colisão

FIG. 6.5: Esquema que denota a aplicação do passo de colisão.

fluido devem estar definidas pelo passo (B) para que sejam verificadas pelo passo (C).

Embora haja tal dependência entre (C) e (B), a equação de condição de contorno do passo (C) utiliza como base para os cálculos, as direções de movimento $F[i]$ dos *Nodes* de fluido calculados no passo (B) que possuem vizinhança com os *Nodes* de parede, juntamente com os atributos internos de velocidade e massa específica dos *Nodes* de parede. O resultado da equação é armazenado no atributo interno $F[i]$ do *Node* de parede. Portanto, o término do passo (B) possibilita a execução em paralelo do passo (C) independentemente.

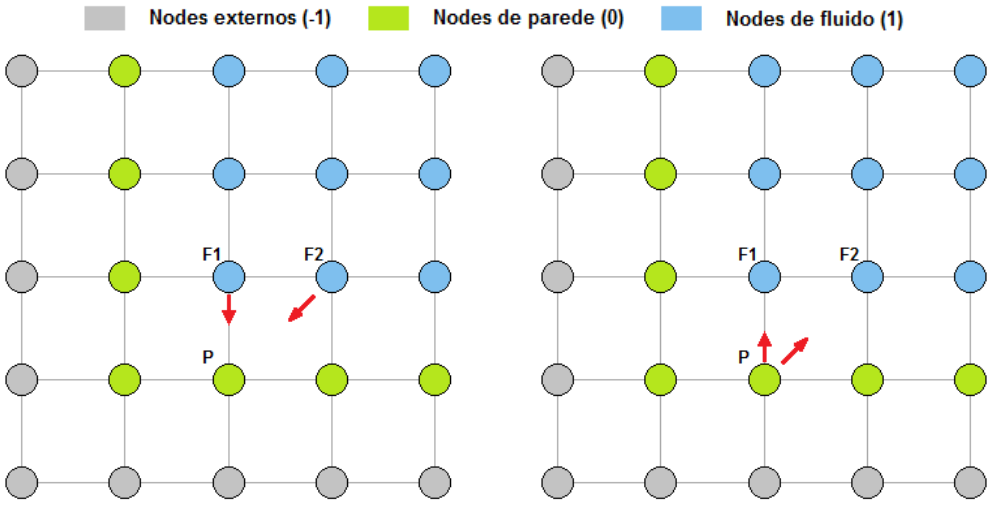
O passo (D) implementa a equação 4.10 apresentada na seção 4.4.4, sendo destinado a todos os *Nodes* (parede e fluido). Realiza a movimentação das distribuições de partículas, até as direções de movimento calculadas pelos passos (B) e (C).

Como as direções de movimento dos *Nodes* de fluido são definidas pelo passo (B), e dos *Nodes* de parede pelo passo (C); logo, o passo (D) é dependente de (B) e (C).

Para exemplificar tal dependência, como sequência da figura 6.5, a figura 6.7(a) denota o estado das distribuições do *Node* central já propagadas, e a figura 6.7(b) a propagação das distribuições que alcançaram o *Node P*, como sequência da figura 6.6.

Apesar de tal dependência do passo (D) com (B) e (C), a metodologia utilizada para executar a propagação realiza trocas entre posições de memória sem haver sobrescrita dos valores, facultando a execução em paralelo do passo (D) de forma independente, conforme descrito na seção 4.4.4.

Após a movimentação completa das distribuições, o passo (E) efetua a atualização dos



(a) Verifica distribuições vizinhas. (b) Redireciona distribuições.

FIG. 6.6: Esquema que denota a aplicação do passo de condição de contorno.

valores macroscópicos em todos os *Nodes* aplicando as equações 4.5 e 4.6, apresentadas na seção 4.4.1, com base nas novas distribuições neles contidas.

A atualização depende de que todas direções tenham sido propagadas, formando as novas distribuições. Por esta razão o passo (E) depende do passo (D). Uma vez atendida tal dependência, como utilizam as próprias distribuições contidas no *Node* como base para a equação, este passo também pode ser completamente paralelizado sem qualquer dependência.

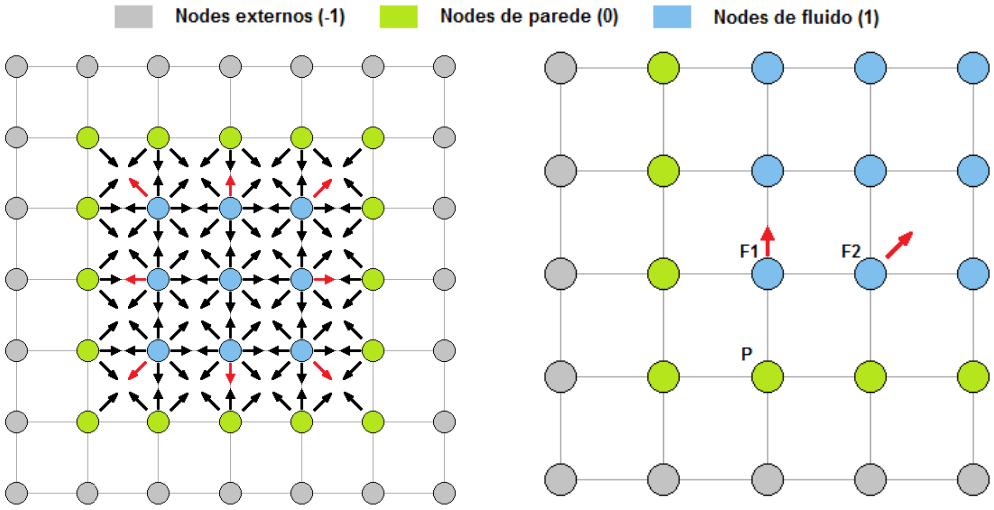
Neste ponto o loop é iterado, tornando a repetir os passos (B), (C), (D) e (E).

6.4 IMPLEMENTAÇÃO DO LBM EM GPU

Conforme exposto na seção 6.3 os passos do *LBM* possuem dependências entre si, mas não possuem a mesma dependência nas suas computações internas. Devido a essa característica, pode-se implementar uma chamada de *kernel* para cada passo iterativo, dispostos de forma sequencial.

Quando todos os *threads* de um bloco terminam seus processamentos referentes a um passo, iniciam o processamento do passo seguinte, sem a garantia de que os outros blocos tenham também terminado seus processamentos referentes ao passo anterior. Como os passos possuem dependência entre si, isso acarretaria em erros na simulação.

Considerando a utilização de múltiplos *kernels*, mesmo dispostos de forma sequencial, as chamadas de *kernel* são *assíncronas*. Isto também causa inconsistência na simulação devido à necessidade de sincronização.



(a) *Nodes* de fluido. (b) *Nodes* de parede.

FIG. 6.7: Aplicação do passo de propagação

Neste caso, a arquitetura *CUDA* oferece funcionalidades que permitem a sincronia de todos os *threads* de um *kernel*. Esta função é chamada pela UCP, em sequência a uma chamada de *kernel*, diferente de sua correlata na *GPU* que precisa ser chamada internamente.

A figura 6.8 ilustra a localização dos pontos de sincronização para a implementação em *GPU*, que não estão presentes para a versão em UCP devido a característica sequencial da implementação.

Para configurar um *kernel*, é necessário definir quantos recursos da *GPU* serão empregados, isto é, o número de blocos e *threads*. A escolha desses recursos está relacionada à computação que será executada, sendo possível configurar esquemas distintos com diferentes quantidades de blocos e *threads* e, desta forma, aproveitar melhor a arquitetura. Adicionalmente podem ser passadas variáveis como parâmetro, para utilização dentro do *kernel*.

Toda *GPU* possui uma capacidade máxima de processamento paralelo, que varia de acordo com a quantidade de *Stream Processor* que cada modelo possui. Essa capacidade está ligada a quantidade de blocos e *threads* executáveis simultaneamente. Entretanto, pode-se utilizar números de blocos e *threads* superior a capacidade simultânea da *GPU*, ficando estes escalonados, sendo a própria *GPU* responsável pelo escalonamento e gerenciamento da utilização desses recursos.

Essa extrapolação de capacidade paralela é limitada pela capacidade total de gerenciamento de recursos da arquitetura da *GPU* utilizada. A exemplo da *GPU GTX 460*, segundo a *NVIDIA* (NVIDIA, 2011), possui 336 *CUDA cores*, organizados em 7 multi-

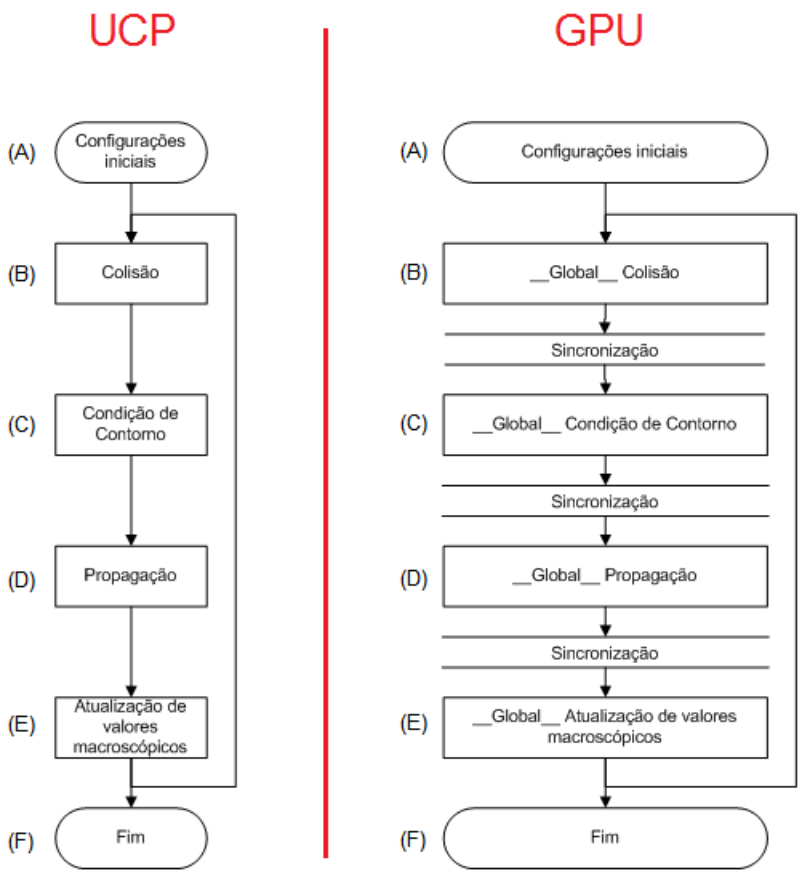


FIG. 6.8: Estrutura do algoritmo do *LBM* para execução sequencial e paralela

processadores com 48 *Stream Processors (SP)*, tendo a capacidade de gerenciar 65535 blocos, cada um contendo 48 *warps* com 32 *threads*, totalizando 100.661.760 *threads* gerenciados.

Com a utilização de vetores unidimensionais, apenas uma dimensão para blocos e *threads* são usados. Desta forma, multiplicando-se o número de blocos pelo número de *threads*, o valor obtido tem que alcançar o número máximo de *Nodes* do vetor a ser computado.

A título de exemplo, supõe-se a existência de *1025 Nodes* no vetor a ser computado, e o número de *threads 64*. Para a configuração de blocos calcula-se *[1025/64] = 16*, no qual considera-se apenas a parte inteira, pois não se pode configurar a fração de um bloco. Multiplicando o número de blocos pela quantidade de *threads* tem-se, *16*64 = 1024*.

Aproveitando o mesmo exemplo, se o número de *Nodes* do vetor é *1025*, um *Node* não será computado. Essa diferença é resolvida acrescentando um bloco a mais na configuração de blocos. Dessa forma, apenas o último bloco será sub-utilizado. Pensando em 16 blocos, a sub-utilização de um bloco é um grande desperdício, mas quando se tem centenas de

milhares de *Nodes* por exemplo, um bloco em *1563* é uma quantidade muito pequena, que se torna ainda menor com quantidades de *Nodes* maiores.

A arquitetura da *GPU* utiliza *N threads* e *M* blocos para executar *N * M* vezes o *Kernel* configurado. Cada *thread* possui um identificador dentro do bloco, que combinando às variáveis de identificação e dimensão do bloco, resultam em um identificador único, crescente, e que no exemplo apresentado, varia de 0 até o número máximo atingido por *(N Threads * (M Blocos + 1 Bloco))*.

Foi adotada uma estratégia para melhor empregar os recursos da *GPU*, na qual foram utilizados dois esquemas de configuração para o número de blocos e *threads*. Cada *kernel* processa um passo iterativo e cada passo iterativo computa um sub-grupo de *Nodes* como exposto na seção 6.3. Pode existir uma grande diferença entre a quantidade de *Nodes* contida em cada sub-grupo e com a utilização de dois esquemas de configuração evita-se o desperdício de recursos, pois cada *kernel* é configurado com o número de blocos e threads necessários para que o passo iterativo possa computar o sub-grupo a ele designado.

6.5 ASPECTOS DE OTIMIZAÇÃO

Duas otimizações estão presentes para ambas as implementações (sequencial e paralela), reduzindo a entrada de dados e favorecendo o acesso à memória de forma coalescente, mas não são usadas como medida de desempenho. Uma única otimização foi desenvolvida para implementação paralela, visando um maior aproveitamento da arquitetura da *GPU*.

A seguir serão abordadas as otimizações executadas em ambas as implementações, seguida das consequências de sua utilização.

Como primeira otimização, foi descrita na seção 6.2 a decomposição da matriz tridimensional de *Nodes* alocados em dois vetores unidimensionais distintos, conforme suas densidades, com o intuito de direcioná-los separadamente aos passos iterativos que irão computá-los. Em (KUZNIK, 2010) o autor destacou que tal decomposição favorece o acesso coalescente à memória, pois garante a proximidade dos dados facilitando a sua busca. A aproximação dos dados em memória foi exibida na seção 6.2 pela figura 6.4 com a decomposição dos dados da matriz tridimensional em vetores unidimensionais.

Ao se decompor a matriz, cada *Node* armazena internamente em sua estrutura os índices que o referenciam tanto na matriz tridimensional quanto no vetor unidimensional, o que se torna útil caso seja necessário retornar com os dados do vetor para a matriz. Técnica semelhante foi apresentada por (DUPUIS, 2002), que decompôs um domínio bidimensional em um vetor unidimensional.

Como os passos iterativos calculam conjuntos diferentes de *Nodes*, essa técnica de decomposição reduz o processamento, pois ao invés de cada um dos passos percorrer toda a matriz tridimensional, ela percorre apenas um vetor unidimensional, que contém o subconjunto de *Nodes* a ser computado pelo passo.

A segunda otimização executada para todas as implementações refere-se ao passo de propagação, visando reduzir ao máximo o uso de memória, especialmente importante ao utilizar *GPU*. Foi adotada uma técnica de *swap* proposta por (MATTILA, 2007), que promove trocas de dados entre posições de memória sem que haja sobrescrita de dados, e sem a necessidade de duplicação da malha. Tal procedimento foi apresentado em detalhes na seção 4.4.4.

