UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE GRADUAÇÃO EM ENGENHARIA AEROESPACIAL

DENIS LEITE GOMES

PARALELIZAÇÃO DO MÉTODO LATTICE BOLTZMANN 2D EM CUDA

Joinville
2016

DENIS LEITE GOMES

PARALELIZAÇÃO DO MÉTODO LATTICE BOLTZMANN 2D EM CUDA

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do Grau de Bacharel em Engenharia Aeroespacial.

Orientador: Prof. Juan Pablo de Lima Costa Salazar, Ph.D.

Joinville
2016

Denis Leite Gomes

PARALELIZAÇÃO DO MÉTODO LATTICE BOLTZMANN 2D EM CUDA

Este trabalho de conclusão de curso foi julgado adequado para obtenção do Título de Engenheiro Aeroespacial, aprovado em sua forma final pelo curso de Curso de Graduação em Engenharia Aeroespacial da Universidade Federal de Santa Catarina.

Joinville, 01 de dezembro de 2016.

Prof. Dr. Talita Sauter Possamai
Coordenador do Curso

Banca Examidora:

Prof. Juan Pablo de Lima Costa Salazar, Ph.D.
Presidente da Banca

Prof. Dr. Luis Orlando Emerich dos Santos
Membro da Banca

Prof. Dr. Diogo Nardelli Siebert
Membro da Banca

Dedico este trabalho aos meus avós, na Terra e no Céu,
aos meus padrinhos, de Batismo e de Crisma,
à minha irmã e aos meus pais.

AGRADECIMENTOS

A Ti, meu Jesus e meu Deus, que tanto nos amais, pela Tua dolorosa Paixão e pela Tua gloriosa Ressurreição, pelo dom da minha vida e de toda a Criação. Pela torrente de graças que me concedestes ao abrir meu coração para Te procurar, Te encontrar e Te amar, e de dizer o *fiat*, como Maria, à minha vocação. "Senhor, Tu sabes tudo; Tu sabes que eu Te amo!" (Jo 21, 17).

Aos pais, Tatiana e Dair, pelo amor incondicional e pelo exemplo de vida. Por terem sempre me amparado ao longo das minhas crises existenciais desde que nos mudamos para Porto Seguro até a minha chegada, perdido, em Joinville. Vocês nunca mediram esforços para me acompanhar, cuidar de mim e garantir o meu pleno desenvolvimento como criatura humana. Qualquer tentativa de fazer um agradecimento justo seria pura vaidade e muito pouco perto da a gratidão que sinto e também pela que é devida aos muitos sacrifícios que vocês fizeram.

À minha *mana*, Gabriela, não só pelo exemplo de força e dedicação, mas principalmente pelos incontáveis risos e abraços, e pela tua presença sempre perto de mim. Bibinha, faltam-me palavras para te agradecer e manifestar tudo o que tu representas para mim.

À minha vó, Jenny, com quem aprendi a confiar mais em Deus e a sair de mim mesmo para *servir*, e ao vô Toninho, pela companhia repleta de brincadeiras que sempre me fazem extravasar em lágrimas de alegria. À minha tia e dinda Mariani, pelo carinho e pelo apoio, e ao meu primo e padrinho, Lalo, pela convivência espirituosa e bem-humorada em Porto Alegre.

Ao meu amigo e Professor, Alexandre, pelo exemplo de simplicidade e caridade, e pelos sábios conselhos depositados no meu abismo de dúvidas. Por me instigar a procurar, encontrar e amar a Cristo, sem o qual eu nada sou. Haverá como agradecer o bastante? "Quem de vós que, tendo cem ovelhas e perdendo uma delas, não deixa as noventa e nove no deserto e vai em busca da que se perdeu, até encontrá-la? E depois de encontrá-la, a põe nos ombros, cheio de júbilo, e, voltando para casa, reúne os amigos e vizinhos, dizendo-lhes: Regozijai-vos comigo, achei a minha ovelha que se havia perdido" (Lc 15, 4-6). *Deo omnis Gloria!*

Às minhas famílias alemãs, às quais devo meu amadurecimento e formação humana.