6.6 UTILIZAÇÃO DE MEMÓRIA CONSTANTE

Além das otimizações compartilhadas para ambas implementações, foi feita de forma específica uma otimização para a implementação paralela, descrita a seguir, visando obter um ganho ainda maior no desempenho.

Como proposto por (WELLEIN, 2006), a escolha correta dos dados a serem carregados nas memórias de menor latência é fator importante pata atingir um bom desempenho e, baseado em sua proposta, em uma das implementações paralelas os dados que permanecem constantes durante toda a simulação foram copiados para a memória constante da *GPU*.

A memória constante é uma memória de baixa latência, podendo ser comparada a memória compartilhada. Essa memória funciona como uma memória cache residente na memória global, acessada diretamente pelos *threads*. Ela pode ser acessada para escrita apenas de fora do *kernel*, por uma função chamada na UCP, mantendo seus dados como somente leitura para os *threads* internamente ao *kernel*.

Os dados escolhidos para carregamento na memória constante são frequentemente requisitados pelos passos iterativos. Quando um dado é requisitado à memória global da *GPU*, que possui a maior das latências na hierarquia de memória, ocorre uma troca de contexto. Essa troca implica em retirar um *warp* de processamento enquanto espera o dado requisitado da memória, e colocar outro. Esse procedimento, embora eficiente para esconder a latência da memória, quando efetuado muitas vezes acarreta em perda no desempenho. Quando os dados são requisitados à memória constante, não há a troca de contexto.

São estes os principais dados carregados na memória constante: ρ_0, v, τ, ω_i, $\vec{e}_i$. Para justificar a escolha destas constantes, serão exibidas suas utilizações pelos passos

iterativos.

A maior parte desses dados[9] compõe a equação de distribuição de equilíbrio 4.11, que é utilizada por dois passos iterativos: colisão e condição de contorno, com o intuito de manter o equilíbrio de massa dos *Nodes* envolvidos na equação.

O passo de colisão chama a função de distribuição de equilíbrio, uma vez para cada direção de movimento de um *Node* de fluido a ser computado. Por exemplo se existirem 100.000 *Nodes*, multiplicando pelo número de direções de movimento, que são 19 conforme o modelo do *Lattice*, serão chamadas 1.900.000 vezes essa função. Sem contar que tal equação requisita 4 das 5 constantes apresentadas em diversos termos da equação. Levando em conta que a cada requisição de uma dessas constantes há uma troca de contexto, é relevante a colocação desses dados na memória constante. Adicionalmente o passo de colisão utiliza o outro dado restante[10].

O passo de condição de contorno chama a função de distribuição de equilíbrio, 2 vezes para cada direção de movimento de um *Node* de parede a ser computado. São feitas 2 chamadas por se tratar de uma distribuição proveniente de um *Node* de fluido, que chega até um *Node* de parede sendo então redirecionada, o que implica na necessidade de garantir o equilíbrio de massa dos dois *Nodes* envolvidos.

6.7 REVISÃO DA PROPOSTA DE IMPLEMENTAÇÃO

O *LBM* apresentado nesta dissertação simula a dinâmica de um fluido dentro de uma geometria qualquer, utilizando uma implementação sequencial e duas paralelas para análise de desempenho.

As implementações recebem como entrada uma malha de nós com 3 dimensões oriunda de um arquivo no formato VTK, apresentado na seção 2.1, com informações de coordenadas e espaçamento entre os nós. Executa um processamento iterativo sobre a malha e finalmente retorna outra malha com dados escalares de velocidade agregados aos nós.

A malha lida representa uma geometria qualquer retratada por um conjunto de nós com coordenadas espaciais formando uma matriz tridimensional. Sobre essa malha são alocadas estruturas computacionais denominadas de *Node*, apresentadas no pseudo-código 6.1. Os *Nodes* podem ter diferentes densidades: os sólidos representam as paredes e os líquidos representam distribuições de micro partículas. Uma malha no âmbito do *LBM* denomina-se *Lattice*.

[9] Os dados requisitados pela distribuição de equilíbrio são:ρ_0, v, ω_i, $\vec{e}_i$

[10] O dado requisitado pela colisão é o τ

Sobre o *Lattice* são executadas configurações que caracterizam as propriedades físicas dos *Nodes*, definindo quais *Nodes* são de parede e quais são de fluído. É definido também o esquema de vizinhança que caracteriza as direções de movimento possíveis para o deslocamento do fluído. O esquema escolhido denomina-se *D3Q19* e foi apresentado na seção 4.3. Os procedimentos de configuração inicial foram expostos em detalhes na seção 6.1.

O processamento no *LBM* ocorre de forma iterativa. Na primeira iteração, é estabelecida uma quantidade de movimento inicial, e a cada iteração a movimentação sucede sobre a condição de velocidade da iteração anterior, causando a continuação do movimento.

Uma iteração é dividida em 4 passos: colisão, condição de contorno, propagação e atualização de valores macroscópicos.

O passo de colisão trata as colisões entre as distribuições de partículas dentro dos *Nodes* de fluido, definindo as direções de movimento para onde cada porção de fluido irá se mover.

O passo de condição de contorno depende do término do passo de colisão. É aplicada em todos os *Nodes* de parede, redirecionando as direções de movimento das porções de partículas provenientes dos *Nodes* de fluido vizinhos.

O passo de propagação depende do término dos passos de colisão e de condição de contorno, pois executa o deslocamento das distribuições de partículas nas direções de movimento definidas por eles. Este passo é aplicado a todos os *Nodes*.

O passo de atualização de valores macroscópicos depende do término do passo de propagação, pois calcula o novo vetor de velocidade e a nova massa específica para cada *Node* de fluido, baseados nas novas distribuições de partículas.

As duas implementações paralelas foram submetidas a processamento em *GPU*, no qual foi utilizado um *kernel* para computar cada passo iterativo. Cada passo iterativo computa um sub-grupo de *Nodes* e devido ao fato de existir uma grande diferença quantitativa entre os sub-grupos foram utilizados dois esquemas para utilização dos recursos da *GPU*, nos quais objetiva-se configurar o *kernel* apenas com os recursos necessários para que o passo iterativo possa computar seu sub-grupo. Devido a dependência entre os passos iterativos houve a necessidade de incluir nas versões paralelas pontos de sincronismo entre eles. Fato este irrelevante para a implementação sequencial.

Foram feitas duas otimizações a nível de pré-processamento, sendo estas presentes em todas as implementações, visando reduzir a complexidade algorítmica e favorecer a coalescência de memória.

Na primeira otimização a matriz tridimensional é decomposta em dois vetores uni-

dimensionais, um contendo os *Nodes* do tipo parede e outro contendo os *Nodes* do tipo fluido. Esse procedimento evita que um passo destinado a apenas um tipo de *Node* tenha que processar toda a matriz em busca dos *Nodes* requeridos, favorecendo também a coalescência de memória, pois agrupam os tipos de *Nodes* que serão processados em conjunto.

A segunda otimização implementa um procedimento de *swap* no passo de propagação, proposta por (MATTILA, 2007), que promove trocas de dados entre posições de memória sem que haja sobrescrita de dados, e sem a necessidade de duplicação da malha.

Além das otimizações de caráter geral, foi feita uma otimização especificamente para uma das implementações paralelas referente a utilização de memória. Os dados constantes requisitados com maior frequência foram copiados para uma memória de menor latência, presente na hierarquia de memória da *GPU*, visando melhorar ainda mais o desempenho.

7 RESULTADOS

Neste capítulo serão apresentadas análises sobre as implementações do algoritmo do *LBM*, a saber: sequencial, paralela, e paralela com otimização de memória.

O objetivo é analisar desempenho e escalabilidade comparando as implementações, medindo também o tempo despendido nas diferentes partes do algoritmo para inferir o peso computacional de cada uma sobre o processamento total. Adicionalmente, serão feitas medições em termos de ganho de desempenho das implementações paralelas em relação a implementação sequencial.

Foram utilizadas as seguintes métricas para avaliar os experimentos: (i) quantidade de nós atualizada por segundo, na qual serão discutidas questões de desempenho e escalabilidade; (ii) percentagem do processamento de cada passo iterativo, no qual serão discutidas questões de desempenho de cada passo separadamente; (iii) *Speedup*, no qual será apresentado o ganho de desempenho apresentado pelas versões paralelas em relação à sequencial.

O cálculo do número de nós atualizados por segundo é obtido através da multiplicação do número de iterações pela quantidade de *Nodes* computados, sendo esse resultado dividido pelo tempo total da simulação. O número de iterações está ligado ao tamanho da entrada de dados, o que resulta em experimentos com número de iterações diferentes. Por esse motivo foi escolhido um número estimado em 3000, para que todos os experimentos possam executar o mesmo número de iterações.

Para calcular a percentagem de processamento dos passos, a cada iteração soma-se o tempo de processamento do passo e armazena-se separadamente em sub-tempos. Os sub-tempos somados formam o tempo total da simulação. Dividindo-se o tempo total da simulação por cada sub-tempo, tem-se a proporção que cada passo ocupou do processamento.

O *Speedup* é uma métrica utilizada na computação para medir o desempenho alcançado por uma aplicação paralela em comparação a uma sequencial. Para isso divide-se o tempo total da aplicação sequencial pelo tempo total da aplicação paralela.

7.1 DESCRIÇÃO DOS EXPERIMENTOS

Os *hardwares* utilizados nos experimentos foram dois computadores equipados com processador *Intel Quad Core Q9450* de 2.66 *GHz* no qual apenas um núcleo do processador foi utilizado. Um dos computadores possui a *GPU GeForce 9600 GT* e o outro a *GPU GeForce GTX 460*. As especificações das *GPUs* estão expostas na tabela 7.1, onde pode ser visto que a *GPU GTX 460* é superior a *GPU 9600 GT* em todas as especificações, sendo inferior apenas em *clock* de processamento.

Processamento	*GeForce 9600 GT*	*GeForce GTX 460*
Arquitetura	G80	FERMI
CUDA Cores	64	336
Clock	1625 *MHz*	1350 *MHz*
Memória	*GeForce 9600 GT*	*GeForce GTX 460*
Capacidade	512 *MB*	1024 *MB*
Clock	900 *MHz*	1800 *MHz*
Largura de banda(GB/seg)	57.6	115.2

TAB. 7.1: Especificações das *GPUs* utilizadas.

Os *softwares* utilizados são os seguintes: Sistema operacional *Linux Ubuntu* 10.04 de 64 *bits* e o *CUDA ToolKit* versão 3.2.

Foram feitos experimentos com dois modelos de entrada de dados para demonstrar a capacidade do *LBM* de simular fluxos em diferentes geometrias. Um deles é popularmente conhecido como Cavidade Dirigida (*Driven Cavity*), utilizado por diversos autores (ZHOU, 2008), (XIAN, 2011), (OBRECHT, 2011), (FEICHTINGER, 2011), (SCHREIBER, 2011), (RIBBROCK, 2010), (KUZNIK, 2010), entre outros. Essa geometria é criada artificialmente na qual são definidas as quantidades de pontos existentes em cada direção cartesiana.

O outro experimento utiliza uma geometria extraída de um conjunto de imagens denominada *DICOM* [11]. A geometria possui o formato de uma artéria e mostra que o *LBM* pode também simular o fluxo em modelos reais aproximados.

7.2 MODELO DA CAVIDADE DIRIGIDA

No modelo conhecido como cavidade dirigida, a dinâmica do fluido ocorre no interior de um cubo, onde todas as dimensões possuem a quantidade de *Nodes* equivalentes.

[11] O procedimento de extração da geometria é feito pelo módulo de processamento de imagens, descrito na seção 2.1

A parede superior ou tampa foi configurada com velocidade constante $\vec{u} = \{1, 0, 0\}$, movimentando a tampa na direção $+X$ do plano cartesiano, semelhante a uma esteira rolante. As demais paredes foram configuradas com velocidade nula $\vec{u} = \{0, 0, 0\}$. A medida em que a tampa se move, provoca a movimentação do fluido contido no interior da cavidade.

Utilizou-se tamanhos diferentes de cavidade para fazer a variação da entrada de dados da simulação.

Tamanhos	*Nodes* de Paredes	*Nodes* de Fluido	% Parede	% Fluido
12^3	728	1.000	42%	58%
22^3	2.648	8.000	25%	75%
32^3	5.768	27.000	18%	82%
42^3	10.088	64.000	14%	86%
52^3	15.608	125.000	11%	89%

TAB. 7.2: Variação do tamanho da cavidade.

A forma de variação do tamanho é mostrada na tabela 7.2, onde podem ser vistas as características da malha como tamanho, quantidade de *Nodes* de parede, de fluido e percentual de cada tipo de *Node* em relação ao número total de *Nodes*.

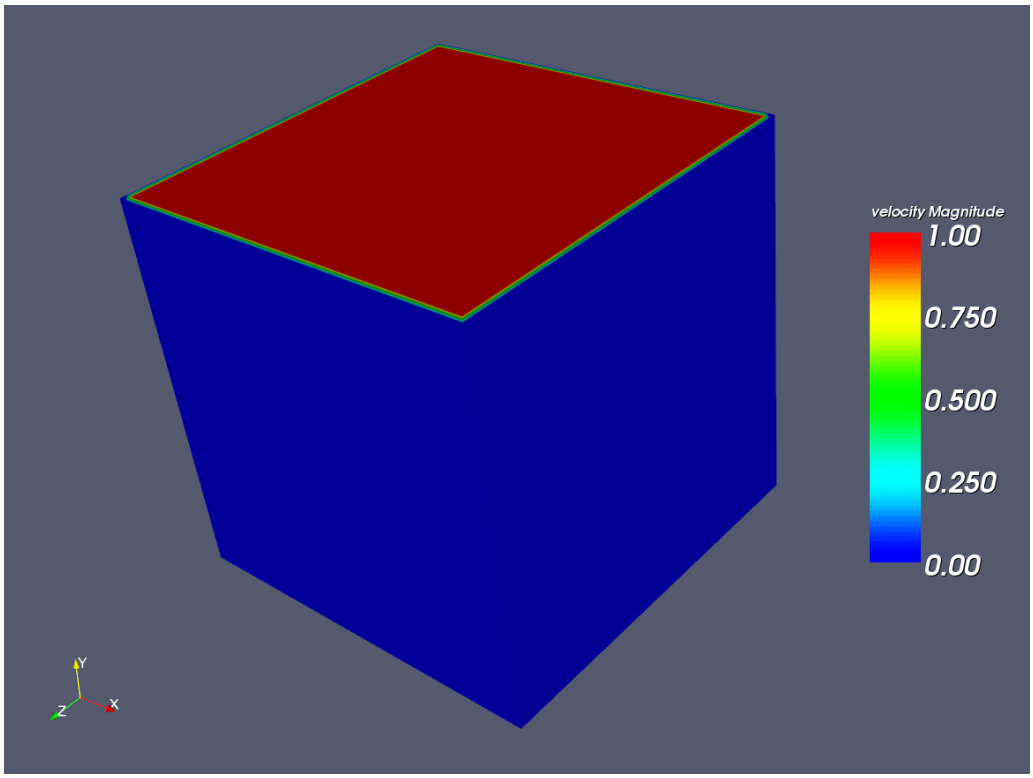
Para visualização da malha resultante, escolheu-se a cavidade com 52^3, por possuir maior resolução devido a maior quantidade de *Nodes*, 52 em cada dimensão.