Aos Profs. Marcos, Fabiano, Rafael Cuenca, Diogo, Emerich, Kleber, De Carli, Dourado, Modesto, Catapan e Cirilo, pelas muitas ajudas e por todo o conhecimento transmitido.

Ao meu orientador, Prof. Juan Pablo, por ter me escolhido lá em 2013 para fazer essa pesquisa, pela sua compreensão das minhas limitações e por ter acreditado em mim.

Aos Profs. Talita, Renato e Vicente, pelo apoio acadêmico e pelo estágio no LabCET em Florianópolis, ao lado dos meus amigos brasileiros, Edemar, Daniel e Flávia, e meus *hermanos* colombianos, Raul, Juan, Jair e Renzo, aos quais agradeço de coração pelo acolhimento, força, aprendizado e diversão proporcionados neste tempo curto, mas verdadeiramente *radiante*.

Aos meus amigos André, Vinícius, José, Freire, Rudimar, Gabriel, Wendel, e tantos outros. Aos colegas do LABCC, pelos momentos de descontração, incentivo e confraternização, e especialmente à Damylle e à Marielle, pelo apoio moral nas horas difíceis. Aos muitos que aqui não foram citados. Isso não diminui em nada o que vocês fizeram de mim e por mim.

Aos meus amigos da UFRGS. Ao Paulo, pelo som metalizado, ao Rafão, *poko loko*, e ao Renan, pela parceria no RU e nas andanças dos semestres inesquecíveis que passei aí.

Aos meus irmãos do coração, Caio *Tirado* Abreu, Rafael e Rodrigo Morawski, e Henrique Teixeira. E a todos meus amigos de infância. A nossa amizade vai daqui para a eternidade.

Aos incontáveis brasileiros anônimos que financiaram meus estudos através da rede pública. À Petrobras e ao CNPq (PIBIC), pelo suporte financeiro nesses mais de três anos.

À UFSC, esse lar em que aprendi a unir gelo e fogo, razão e coração, equação e oração, material e espiritual; onde aprendi a ser mais humano: homem e filho de Deus.

"Não tenhas espírito de caipira. - Dilata o teu coração, até que seja universal, católico.

Não voes como ave de capoeira, quando podes subir como as águias."

SÃO JOSEMARIA ESCRIVÁ

"Tudo o que temos de decidir é o que fazer com o tempo que nos é dado."

J. R. R. TOLKIEN

"De tudo, ficaram três coisas:

a certeza de que estamos sempre começando,
a certeza de que precisamos continuar,
a certeza de que seremos interrompidos antes de terminar.

Portanto, devemos:

fazer da interrupção, um caminho novo;
da queda, um passo de dança;
do medo, uma escada;
do sonho, uma ponte;
da procura, um encontro."

FERNANDO SABINO

RESUMO

A dinâmica dos fluidos computacional tem exigido uma capacidade de processamento cada vez maior para a simulação de cenários reais que envolvem escoamento de fluidos na engenharia aeroespacial. Processadores gráficos tem ganhado espaço na aceleração de métodos numéricos empregados em diversas aplicações, pois seu uso se torna tanto eficiente quanto maior for a intensidade aritmética e o grau de paralelismo do algoritmo. O Método Lattice-Boltzmann é conhecido por agregar tais características no contexto de métodos numéricos de CFD e seu processamento em plataformas paralelas como as GPUs tem se mostrado promissor. O presente trabalho realiza uma implementação paralela em linguagem C do modelo LBGK D2Q9 monofásico utilizando a plataforma CUDA™ para o processamento em placas gráficas da NVIDIA®. Uma função para a imposição da condição de contorno periódica foi desenvolvida sobre um modelo de paralelização difundido na literatura para simular um escoamento entre placas planas paralelas infinitas. Este último, por sua vez, é usado para verificar o código, demonstrando boa concordância com a solução analítica. O algoritmo foi então usado para simular um *benchmark* do escoamento da cavidade quadrada com tampa móvel e apresentou resultados qualitativos satisfatórios.