De acordo com a figura 7.1(a) é apresentado o corpo da cavidade. Na figura 7.1(b) foi feito um corte no eixo Z expondo o plano (X,Y), possibilitando a visualização dos valores escalares de velocidade associados aos *Nodes* dentro da cavidade. Na figura 7.2(a) são utilizadas linhas de corrente para destacar a trajetória do fluido. Na figura 7.2(b) a cavidade pode ser vista sob outro ângulo, no qual percebe-se que o deslocamento do fluido em direção às paredes tem sua trejetória redirecionada, formando um vórtice.

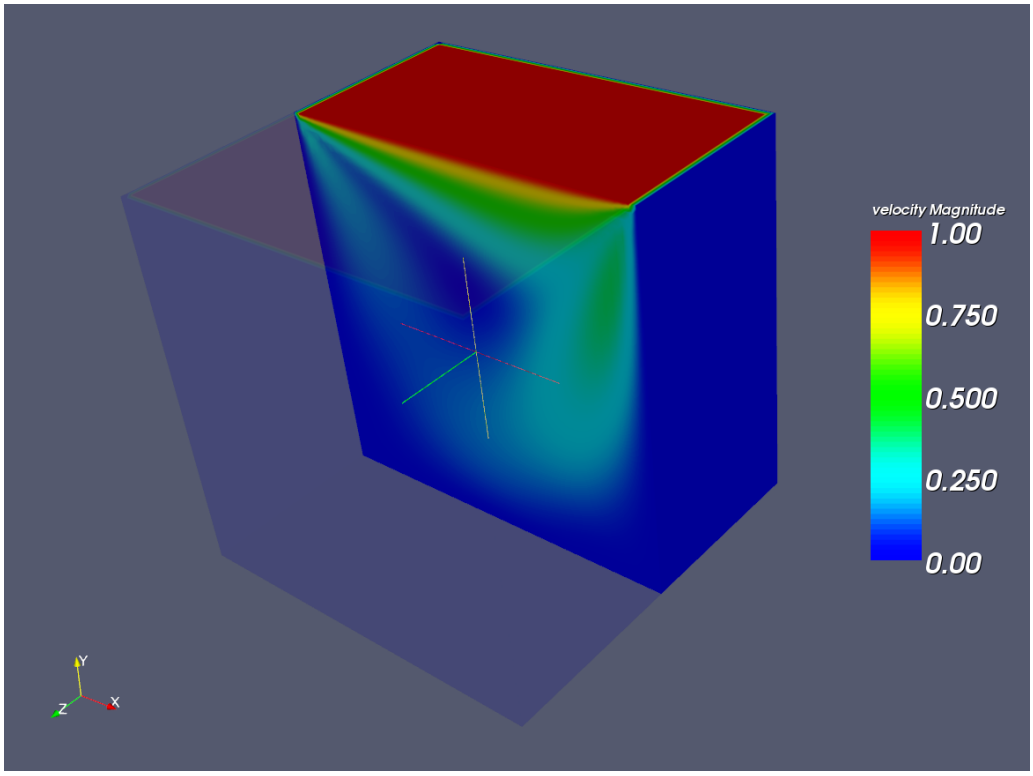
7.3 ANÁLISE DOS RESULTADOS DA CAVIDADE

Foram feitas 5 simulações com 3000 iterações para cada um dos 5 tamanhos de cavidade. Obteve-se as médias dos tempos de execução de cada passo medidas em segundos, juntamente com a soma delas para a geração dos gráficos. A tabela 7.3 exibe as médias dos tempos totais.

No que tange a alocação de memória e transferência de dados da memória principal da UCP para a memória global da *GPU*, o tempo desses procedimentos somados são inferiores a 5 centésimos de segundo para o maior caso. Portanto, não serão avaliadas

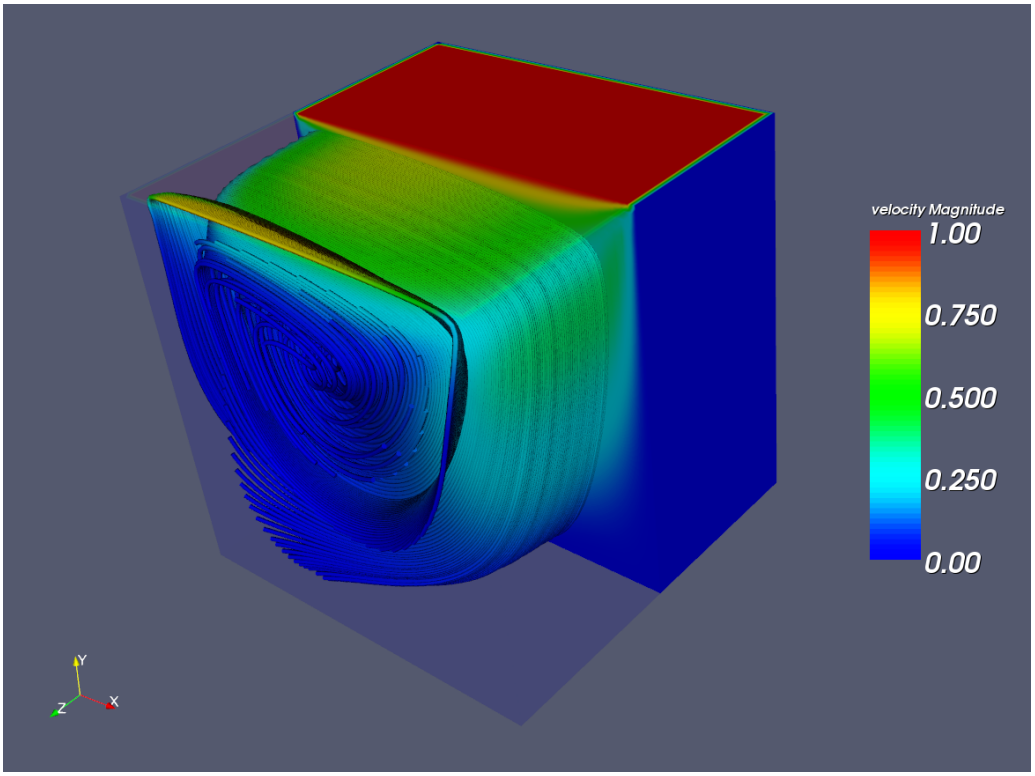


(a) Superfície

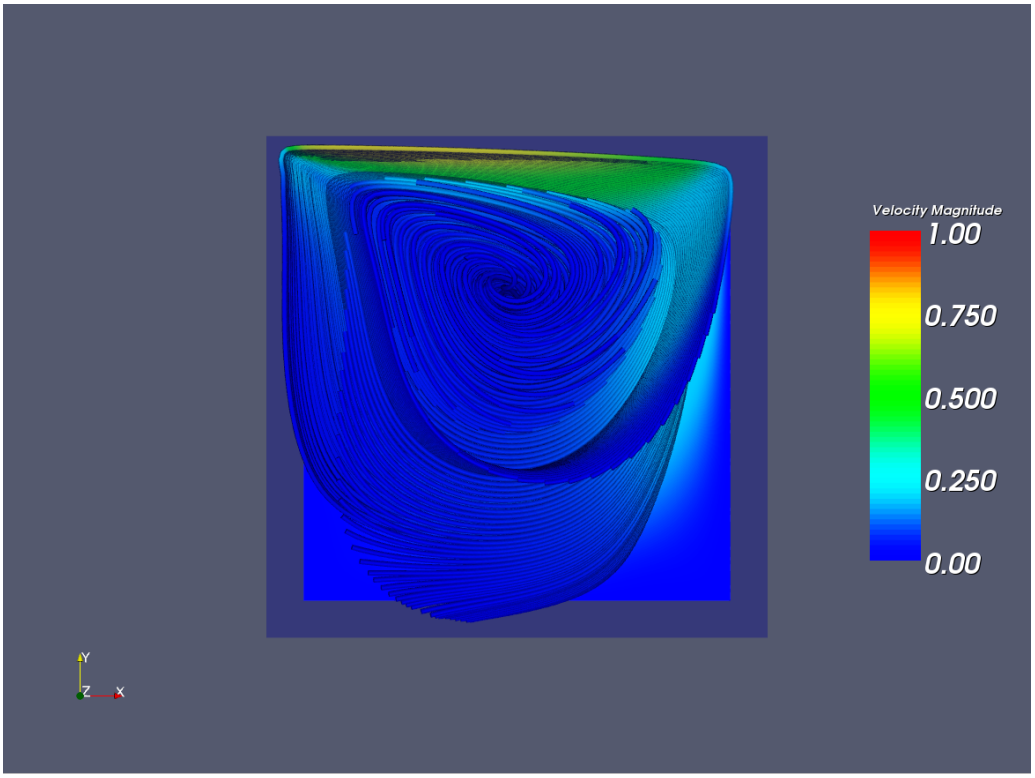


(b) Filtro Clip

FIG. 7.1: Visualização baseada na escala de velocidade do fluido.



(a) Linhas de corrente



(b) Formação de vórtice

FIG. 7.2: Visualização baseada na escala de velocidade do fluido com linhas de corrente.

Tamanhos	UCP	*9600 GT*	*9600 GT* Otmz.	*GTX 460*	*GTX 460* Otmz.
12^3	14,00	2,42	1,46	1,40	0,85
22^3	99,92	12,22	4,90	4,11	1,65
32^3	336,28	38,89	13,84	11,12	3,96
42^3	789,05	89,88	30,79	25,09	8,60
52^3	1528,20	172,64	57,56	47,58	15,86

TAB. 7.3: Média em segundos dos tempos de execução da cavidade dirigida.

questões de transferências de dados de entrada e saída para a *GPU*. Essas questões não tem relação com a utilização da hierarquia de memória internamente a *GPU*, pois se trata apenas do tráfego de entrada e saída de dados para utilização em *GPU*.

O primeiro aspecto apontado para análise, quantidade de nós atualizada por segundo, é uma métrica utilizada para se fazer inferências ao *LBM* que quantifica os *Nodes* computados por segundo. Diversos autores como (OBRECHT, 2011), (SCHREIBER, 2011), (HABICH, 2011) e (SCHONHERR, 2011) utilizaram tal métrica. Em (FEICHTINGER, 2011) os autores afirmam que o desempenho de implementações do *LBM* são medidos em *MLUPS* (*million lattice cell updates per second*).

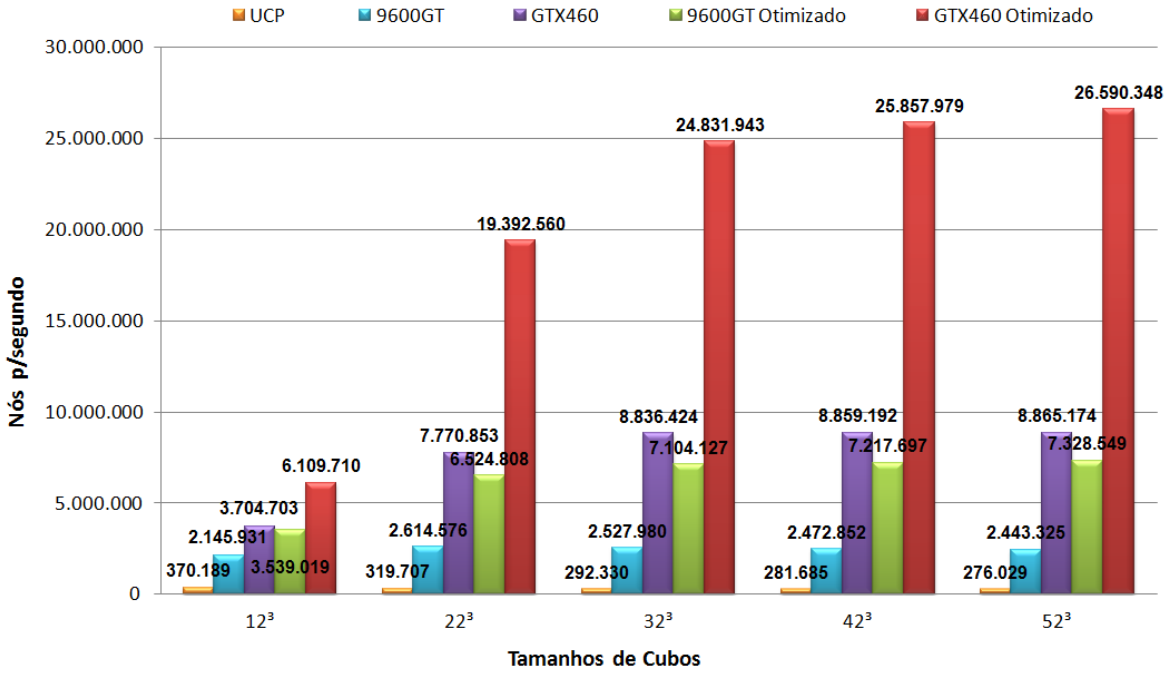


FIG. 7.3: Atualização de nós por segundo

Na figura 7.3 observa-se que ambas as implementações paralelas obtiveram ganhos expressivos em relação a implementação sequencial.

Nota-se que na implementação sequencial, quanto maior a quantidade de dados a

ser processado, menor é a quantidade de *Nodes* atualizada por segundo. Em todas as iterações de cada tamanho de cavidade os passos executam a mesma equação sobre a mesma quantidade de *Nodes*, como a UCP esconde a latência de memória com cache de dados, é lícito supor que o motivo para tal queda no desempenho seja o fato da memória cache do processador ter atingindo o seu limite de armazenamento, aumentando a latência no acesso aos dados armazenados na memória principal.

Nota-se que na implementação sequencial, quanto maior a quantidade de dados a ser processado, menor é a quantidade de *Nodes* atualizada por segundo. Tomando como exemplo um tamanho de cavidade, cada passo executa a mesma equação sobre a mesma quantidade de *Nodes* iteração após iteração, ocorrendo de forma igual para os demais tamanhos de cavidade. Sabendo que a UCP esconde a latência de memória com cache de dados, com o aumento do tamanho da cavidade e consequente aumento na quantidade de *Nodes* a ser computado por um passo, é lícito supor que o motivo para tal queda no desempenho seja o fato da memória cache do processador ter atingindo o seu limite de armazenamento, aumentando a latência no acesso aos dados armazenados na memória principal.

Para a implementação paralela executada na *GPU 9600 GT* sem otimização ocorreram quedas de desempenho. Mesmo a *GPU* escondendo a latência de memória com processamento, inúmeras requisições de memória podem impactar no desempenho da simulação. O mesmo não aconteceu para a versão com otimização de memória executada na mesma *GPU 9600 GT*, que apresentou ganhos de desempenho modestos com o aumento da malha de entrada. A partir do tamanho 22^3 os ganhos de desempenho foram menores do que 10%, e diminuindo a cada novo tamanho de malha simulado. A utilização da memória constante leva a crer que a diminuição na latência de memória favoreceu a implementação, havendo uma certa disposição de estagnação no ganho de desempenho.

Quando submeteu-se a versão paralela sem otimizações a *GPU GTX 460* não ocorreram quedas no desempenho. Entretanto, nota-se a tendência de que essa situação pode não permanecer, caso maiores massas de dados sejam utilizadas. Na implementação otimizada mais uma vez a utilização da memória constante melhorou o desempenho e, devido a superioridade da *GPU GTX 460* sobre a *GPU 9600 GT*, ganhos ainda maiores foram obtidos.

Embora as *GPUs* tenham obtido ganhos expressivos ainda mais em suas versões com otimização, é possível notar uma certa estabilização no ganho de desempenho. Supõe-se que se maiores massas de dados forem submetidas, as *GPUs* podem ter seus limites de

processamento alcançados.

No trabalho executado por (BAILEY, 2009) os autores conseguiram atingir a marca de 300 milhões de *Nodes* atualizados por segundo, e utilizaram uma *GPU 9800 GX2* com 256 *CUDA cores* de 1500 *MHz* e um *lattice D3Q19*, mesmo modelo de *lattice* utilizado neste trabalho. Eles fizeram uso da memória compartilhada da *GPU*, que é menos latente do que a memória constante, juntamente com outra abordagem de implementação para a condição de contorno denominada *bounce back*. Tal abordagem devolve a mesma quantidade de distribuições que atingem os *Nodes* de parede para os vizinhos de fluido de origem. Embora menos dispendioso ao processador do que o esquema de interpolação executado neste trabalho, o *bounce back* pode gerar inconsistências em simulações com superfícies irregulares, pois não utiliza da mesma forma a distribuição de equilíbrio existente na interpolação.

No segundo aspecto apontado para avaliação, foi analisado o quanto cada passo iterativo consome do processamento total do *LBM*. As figuras 7.4, 7.5 e 7.6 ilustram essa distribuição para as respectivas implementações sequencial, paralela e paralela com otimização.

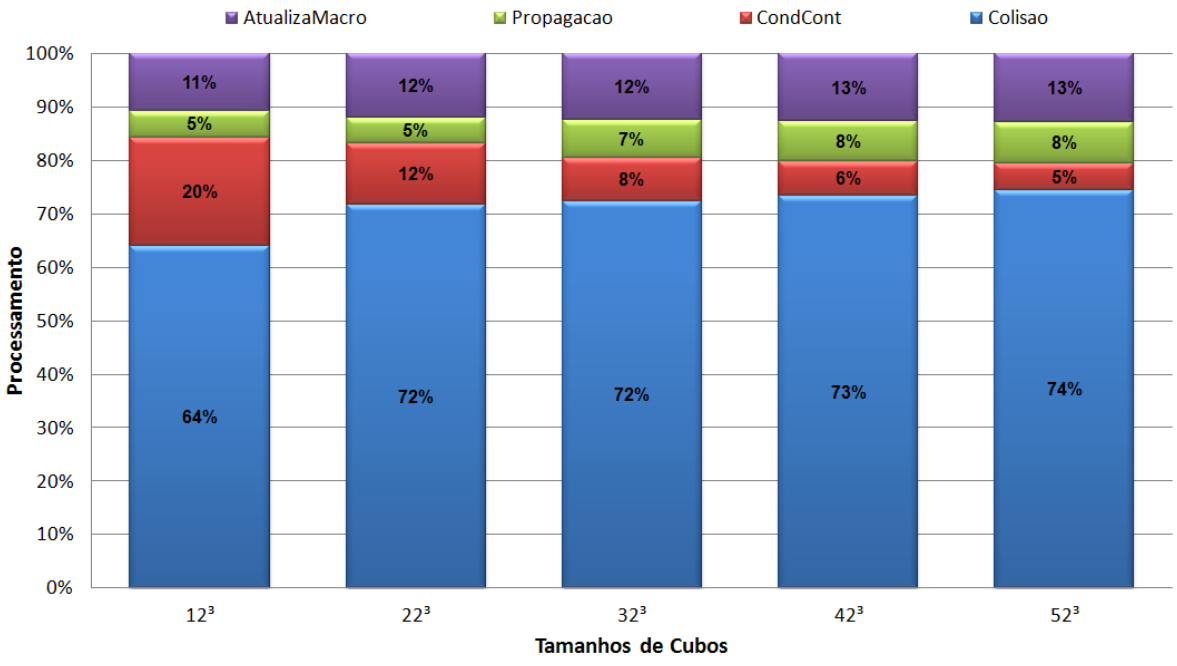


FIG. 7.4: Percentual de processamento das funções do *LBM* para implementação sequencial

Observando a figura 7.4 que representa a implementação sequencial, devido a cavidade possuir a maioria dos *Nodes* de fluido, retratado na tabela 7.2, o passo de colisão,

que é aplicado exclusivamente aos *Nodes* de fluido, representa 64% do processamento para o menor tamanho de cavidade 12^3 e 74% para o maior 52^3. Em que pese o fato do passo de propagação ser aplicado a todos os *Nodes*, e o passo de atualização de valores macroscópicos ser aplicado aos *Nodes* de fluido, os procedimentos neles executados envolvem menores quantidades de cálculos do que os passos de colisão e condição de contorno.

Nessa avaliação, ambas as implementações paralelas sem otimização apresentaram distribuições similares no consumo de processamento dos passos iterativos, ocorrendo da mesma forma entre as versões paralelas com otimização. Por este motivo, será ilustrada na figura 7.5 a comparação das implementações considerando a ausência de otimizações e a figura 7.6 a comparação das implementações com a utilização de otimizações.

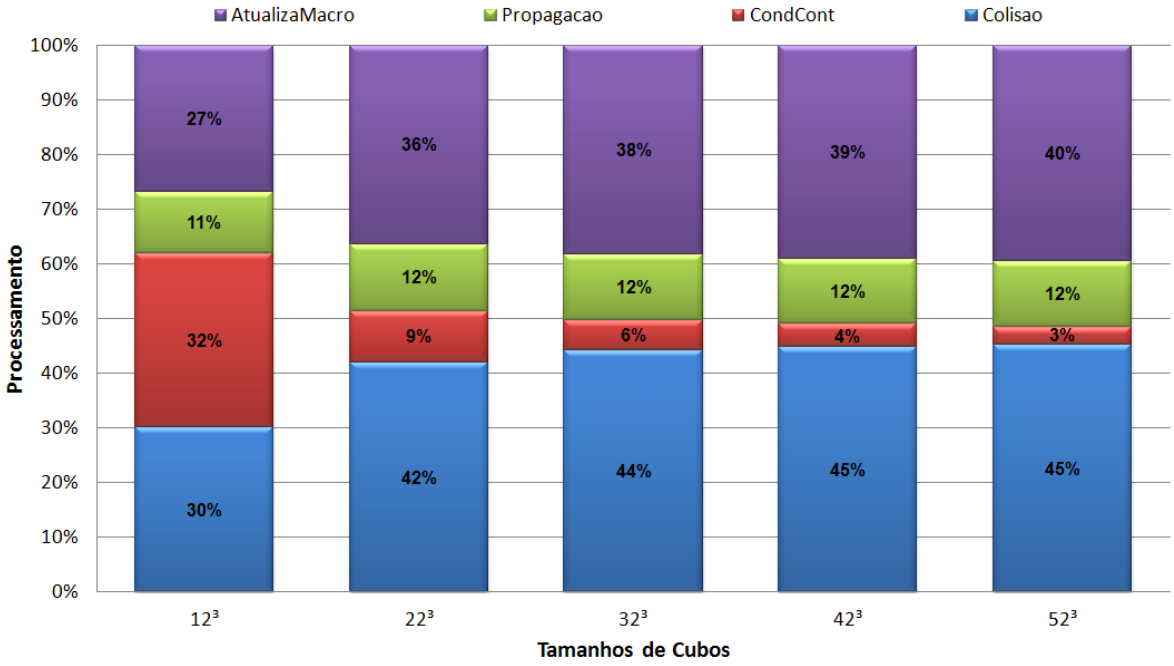


FIG. 7.5: Percentual de processamento das funções do *LBM* para implementação paralela

Observa-se na figura 7.5 que na implementação paralela sem otimizações, os passos de propagação e atualização dos valores macroscópicos se tornarem mais evidentes, sendo lícito supor que a utilização da memória global tenha causado esse maior consumo.

Analisando as implementações paralelas com otimização de memória na figura 7.6, e considerando que os principais beneficiados pela otimização foram os passos de colisão e condição de contorno, discutidos na seção 6.6, tais passos tiveram o consumo de processamento reduzido.

Ao se minimizar os acessos a memória global, o passo de propagação aparece como o

que mais consome processamento. Pretende-se mostrar com isso que o gargalo que influencia o processamento neste caso são os acessos a memória global. O passo de atualização de valores macroscópicos utiliza dois dos dados copiados para a memória constante para atualizar cada *Node* de fluido, provocando uma pequena redução no consumo de processamento.

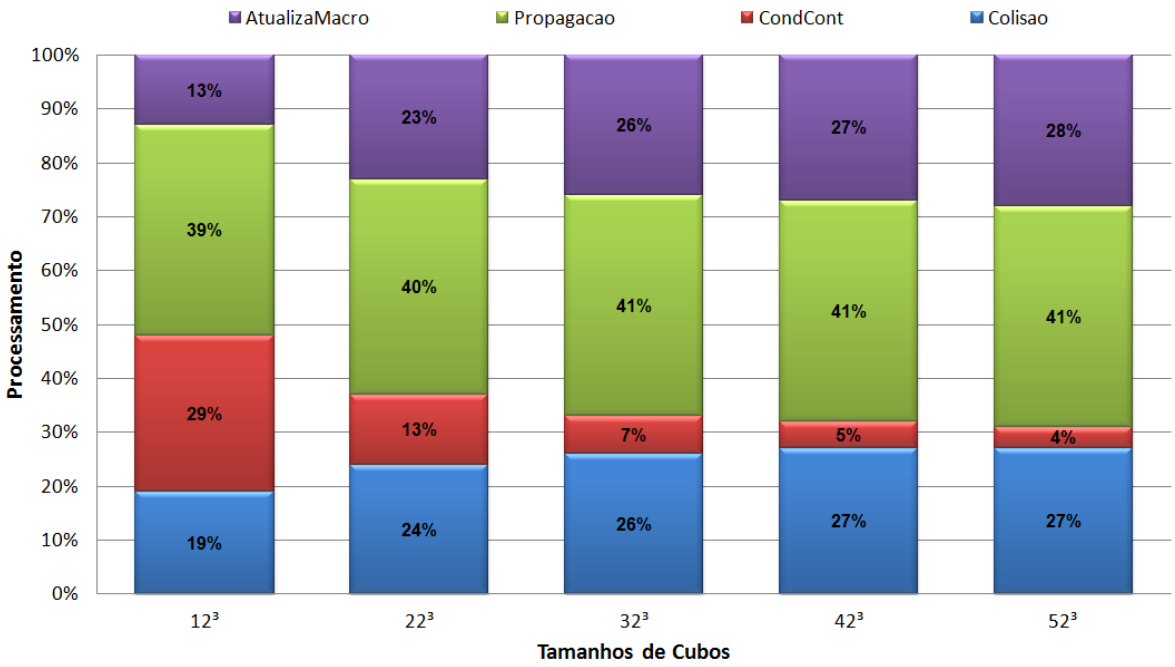


FIG. 7.6: Percentual de processamento das funções do *LBM* para implementação paralela com otimização de memória

No trabalho de (KUZNIK, 2010) os autores afirmam que 70% do processamento é consumido pela colisão e 30% pela propagação. Embora a implementação dos autores seja diferente da presente neste trabalho, pois escolheram o bounce back como condição de contorno e o *D2Q9* como modelo de lattice, eles ressaltaram a importância da utilização das memórias de menor latência e a utilização de acessos coalescentes como fator impactante para atingir um alto desempenho computacional. Nos experimentos os autores utilizaram uma *GPU GTX 280* com 240 *CUDA cores* de 1296 *MHz* fazendo comparações entre implementações com precisão simples e precisão dupla, mostrando um *speedup* de 3.8 vezes na implementação com precisão simples em relação a com precisão dupla.

No terceiro aspecto apontado para avaliação, foi analisado o *Speedup*. A figura 7.7 mostra o resultado das implementações paralelas utilizando *GPU*, principalmente quando são utilizadas as memórias de menor latência.

Considerando a tabela 7.4, embora as *GPUs* tenham teoricamente capacidades de

processamento superiores a de um núcleo da UCP *Intel Quad Core Q9450*, 39 vezes para a *GPU 9600 GT* e 170 vezes para a *GPU GTX 460*, percebe-se na figura 7.7 que o *speedup* teórico não é alcançada, mesmo nas versões otimizadas. Esse fato leva a crer que mesmo utilizando otimizações, a memória exerceu grande influência na escalabilidade da simulação.

Hardware	*Cores*	*Clock*	*Cores* * *Clock*	*Speedup* teórico
Intel Quad Core Q9450	1	2660 *MHz*	2660 *MHz*	
GeForce 9600 GT	64	1625 *MHz*	104000 *MHz*	39,10
GeForce GTX 460	336	1350 *MHz*	453600 *MHz*	170,53

TAB. 7.4: Comparativo de poder de processamento e *Speedup* teórico das *GPUs*. Apesar do processador *Intel Quad Core Q9450* possuir 4 núcleos, é utilizado somente um deles

O maior *Speedup* alcançado pela versão paralela otimizada, submetida a *GPU* de maior desempenho, atingiu a marca de 96,33 vezes sobre a implementação sequencial. Comparando as duas implementações paralelas otimizadas, a *GPU GTX 460* atingiu a marca de 3.6 vezes sobre a *GPU 9600 GT*, ilustrados na figura 7.7.