Keywords: paralelização do Método Lattice Boltzmann, LBGK D2Q9 em CUDA, GPGPU, CFD, cavidade quadrada com tampa móvel.

ABSTRACT

Computational fluid dynamics has been requesting an increasingly computing capacity for simulating real world scenarios involving fluid flows in aerospace engineering. Graphics processors have gained ground on the acceleration of numerical methods that are employed in several applications, for its use becomes more efficient the higher the arithmetic intensity and the degree of parallelism of the algorithm. The Lattice-Boltzmann Method is known for embracing such characteristics in the context of numerical methods in CFD and its processing in parallel platforms such as GPUs has been shown to be promising. The current work performs a parallel implementation in C language of the LBGK D2Q9 model utilizing CUDA™ for the processing in NVIDIA® graphics cards. A function for imposing periodic boundary conditions has been developed over a widespread parallelization model within the literature in order to simulate a flow between infinite parallel plates. The latter, for once, is used for verifying the code, demonstrating it is in good agreement with the analytic solution. The verified program was used to simulate a *benchmark* of lid-driven cavity flow and presented satisfactory qualitative results.

Keywords: Lattice Boltzmann Method parallelization, LBGK D2Q9 with CUDA, GPGPU, CFD, Lid-driven cavity flow.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

2D	Bidimensional
AoS	Vetor de estruturas (do inglês, *Array of Structures*)
API	Interface de programação de aplicação (do inglês, *Application Programming Interface*)
BC	Condição de contorno (do inglês, *Boundary Condition*)
CFD	Dinâmica dos fluidos computacional (do inglês, *Computational Fluid Dynamics*)
CPU	Unidade de processamento central (do inglês, *Central Processing Unit*)
CUDA	Arquitetura de dispositivo unificado de computação (do inglês, *Compute Unified Device Architecture*)
IDE	Ambiente de desenvolvimento integrado (do inglês *Integrated Development Environment*)
D2Q9	Modelo de rede LBM de duas direções e nove velocidades
DRAM	Memória dinâmica de acesso aleatório (do inglês, Dynamic random-access memory)
FLOPs	Operações de ponto flutuante por segundo (do inglês, *Floating-point Operations Per Second*)
GPGPU	Computação de propósito geral em unidades de processamento gráfico (do inglês, *General-purpose computing on Graphics Processing Units*)
GPU	Unidade de processamento gráfico (do inglês, *Graphics Processing Unit*)
LBGK	Lattice Bhatnagar-Gross-Krook
LBM	Método Lattice Boltzmann (do inglês, *Lattice Boltzmann Method*)
lu	Unidades de rede (do inglês, *lattice units*)
NVCC	Compilador CUDA da NVIDIA (do inglês, *NVIDIA CUDA Compiler*)
ts	Passo de tempo (do inglês, *timestep*)
RMSD	Desvio quadrático médio (do inglês, *Root Mean Square Deviation*)
RMSE	Erro quadrático médio (do inglês, *Root Mean Square Error*)
SIMT	Instrução única e múltiplas threads (do inglês, *single-instruction multiple-thread*)
SIMD	Instrução única e dados múltiplos (do inglês *single instruction, multiple data*)
SISD	Instrução única e dado único (do inglês *single instruction, single data*)
SM	Multiprocessador de fluxo (do inglês, *Streaming Multirocessor*)
SoA	Estrutura de vetores (do inglês, *Structure of Arrays*)
SP	Processador de fluxo (do inglês, *Streaming Processor*)