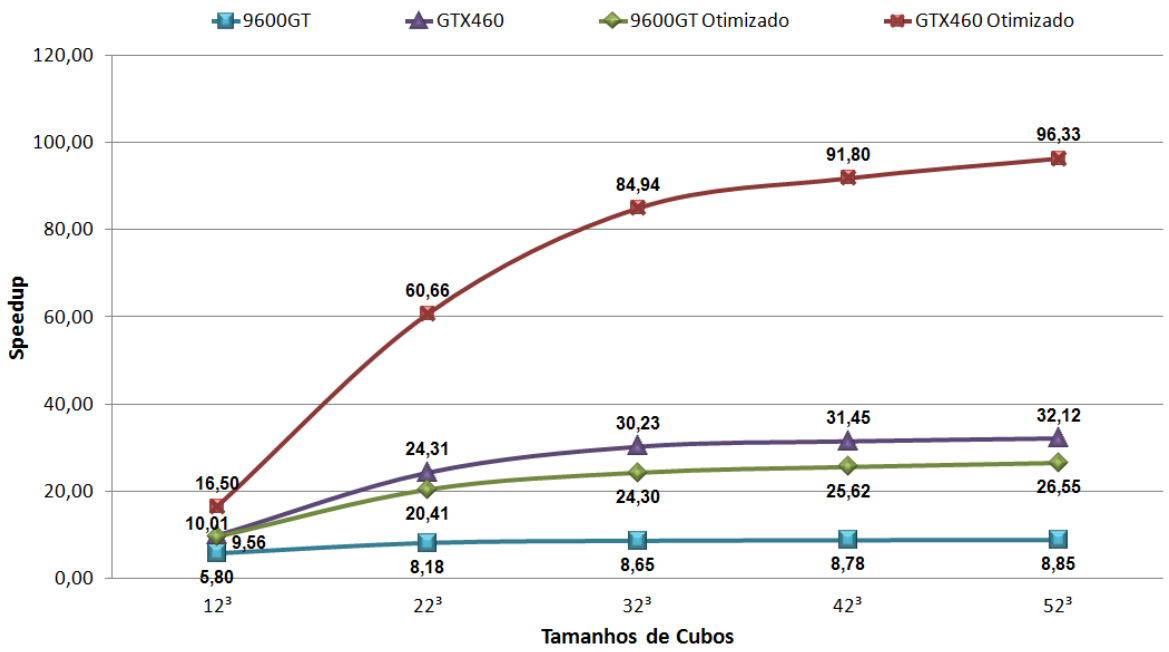


FIG. 7.7: *Speedup* das implementações paralelas sobre a implementação sequencial.

Na implementação do *LBM* apresentada por (BAILEY, 2009), os autores utilizaram uma *GPU 9800 GX2* com 256 *CUDA cores* de 1500 *MHz* e obtiveram um *speedup* de 40 vezes sobre uma UCP *Intel Quad Core Q6600* de 2.4 *GHz* com quatro núcleos, na qual apenas um núcleo foi utilizado e 27 vezes sobre a mesma UCP utilizando seus 4 núcleos.

A abordagem do *LBM* utilizada pelos autores é diferente da executada nesta dissertação, mas fazendo um comparativo entre os speedups alcançados, apesar da *GPU 9800 GX2* possuir uma capacidade de processamento[12] que equivale a 84% da capacidade da *GPU GTX 460*, exposta na tabela 7.4, o maior speedup atingido pelo experimento dos autores equivale a 41% do maior speedup apresentado na figura 7.7 que é de 96,33%.

O *LBM* faz uso intenso de processamento e de memória (GUO, 2009), (GOLBERT, 2009b) e (KUZNIK, 2010). Um fator importante para atingir o máximo do poder de processamento de uma *GPU* é utilizar as memórias de menor latência na hierarquia de memória, evitando o acesso a memória global (NVIDIA, 2009).

Apesar da arquitetura das *GPUs* mascarar a latência da memória, gerenciando *warps* de modo a manter os *Stream Processors* sempre ocupados com processamento, ainda assim a utilização da memória global causa impactos negativos no desempenho.

7.4 MODELO DA ARTÉRIA

Neste modelo, a simulação ocorre em uma artéria extraída de uma imagem médica, que simula o escoamento de um fluido de uma extremidade para outra. Esse experimento mostra a capacidade do LBM em simular a dinâmica do fluido em geometrias mais próximas das reais.

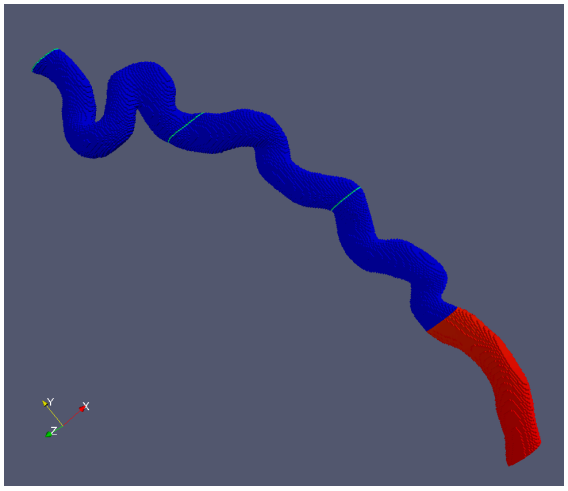
Tanto a tampa inferior quanto a superior foram configuradas com velocidade constante $\vec{u} = 0, 1, 0$. Isso simula a injeção de fluido na tampa inferior e ejeção na superior. As demais paredes são configuradas com velocidade nula $\vec{u} = 0, 0, 0$. A medida em que a tampa inferior injeta o fluido, ele se desloca em direção a tampa superior.

Foram feitas réplicas de pedaços da artéria, gerando 3 partes de tamanhos menores mais a malha completa, para poder variar o tamanho da entrada de dados.

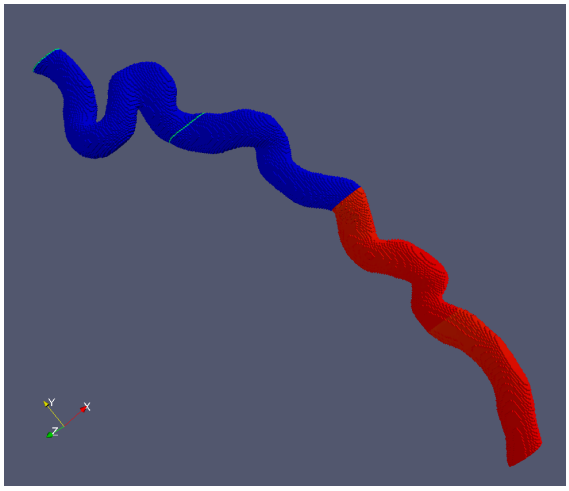
A metodologia para atingir tal variação é ilustrada na figura 7.8. Em 7.8(a) a parte em vermelho forma a primeira malha denominada *M1*. Em 7.8(b) é formada outra malha denominada *M2*, em 7.8(c) a malha denomina-se *M3*, e 7.8(d) representa a malha completa chamada de *MC*.

A tabela 7.5 caracteriza em números a variação nos tamanho das malhas. São exibidos o número de *Nodes* de fluido, o número de *Nodes* de parede, e o percentual de cada tipo em relação ao número total de *Nodes*. Nota-se que praticamente em todas as malhas de entrada, a proporção de *Nodes* de fluido é de 69% contra 31% de *Nodes* de parede em relação a quantidade total de *Nodes*. A única exceção é a malha *M1*, mas a variação é de

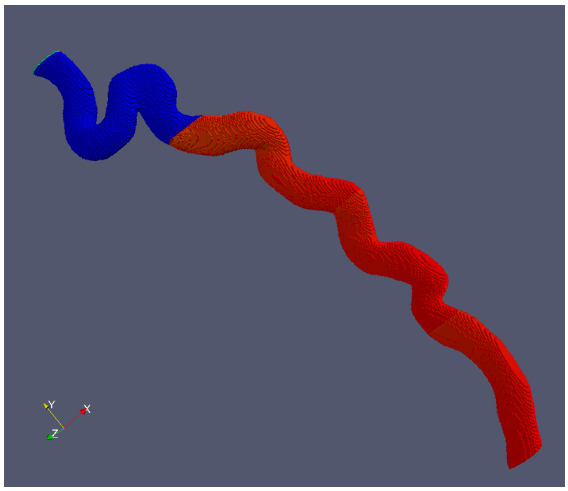
[12] cores * clock = 256 * 1500 = 384000



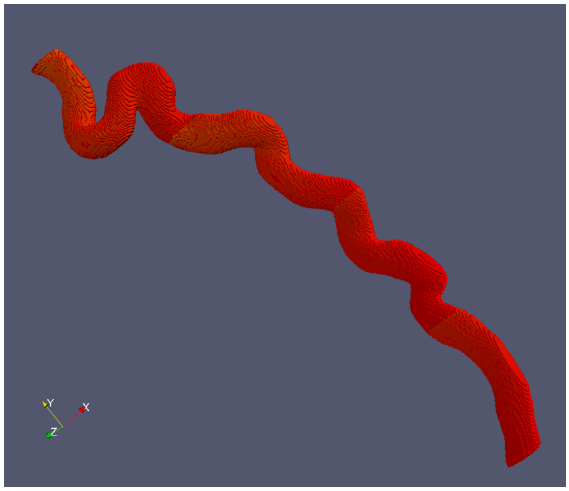
(a) Malha 1



(b) Malha 2



(c) Malha 3



(d) Malha completa

FIG. 7.8: Variação da entrada de dados, com 3 sub-partes mais a artéria completa.

apenas 1% a favor dos *Nodes* de parede. Essas proporções constantes são diferentes das proporções variáveis apresentadas pelo modelo da cavidade, ilustrado na tabela 7.2.

Para visualização da malha resultante, escolheu-se a malha *MC* por representar a malha completa, expondo assim toda a dinâmica do fluido no decorrer da artéria.

De acordo com a figura 7.9(a), pode ser vista a artéria com os valores escalares associados aos *Nodes*, que definem seus tipos. Os *Nodes* de parede, do tipo 0, são apresentados com uma certa transparência, com a intenção de possibilitar a visualização dos *Nodes* internos, do tipo 1. Para mostrar as tampas, dos tipos 2 e 3, foram anexados outros pontos de visualização, expondo as tampas superior e inferior.

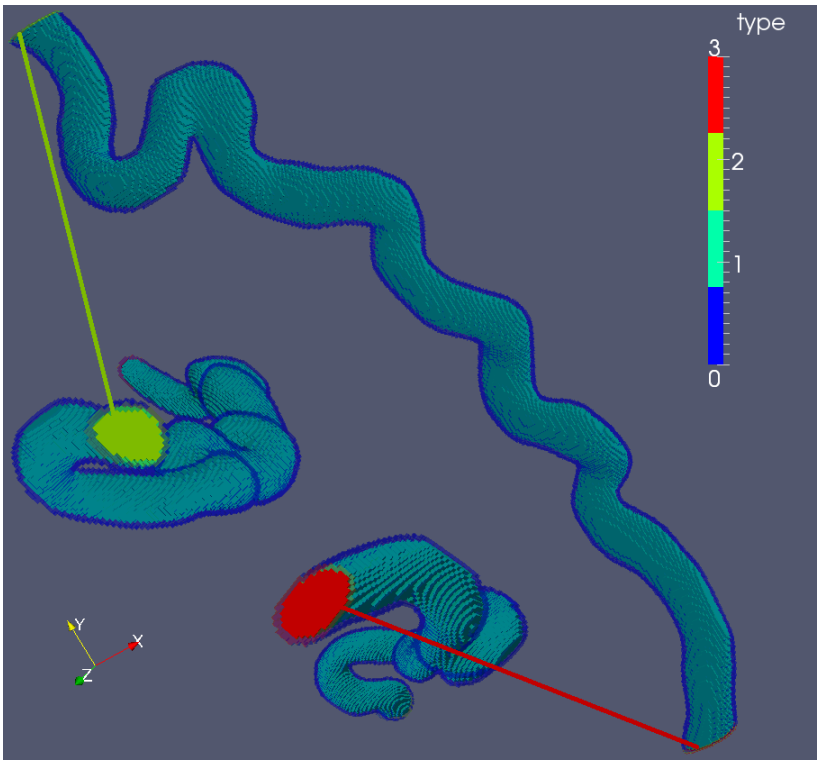
A forma de visualização utilizada na figura 7.9(b) mostra a direção do fluxo, na qual é possível perceber que no centro da artéria a velocidade é maior, e quanto mais próximo da parede menor a velocidade, devido ao atrito com a parede.

Tamanhos	*Nodes* de Paredes	*Nodes* de Fluido	% Parede	% Fluido
M1	8.392	17.715	32%	68%
M2	18.229	40.598	31%	69%
M3	28.666	64.331	31%	69%
MC	42.551	94.528	31%	69%

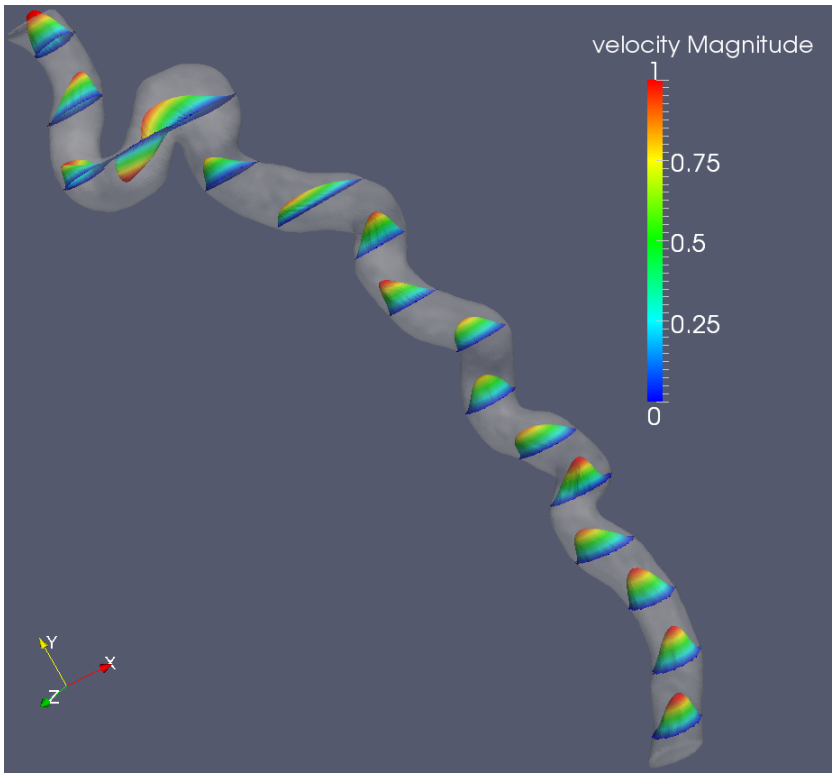
TAB. 7.5: Características das malhas geradas.

Linhas de corrente foram inseridas na figura 7.10(a) que seguem o fluxo apontado pela figura 7.9(b), na qual pode-se perceber que o fluxo em determinadas regiões é mais veloz do que em outras, devido imposição da geometria.

Na forma de visualização ilustrada pela figura 7.10(b) o aumento do raio das linhas de corrente, proporciona a ideia de como o fluxo preenche o corpo interno da artéria como um todo.

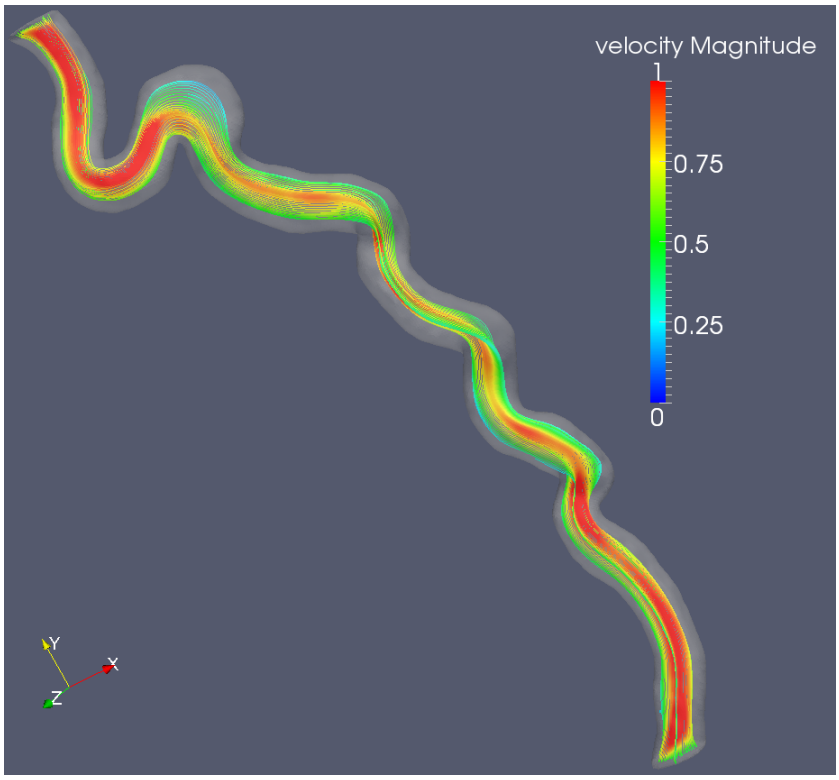


(a) Visualização dos tipos de nós da artéria.

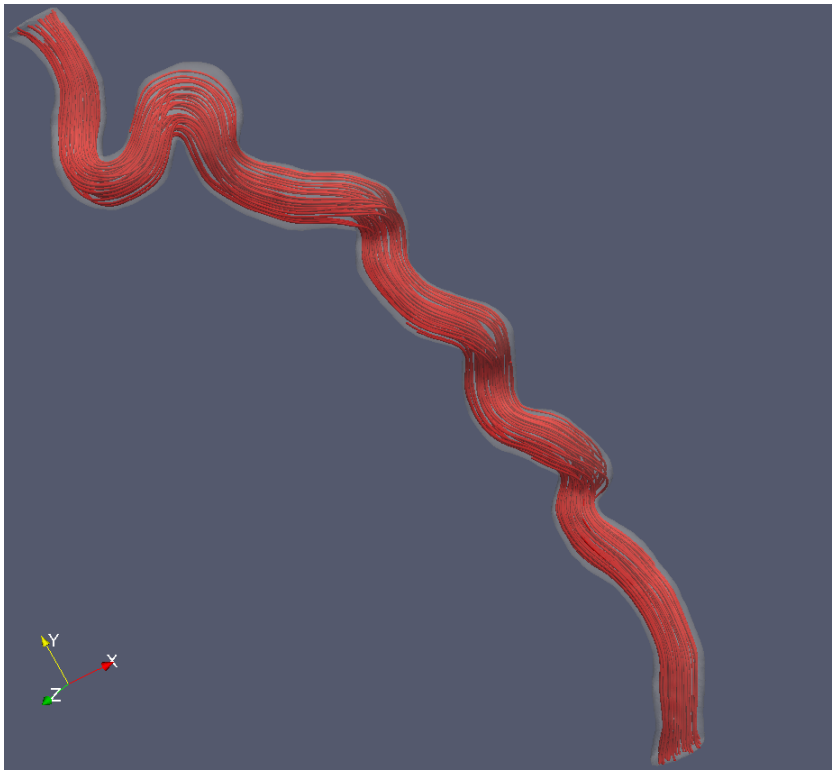


(b) Visualização da direção do escoamento do fluido.

FIG. 7.9: Formas de visualização da artéria.



(a) Visualização com linhas de corrente.



(b) Visualização com linhas de corrente com raio aumentado.

FIG. 7.10: Formas de visualização da artéria com linhas de corrente.

A metodologia dos experimentos segue o mesmo padrão da cavidade, trocando apenas as malhas de entrada pelas apresentadas nas figuras 7.8 (a), (b), (c) e (d). Foram feitas 5 simulações com 3000 iterações para cada uma das 4 malhas e as medições foram feitas em segundos. Foram armazenados os tempos de execução de cada passo iterativo e utilizadas suas médias para geração dos gráficos, que podem ser vistas na tabela 7.6.

Tamanhos	UCP	*9600 GT*	*9600 GT* Otmz.	*GTX 460*	*GTX 460* Otmz.
M1	228,75	25,37	10,73	8,59	3,63
M2	542,83	56,87	23,11	18,98	7,71
M3	886,77	90,03	35,96	29,20	11,67
MC	1282,28	132,49	51,02	42,42	16,33

TAB. 7.6: Média em segundos dos tempos de execução da artéria.

De forma similar à cavidade, o tempo gasto com alocação de memória e transferência de dados da memória principal da UCP para a memória global da *GPU*, são também inferiores a 5 centésimos de segundo para o maior caso. Portanto, não serão avaliadas questões de transferência de dados de entrada e saída para a *GPU*.

Na figura 7.11, são apresentados os números de *Nodes* que são atualizados por segundo por cada implementação. Observa-se que todas as implementações paralelas superam a implementação sequencial neste quesito. Na sequencial, quanto maior a quantidade de dados a ser processado, menor é a quantidade de *Nodes* atualizada. Este fato já visto na análise da cavidade leva a crer que, como em todas as iterações os passos executam a mesma equação sobre a mesma quantidade de *Nodes*, supõe-se que tal queda no desempenho seja a latência de memória.

Para a implementação paralela sem otimização executada na *GPU 9600 GT*, houve uma pequena oscilação entre o número de atualizações por segundo em todas as malhas de entrada, menos de 5% a cada nova malha. É lícito supor que as inúmeras requisições de memória impactaram no desempenho da simulação, ainda que a *GPU* possua em sua arquitetura procedimentos para gerenciar a utilização massiva de processamento, escondendo assim a latência de memória.

Na versão paralela otimizada submetida à *GPU 9600 GT*, a utilização da memória constante, de menor latência, teve moderados ganhos de desempenho nos tamanhos de entrada de dados, mas em comparação com a versão sem otimizações submetida à mesma *GPU* a otimização proporcionou um desempenho 2 vezes superior.

Na submissão da versão sem otimização à *GPU GTX 460*, analisando pela variação nas

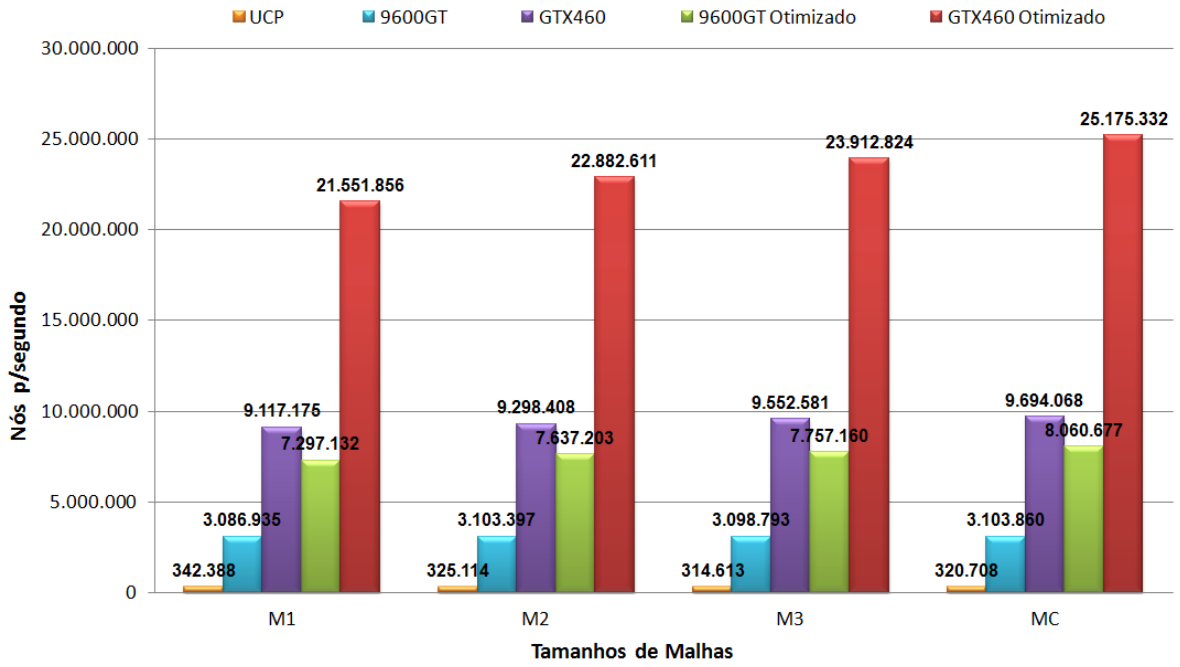


FIG. 7.11: Atualização de nós por segundo

entradas de dados os ganhos foram modestos, menos de 2% a cada nova malha de entrada, deixando exposta a tendência de estagnação no ganho de desempenho. Na implementação otimizada submetida a mesma *GPU* os ganhos de desempenho com a variação dos dados de entrada foram menos tímidos, menos de 6% a cada nova malha simulada. A utilização da memória constante proporcionou maiores ganhos de desempenho se comparada à versão sem otimização submetida à mesma *GPU*, mas da mesma forma se mostrou disposta a estabilizar o ganho de desempenho.

Apesar das *GPUs* apresentarem os maiores desempenhos, se maiores massas de dados forem utilizadas as *GPUs* podem ter seus limites de processamento alcançados.

Conforme o segundo aspecto de avaliação, serão apresentados resultados sobre o percentual de utilização do processamento pelos passos iterativos, e para a análise, as figuras 7.12, 7.13 e 7.14 apresentam respectivamente as implementações sequencial, paralela e paralela com otimização.

Na figura 7.12, que representa a implementação sequencial, como a maioria dos *Nodes* a serem comutados são do tipo fluido, e o passo que executa a maior quantidade de cálculos sobre esse tipo de *Node* é o de colisão, este passo ocupa a fatia de 65 a 67% do processamento total.

No experimento da artéria, todas as malhas possuem uma regularidade na proporção de *Nodes* de fluido e *Nodes* de parede em relação ao número total de *Nodes*, conforme

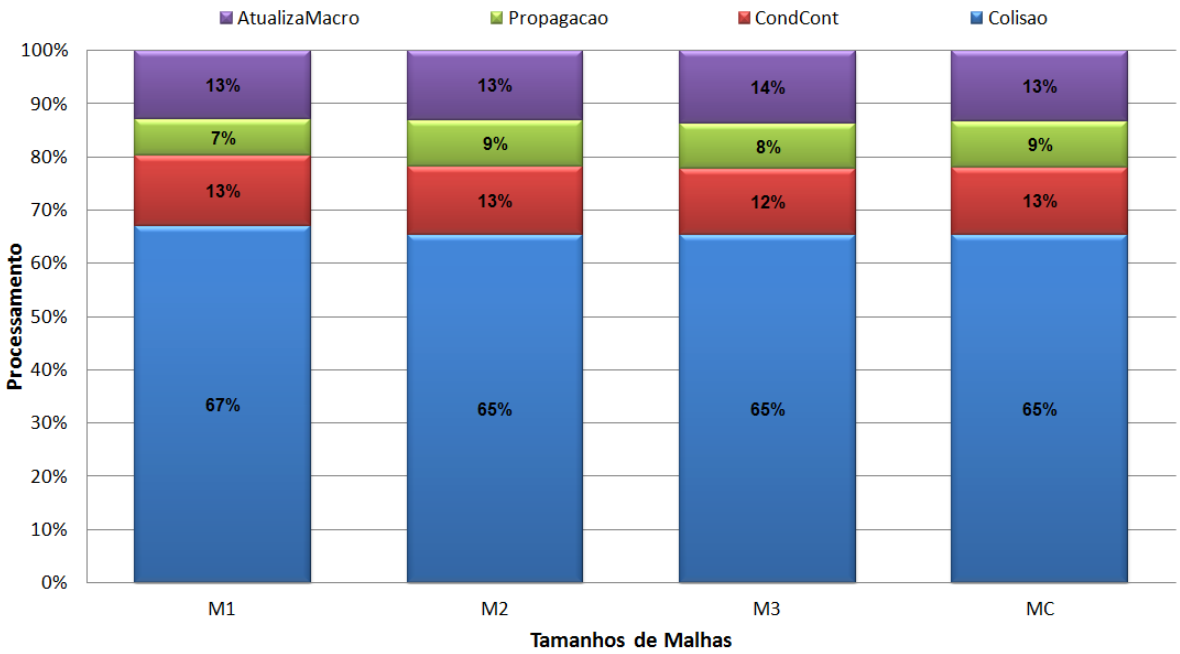


FIG. 7.12: Percentual de processamento das funções do *LBM* para implementação sequencial

apresentado na tabela 7.5, ao contrário do experimento da cavidade exposto na tabela 7.2. Essa regularidade é refletida na divisão do processamento pelos passos iterativos, uniforme para todas as malhas de entrada.

A exemplo do modelo da cavidade, ambas as implementações paralelas sem otimização obtiveram fatias de processamento divididas de forma similar, ocorrendo da mesma forma entre as versões paralelas com otimização. Serão ilustradas, portanto, a mesma figura para avaliar as implementações sem otimizações e outra com otimizações.

Para as implementações paralelas sem otimização ilustradas na figura 7.13, os passos de propagação e atualização dos valores macroscópicos apresentaram fatias maiores do que as apresentadas na implementação sequencial. Isto leva a crer que a utilização da memória global tenha causado esse maior consumo.

Como discutido na seção 6.6, os passos mais beneficiados pela utilização de memória constante foram os de colisão e condição de contorno e, por conseguinte, obtiveram o consumo de processamento reduzido.

O surgimento do passo de propagação com a maior fatia de processamento deixa exposta a recorrência da memória global por tal passo, e a redução modesta na fatia de processamento do passo de atualização de valores macroscópicos, ocorre devido a utilização de dois dos dados copiados para a memória constante para atualizar os *Nodes*.