LISTA DE SÍMBOLOS

Símbolo	Descrição	Unidade
ρ	Densidade	$\mathrm{kg\,m^{-3}}$
∂	Operador del, para derivadas parciais	-
$\mathbf{u}$	Vetor velocidade macroscópica	$\mathrm{m\,s^{-1}}$
t	Tempo	s
∇	Operador (diferencial vetorial) nabla	-
g	Aceleração de corpo por unidade de volume	$\mathrm{lu\,ts^{-2}\,m^{-3}}$
p	Pressão	Pa
μ	Viscosidade dinâmica	$\mathrm{Pa\,s}$
∇^2	Operador Laplace ou Laplaciano	-
Re	Número de Reynolds	-
U	Velocidade característica	$\mathrm{m\,s^{-1}}$
L	Comprimento característico	m
ν	Viscosidade cinemática	$\mathrm{m^2\,s^{-1}}$
f	Função distribuição de partícula	$\mathrm{ts\,lu^{-6}}$
$\mathbf{c}$	Velocidade microscópica das partículas	$\mathrm{lu\,ts^{-1}}$
$\Omega(f)$	Operador de colisão	-
f_i	Função distribuição de partícula na direção i	$\mathrm{ts\,lu^{-6}}$
$\mathbf{x}$	Posição da partícula na rede	lu
c	Velocidade básica da rede	$\mathrm{lu\,ts^{-1}}$
$\mathbf{e_i}$	Velocidade microscópica da partícula na direção i	$\mathrm{lu\,ts^{-1}}$
Δt	Incremento temporal	ts
f_i^{eq}	Função distribuição de equilíbrio na direção i	$\mathrm{ts\,lu^{-6}}$
τ	Tempo de relaxação para o equilíbrio	-
$\Omega(f)$	Operador de colisão na direção i	-
Δx	Distância entre nós da malha na direção cartesiana X	lu
ω_i	Peso associado a cada velocidade microscópica no D2Q9	-
c_s	Velocidade do som no modelo de rede	$\mathrm{lu\,ts^{-1}}$
$\mathbf{u}^{eq}$	Velocidade macroscópica atualizada pela força de corpo	$\mathrm{m\,s^{-1}}$
$\mathbf{F}$	Força de corpo por unidade de volume	$\mathrm{N\,m^{-3}}$
f_n	Função distribuição de partícula na direção normal à fronteira	$\mathrm{ts\,lu^{-6}}$
u	Componente X da velocidade macroscópica	$\mathrm{m\,s^{-1}}$
v	Componente Y da velocidade macroscópica	$\mathrm{m\,s^{-1}}$
$u_{n,i}$	Aproximação numérica para a componente X da velocidade em $\mathbf{x_i}$	$\mathrm{lu\,ts^{-1}}$
$u_{a,i}$	Componente X da velocidade em $\mathbf{x_i}$ na solução analítica	$\mathrm{lu\,ts^{-1}}$
N	Número de total de pontos no domínio do fluido	-
$\overline{RMSD}$	RMSD normalizado	-
$\bar{u}$	Velocidade média	$\mathrm{m\,s^{-1}}$
NY	Número de pontos da rede na direção Y	-
N_x	Número de *threads* por bloco	-
$\tilde{t}$	Constante de tempo adimensional	-
T_0	Tempo característico da simulação	ts
N_{it}	Número de passos de tempo da simulação	ts
G	Gradiente de pressão (gravitacional)	$\mathrm{Pa\,m}$
t_P	Tempo de processamento	s
$\overline{RMSE}$	RMSE normalizado	-