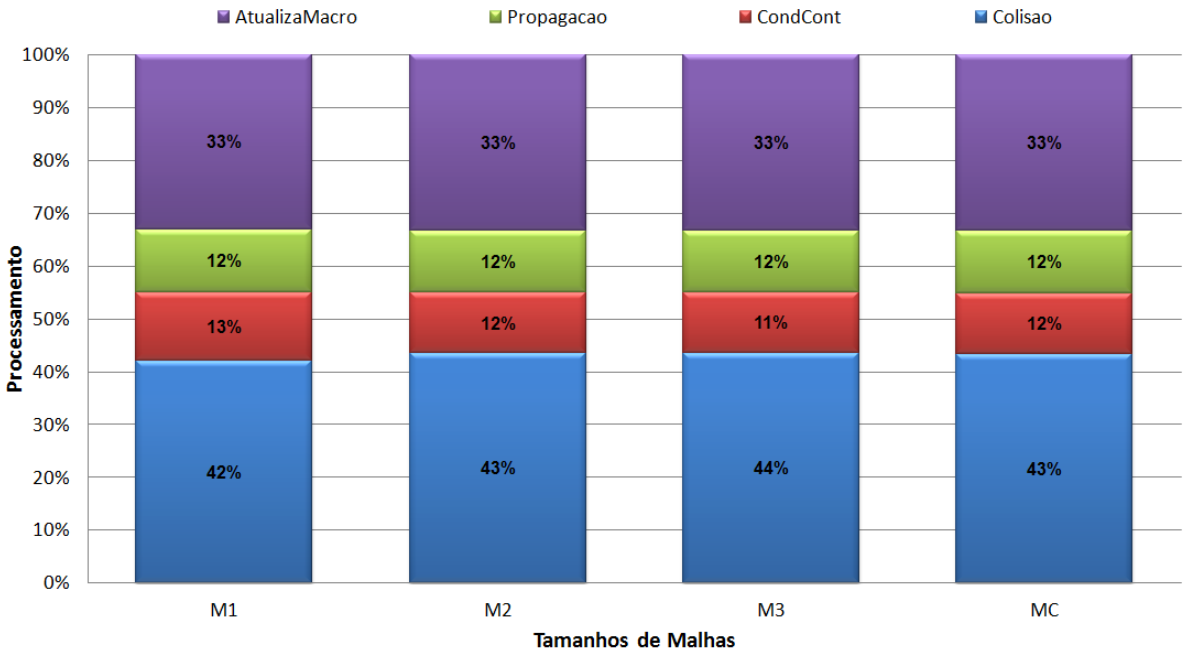


FIG. 7.13: Percentual de processamento das funções do *LBM* para implementação paralela

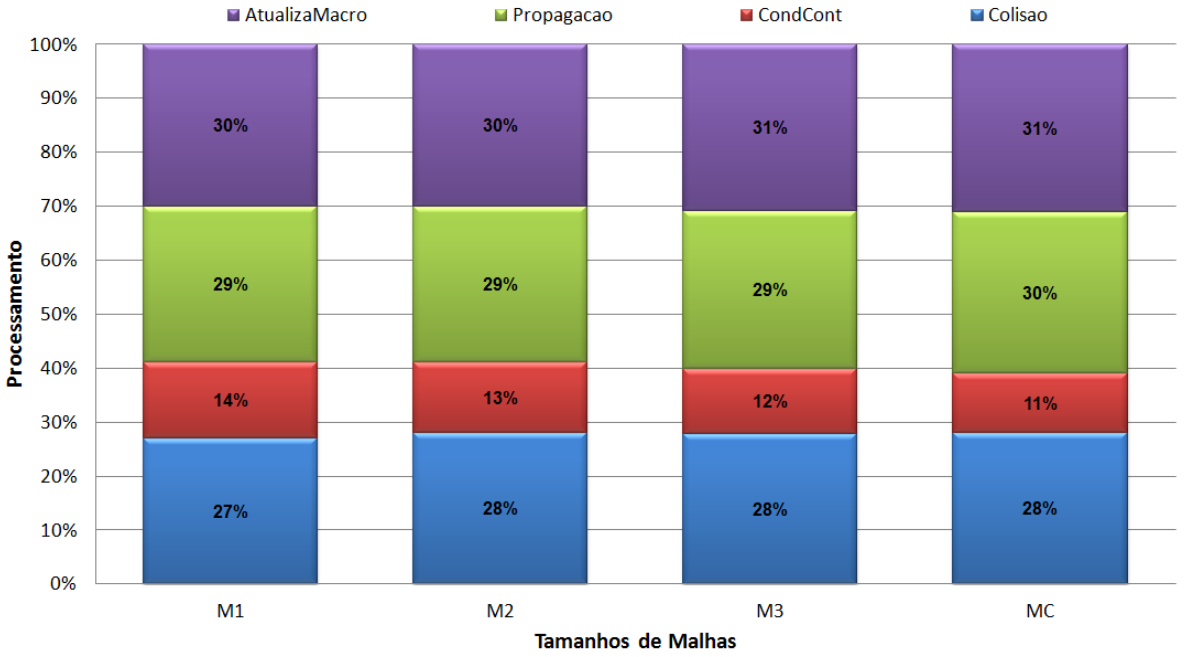


FIG. 7.14: Percentual de processamento das funções do *LBM* para implementação paralela com otimização de memória

A figura 7.15 ilustra o *Speedup* das implementações paralelas sobre a implementação sequencial, e mostra a relevância das implementações paralelas, principalmente as que implementam otimizações envolvendo a utilização das memórias de menor latência.

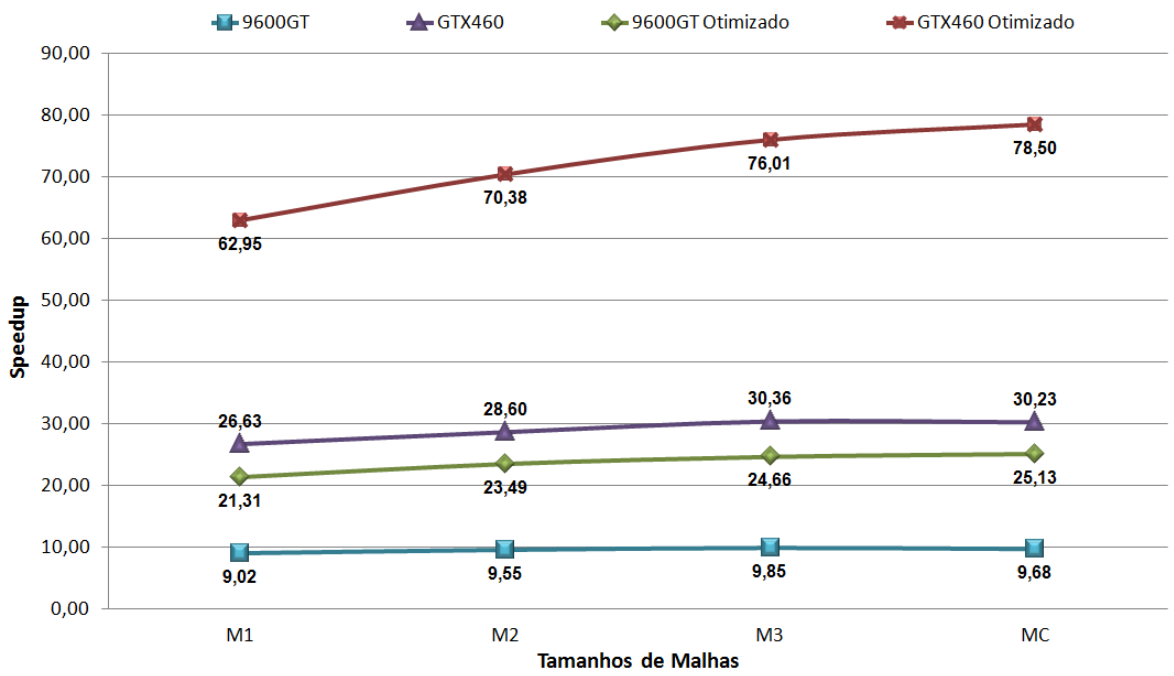


FIG. 7.15: *Speedup* das implementações paralelas sobre a implementação sequencial.

Para esta análise serão utilizados os dados da tabela 7.4, que expõem as capacidades de processamento da UCP *Intel Quad Core Q9450*, da *GPU 9600 GT* e da *GPU GTX 460*, assim como o *Speedup* que teoricamente as *GPUs* deveriam obter sobre a UCP.

As *GPUs* possuem teoricamente capacidades de processamento significativamente superiores a da UCP, 39 vezes para a *GPU 9600 GT* e 170 vezes para a *GPU GTX 460*, mas na prática o *Speedup* máximo alcançado é de 25,13 vezes para a *GPU 9600 GT* e 78,50 vezes para a *GPU GTX 460*, como pode ser visto na figura 7.7.

Fazendo uma comparação adicional entre as implementações paralelas sem otimização e entre as com otimização, a *GPU GTX 460* teoricamente deveria obter o desempenho 4,36 vezes superior ao*9600 GT*, tendo apresentado em média 3 vezes apenas.

Dentre os fatores que afetam o desempenho da *GPU* já analisadas, como a importância de utilizar as memórias de menor latência evitando o acesso a memória global (NVIDIA, 2009), outros fatores caros para todas as implementações apresentadas, são as características do *LBM* de ser computacionalmente caro e exigente de memória como indicado por (GUO, 2009), (GOLBERT, 2009b) e (KUZNIK, 2010).

8 CONCLUSÕES E TRABALHOS FUTUROS

8.1 CONCLUSÕES

O *LBM* é um método numérico da modelagem matemática e simulação computacional. Ele possui como vantagens a possibilidade de divisão do algoritmo principal em passos menores, e a característica de localidade presente em todos os passos, possibilitando a implementação paralela do algoritmo. Tais vantagens são significativas quando aliadas ao alto poder de processamento paralelo das *GPUs* atuais, principalmente quando pretende-se atravessar os obstáculos criados pelo alto custo computacional envolvido nas aplicações no âmbito científico e tecnológico.

Este trabalho demonstrou como a utilização de placas gráficas para a computação de propósito geral pode oferecer ganhos de desempenho para aplicações que envolvem a dinâmica de fluidos, através do desenvolvimento de um resolvedor numérico baseado no método de *Lattice Boltzmann LBM* em unidades de processamento gráfico utilizando a arquitetura *CUDA*.

Os objetivos propostos foram atingidos com a realização da nova abordagem do *LBM*, que foi desenvolvida combinando o uso de diferentes técnicas estudadas na literatura. Destacam-se como técnicas importantes para o desenvolvimento deste trabalho, o procedimento de decomposição da matriz que favorece a coalescência dos dados na memória, a técnica de *swap* que promove trocas entre posições de memória sem sobrescrita de dados, e por último a escolha correta dos dados que foram copiados para uma memória de menor latência na hierarquia de memória da GPU.

O procedimento de decomposição da matriz e a técnica de *swap*, não foram analisados como fatores impactantes para o ganho de desempenho, pois o foco do trabalho está no aproveitamento da arquitetura das *GPUs*.

A utilização de memória constante aliada a escolha correta dos dados nela carregados, mostrou que a utilização das memórias de menor latência podem impactar no desempenho da simulação como um todo, sendo lícito supor que ao elaborar outros esquemas de partição dos dados, de maneira que os mesmos utilizem as demais memórias de baixa latência na hierarquia de memórias da *GPU*, ganhos ainda maiores podem ser alcançados.

As análises realizadas mostraram o potencial da arquitetura de software *CUDA* em conjunto com a arquitetura de hardware das *GPUs*, estimulando a realização de novos

trabalhos direcionados a essas arquiteturas.

Os diferentes experimentos apresentados demonstraram que as placas gráficas podem proporcionar ganhos de desempenho consideráveis. Foi mostrado ainda que, com a utilização das memórias de menor latência na hierarquia de memória das *GPUs*, ganhos maiores puderam ser alcançados. No caso da cavidade cúbica, a tabela 7.3 mostrou para o caso de 52^3 *Nodes*, que o tempo foi reduzido de 1528 segundos (equivalente a cerca de 25 minutos) para 15 segundos, o que representa uma redução de 99% no tempo de processamento.

8.2 TRABALHOS FUTUROS

Como sugestões para trabalhos futuros envolvendo o presente trabalho, pretende-se estudar a utilização de memória compartilhada buscando diminuir ainda mais a latência de memória e a colaboração entre os *threads*.

Como outra sugestão, as implementações apresentadas neste trabalho podem ser desenvolvidas e avaliadas em ambientes paralelos que disponham de um número maior de *GPUs* usando, para isso, a alocação de múltiplas *GPU* no mesmo computador, ou em *clusters* de *GPUs*.

Dessa forma, será possível aumentar consideravelmente o tamanho dos *Lattices* que serão utilizados como entrada de dados, e elaborar diferentes esquemas de decomposição, menos granulares do que a apresentada neste trabalho, permitindo a comparação entre tanto os esquemas de decomposição, quanto entre as arquiteturas paralelas utilizadas.

Com base nos resultados deste trabalho, acredita-se que o constante avanço na capacidade de processamento das *GPUs* e a possibilidade de sua utilização em ambientes paralelos, aliado a busca frequente de resultados experimentais neste contexto, podem oferecer à comunidade científica reduções ainda maiores no tempo de espera pelos resultados de trabalhos, que envolvem técnicas de modelagem matemática e simulação computacional.

9 REFERÊNCIAS BIBLIOGRÁFICAS

ARTOLI, A., HOEKSTRA, A. e SLOOT, P. **Mesoscopic simulations of systolic flow in the human abdominal aorta**. *Journal of Biomechanics*, 39(5):873 – 884, 2006. ISSN 0021-9290.

BAILEY, P., MYRE, J., WALSH, S. D. C., LILJA, D. J. e SAAR, M. O. **Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors**. Em *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, págs. 550–557, Washington, DC, USA, 2009. IEEE Computer Society.

BHATNAGAR, P. L., GROSS, E. P. e KROOK, M. **A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems**. *Phys. Rev.*, 94(3):511–525, 1954.

BOHN, C.-A. **Kohonen Feature Mapping through Graphics Hardware**. Em *In Proceedings of Int. Conf. on Compu. Intelligence and Neurosciences*, págs. 64–67, 1998.

BROADWELL, J. E. **Study of rarefied shear flow by the discrete velocity method**. *Journal of Fluid Mechanics*, 19:401 - 414, 1964.

BUICK, J. M. ***Lattice Boltzmann methods in interfacial wave modelling***. Tese de Doutorado, 1997.

CAMARGO, E., KOSTIN, S. e PINTO, R. **A Tool for Scientific Visualization Based on Particle Tracing Algorithm on Graphics Processing Units**. Em *Sistemas Computacionais (WSCAD-SSC), 2011 Simposio em*, oct. 2011.

CARR, N. A., HALL, J. D. e HART, J. C. **The ray engine**. Em *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, págs. 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

CASTANO-DIEZ, D., MOSER, D., SCHOENEGGER, A., PRUGGNALLER, S. e FRANGAKIS, A. S. **Performance evaluation of image processing algorithms on the GPU**. *Journal of Structural Biology*, 164(1):153 – 160, 2008. ISSN 1047-8477.

CERCIGNANI, C. ***The Boltzmann equation and its applications***. Springer-Verlag, New York, 1988.

CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W. e SKADRON, K. **A performance study of general-purpose applications on graphics processors using CUDA**. *Journal of Parallel and Distributed Computing*, 68:1370 – 1380, 2008. ISSN 0743-7315. cited By (since 1996) 2.

CHEN, H., CHEN, S. e MATTHAEUS, W. H. **Recovery of the Navier-Stokes equations using a lattice-gas Boltzmann method**. *Phys. Rev. A*, 45(8):R5339–R5342, Apr 1992.

CHEN, S. e DOOLEN, G. **Lattice Boltzmann method for fluid flows**. *Annual Review of Fluid Mechanics*, 30:329–364, 1998. cited By (since 1996) 1402.

CHOPARD, B., DUPUIS, A., MASSELOT, A. e LUTHI, P. **Cellular Automata And Lattice Boltzmann Techniques: An Approach To Model And Simulate Complex Systems**. *Advances in Complex Systems (ACS)*, 5(02):103–246, 2002.

COLLARES, M., CAMARGO, E., GOLBERT, D., KOSTIN, S. e PINTO, R. C. **Utilização de GPU Para Comparar o Desempenho da Simulação Numérica Computacional e da Visualização Científica do Sistema Cardiovascular Humano.** Em *CILAMCE 2011*, Ouro Preto, Brazil, nov 2011.

DUPUIS, A. ***From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river***. Tese de Doutorado, University of Geneva, June 2002.

FEICHTINGER, C., HABICH, J., KÖSTLER, H., HAGER, G., RÜDE, U. e WELLEIN, G. **A flexible Patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters.** *Parallel Computing*, págs. 536–549, 2011.

FEIJOO, R. A., BLANCO, P. J., ZIEMER, P. G. P., COLLARES, M., DE FREITAS, I. C. e CAMARGO, E. **HeMoLab - Laboratorio de Modelagem em Hemodinamica**.

FLYNN, L. J. **Intel Halts Development of 2 New Microprocessors**. http://www.nytimes.com /2004/05/08/business/08chip.html. Acessado em 01 de outubro de 2009, 05 2004.

FRISCH, U., HASSLACHER, B. e POMEAU, Y. **Lattice-Gas Automata for the Navier-Stokes Equation**. *Physical Review Letters*, 56(14):1505 – 1508, Abril 1986.

GASPAROTO, H. F. e FERREIRA, L. O. S. **Development of a Heuristic Type Generate And Test, Parallel and Random, to Optimize Functions in GPU.** Em *Metodos Numericos e Computacionais em Engenharia - CMNE CILAMCE 2009*, Buzios, Brazil, november 2009.

GOLBERT, D. R., BLANCO, P. J. e FEIJÓO, R. A. **Lattice Boltzmann Simulations In Computational Hemodynamics.**, booktitle = CILAMCE 2009. Buzios, Brazil, nov 2009a.

GOLBERT, D. R. **Modelos de Lattice Boltzmann Aplicados a Simulacao Computacional do Escoamento de Fluidos Incompressiveis**. Dissertação de Mestrado, Laboratorio Nacional de Computacao Cientifica, April 2009b.

GUO, W., JIN, C. e LI, J. **High Performance Lattice Boltzmann Algorithms for Fluid Flows**. Em *Proceedings of the 2008 International Symposium on Information Science and Engieering - Volume 01*, págs. 33–37, Washington, DC, USA, 2008. IEEE Computer Society.

GUO, W., JIN, C., LI, J. e HE, G. **Parallel Lattice Boltzmann Simulation for Fluid Flow on Multicore Platform**. Em *Proceedings of the 2009 WASE International Conference on Information Engineering - Volume 01*, págs. 107–110, Washington, DC, USA, 2009. IEEE Computer Society.

GUO, Z., ZHENG, C. e SHI, B. **An extrapolation method for boundary conditions in lattice Boltzmann method**. *Physics of Fluids*, 14(6):2007–2010, 2002.

HABICH, J., ZEISER, T., HAGER, G. e WELLEIN, G. **Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA**. *Advances in Engineering Software*, 42(5):266 – 272, 2011. ISSN 0965-9978.

HARDY, J., DE PAZZIS, O. e POMEAU, Y. **Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions**. *Phys. Rev. A*, 13(5):1949–1961, May 1976.

HE, X. e LUO, L.-S. **Lattice Boltzmann Model for the Incompressible Navier Stokes Equation**. *Journal of Statistical Physics*, 88(18):927–944, 1997.

HE, X., DUCKWILER, G. e VALENTINO, D. J. **Lattice Boltzmann simulation of cerebral artery hemodynamics**. *Computers & Fluids*, 38(4):789 – 796, 2009. ISSN 0045-7930.

HIRABAYASHI, M., OHTA, M., BAURTH, K., RUFENACHT, D. A. e CHOPARD, B. **Numerical analysis of the flow pattern in stented aneurysms and its relation to velocity reduction and stent efficiency**. *Mathematics and Computers in Simulation*, 72(2-6):128 – 133, 2006. ISSN 0378-4754. Discrete Simulation of Fluid Dynamics in Complex Systems.

HOFF, III, K. E., KEYSER, J., LIN, M., MANOCHA, D. e CULVER, T. **Fast computation of generalized Voronoi diagrams using graphics hardware**. Em *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, págs. 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

HUANG, Q., HUANG, Z., WERSTEIN, P. e PURVIS, M. **GPU as a General Purpose Computing Resource**. Em *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, págs. 151–158, Dec. 2008.

INCT-MACC. **Instituto Nacional de Ciência e Tecnologia em Medicina Asisstida por Computação Científica**. http://macc.lncc.br, 2010.

INTEL. **Microprocessor Quick Reference Guide**, Dezembro 2008. URL `http://www.intel.com/pressroom/kits/quickrefyr.htm#2005`.

KAPASI, U. J., RIXNER, S., DALLY, W. J., KHAILANY, B., AHN, J. H., MATTSON, P. e OWENS, J. D. **Programmable Stream Processors**. *IEEE Computer*, págs. 54–62, agosto 2003.

KEDEM, G. e ISHIHARA, Y. **Brute force attack on UNIX passwords with SIMD computer**. Em *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, págs. 8–8, Berkeley, CA, USA, 1999. USENIX Association.

KIRK, D. B. e HWU, W.-M. W. ***Programming Massively Parallel Processors: A Hands-on Approach***. Morgan Kaufmann, 1 edition, fevereiro 2010.

KITWARE. **ParaView**. http://www.paraview.org, 2009a.

KITWARE. **VTK - Visualization Toolkit**. http://www.vtk.org, 2009b.

KRAFCZYK, M., CERROLAZA, M., SCHULZ, M. e RANK, E. **Analysis of 3D transient blood flow passing through an artificial aortic valve by Lattice-Boltzmann methods**. *Journal of Biomechanics*, 31(5):453 – 462, 1998. ISSN 0021-9290.

KUZNIK, F., OBRECHT, C., RUSAOUEN, G. e ROUX, J.-J. **LBM based flow simulation using GPU computing processor**. *Computers & Mathematics with Applications*, In Press, Corrected Proof:–, 2010. ISSN 0898-1221.

LABAKI, J., FERREIRA, L. O. S. e MESQUITA, E. **Implementation of Quadratic Boundary Elements for 2D Potential Problems on Graphics Hardware - GPU.** Em *Metodos Numericos e Computacionais em Engenharia - CMNE CILAMCE 2009*, Buzios, Brazil, november 2009.

LANDAU, L. D. e LIFSHITZ, E. M. ***Fluid Mechanics***, volume 6. Oxford, United Kingdon, 2nd edition, 1982.

LARRABIDE, I. e FEIJÓO, R. **Hemolab: Laboratório de Modelagem em Hemodinâmica**. Technical Report 13, Laboratório Nacional de Computação Científica, 2006.

LENGYEL, J., REICHERT, M., DONALD, B. R. e GREENBERG, D. P. **Real-time robot motion planning using rasterizing computer graphics hardware**. *SIGGRAPH Comput. Graph.*, 24:327–335, September 1990. ISSN 0097-8930.

LI, J. **Computational Fluid Dynamics. By T. J. CHUNG. Cambridge University Press**. *Journal of Fluid Mechanics*, 491:411–412, 2003.

MATTILA, K., HYVÄLUOMA, J., ROSSI, T., ASPNÄS, M. e WESTERHOLM, J. **An efficient swap algorithm for the lattice Boltzmann method**. *Computer Physics Communications*, 176(3):200 – 210, 2007. ISSN 0010-4655.

MINISTERIO DA SAUDE. **Pratique Saúde**, 2007. URL `http://portal.saude.gov.br/ portal/saude/visualizar_texto.cfm?idtxt= 23615`.

MITTMANN, A., DANTAS, M. e VON WANGENHEIM, A. **Design and Implementation of Brain Fiber Tracking for GPUs and PC Clusters**. Em *Computer Architecture and HPC, 2009. SBAC-PAD '09. 21st International Symposium on*, págs. 101 –108, 28-31 2009.

NEMA, N. E. M. A. **NEMA standards and guideline publications,**. URL `<http://medical.nema.org/dicom/2004.html`. Acessado em 04 de julho de 2011.

NVIDIA. ***CUDA Programming Guide***. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 3.1 edition, 5 2009.

NVIDIA. **CUDA Toolkit 4.0**. http://developer.nvidia.com/cudatoolkit40 acessado em 07/07/2011, 5 2011.

NVIDIA. **Produtos NVidia.** http://developer.nvidia.com/cuda-gpus Acessado em 06 de julho de 2011, 2011.

OBRECHT, C., KUZNIK, F. e TOURANCHEAU. **Multi GPU implementation of the lattice Boltzmann method**. *Computers & Mathematics with Applications*, 2011. ISSN 0898-1221.

PELLICCIONI, O., CERROLAZA, M. e HERRERA, M. **Lattice Boltzmann dynamic simulation of a mechanical heart valve device**. *Mathematics and Computers in Simulation*, 75(1-2):1 – 14, 2007. ISSN 0378-4754.

PURCELL, T. J., BUCK, I., MARK, W. R. e HANRAHAN, P. **Ray tracing on programmable graphics hardware**. *ACM Trans. Graph.*, 21:703–712, July 2002. ISSN 0730-0301.

QIAN, Y. H., D'HUMIÈRES, D. e LALLEMAND, P. **Lattice BGK Models for Navier-Stokes Equation**. *EPL (Europhysics Letters)*, 17(6):479, 1992.

RAABE, D. **Overview on the Lattice Boltzmann Method for Nano and Microscale Fluid Dynamics in Materials Science and Engineering**, 2006. URL `http://www.mpie.de/index.php?id=1208. Acessado em 30 de maio de 2012`.

RIBBROCK, D., GEVELER, M., GODDEKE, D. e TUREK, S. **Performance and accuracy of Lattice-Boltzmann kernels on multi- and manycore architectures**. *Procedia Computer Science*, 1(1):239 – 247, 2010. ISSN 1877-0509.

RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B. e HWU, W.-M. W. **Optimization principles and application performance evaluation of a multithreaded GPU using CUDA**. Em *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, págs. 73–82, New York, NY, USA, 2008. ACM.

SCHEPKE, C. e MAILLARD, N. **Performance Improvement of the Parallel Lattice Boltzmann Method Through Blocked Data Distributions**. Em *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, págs. 71 –78, 24-27 2007.

SCHONHERR, M., KUCHER, K., GEIER, M., STIEBLER, M., FREUDIGER, S. e KRAFCZYK, M. **Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for CPUs and GPUs**. *Computers & Mathematics with Applications*, 61(12):3730 – 3743, 2011. ISSN 0898-1221.

SCHREIBER, M., NEUMANN, P., ZIMMER, S. e BUNGARTZ, H.-J. **Free-Surface Lattice-Boltzmann Simulation on Many-Core Architectures.** *Procedia CS*, págs. 984–993, 2011.

SCHROEDER, W., MARTIN, K. e LORENSEN, B. ***The Visualization Toolkit, Third Edition***. Kitware Inc., 2004.

SUCCI, S. **Lattice Boltzmann equation: Failure or success?** *Physica A: Statistical and Theoretical Physics*, 240(1-2):221 – 228, 1997. ISSN 0378-4371. Proceedings of the Euroconference on the microscopic approach to complexity in non-equilibrium molecular simulations.

SUCCI, S., SBRAGAGLIA, M. e UBERTINI, S. **Lattice Boltzmann Method**. Scholarpedia, Maio 2010. URL `http://www.scholarpedia.org /article /Lattice _Boltzmann _Method`.

TRENDALL, C. e STEWART, A. J. **General Calculations using Graphics Hardware with Applications to Interactive Caustics**. Em *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, págs. 287–298, London, UK, 2000. Springer-Verlag.

URQUIZA, S. A. e VENERE, M. J. **An Application Framework Architecture for FEM and Other Related Solvers**. *Mecânica Computacional*, XXI:3099 – 3709, 2002.

WANG, J., ZHANG, X., BENGOUGH, A. G. e CRAWFORD, J. W. **Domain-decomposition method for parallel lattice Boltzmann simulation of incompressible flow in porous media**. *Phys. Rev. E*, 72:016706, Jul 2005.

WELLEIN, G., ZEISER, T., HAGER, G. e DONATH, S. **On the single processor performance of simple lattice Boltzmann kernels**. *Computers & Fluids*, 35(8-9): 910 – 919, 2006. ISSN 0045-7930.

WOLF, F. G., SANTOS, L. O. D. e PHILIPPI, P. C. **Formação e dinâmica da interface líquido-vapor simulada pelo método Lattice-Boltzmann**. *Revista Brasileira de Ensino de FÃsica*, 28:167 – 175, 06 2006. ISSN 1806-1117.

WORLD HEALTH ORGANIZATION. **Cardiovascular Diseases (CVDs)**. World Health Organization, Janeiro 2011. URL `http:// www.who.int/ mediacentre/ factsheets/ fs317/en/`.

XIAN, W. e TAKAYUKI, A. **Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster**. *Parallel Computing*, 37(9):521 – 535, 2011. ISSN 0167-8191.

YU, D. e GIRIMAJI, S. S. **Multi-block Lattice Boltzmann method: Extension to 3D and validation in turbulence**. *Physica A: Statistical Mechanics and its Applications*, 362(1):118 – 124, 2006. ISSN 0378-4371.

ZHAO, Y., QIU, F., FAN, Z. e KAUFMAN, A. **Flow simulation with locally-refined LBM**. Em *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, págs. 181–188, New York, NY, USA, 2007. ACM.

ZHOU, J., ZHONG, C., XIE, J. e YIN, S. **Multiple-GPUs algorithm for lattice boltzmann method**. volume 2, págs. 793–796, 2008.

ZIEMER, P., COLLARES, M., PIVELLO, M., BLANCO, P. e FEIJOO, R. **HeMoLab 1D: Modelagem Simplificada do Sistema Cardiovascular Humano**. *XI Congresso Brasileiro de Informática em Saúde*, pág. 5, 2008.