SUMÁRIO

1 INTRODUÇÃO

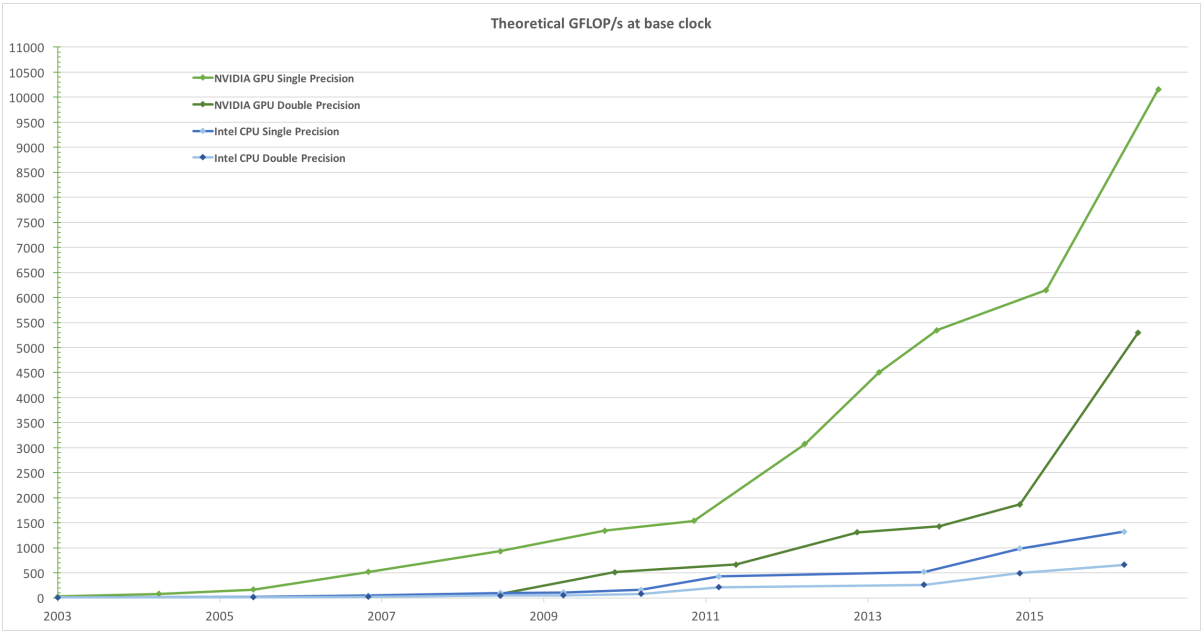
Desde seu surgimento na década de 1940, computadores eletrônicos programáveis tem o processo de desenvolvimento de produtos na indústria, possibilitando otimizar custos e tempo de trabalho através de simulações numéricas. Especificamente, na engenharia aeroespacial, o estudo de escoamento de fluidos costuma ser crítico em projetos, de modo que está amplamente difundido hoje o uso de modelos de dinâmica dos fluidos computacional, Computational Fluid Dynamics (CFD). Todavia, na busca por soluções para problemas cada vez mais complexas, as aplicações de CFD trazem normalmente consigo a demanda por hardwares com alta capacidade de processamento para que se tornem viáveis em uma escala de tempo tolerável. A performance computacional necessária para execução desses simuladores demanda frequentemente o uso de supercomputadores e clusters de computadores, os quais tem um alto custo de operação em relação ao número de operações de ponto flutuante por segundo (ou FLOPs, do inglês *floating-point operations per second*). Nesta busca por dispositivos de computação que combinem eficiência e baixo custo, as unidades de processamento gráfico (GPUs, do inglês *Graphics Processing Units*) tem se destacado como uma alternativa capaz de sintetizar ambas características. Recentemente, um supercomputador equipado com GPUs da NVIDIA® assumiu o posto de computador mais rápido do mundo, demonstrando que o uso dessa tecnologia veio consolidar a computação heterogênea, a qual une o controle e execução serial de tarefas na unidade de processamento central, Central Processing Unit (CPU), à capacidade de processamento paralelo das GPUs, sendo uma alternativa eficaz no cenário da computação de alta performance. Dessa maneira, a escolha e aplicação de um método numérico capaz de explorar esse potencial, mostra-se como uma solução efetiva para minimização do tempo computacional e gastos energéticos. Em resposta a essa demanda, uma técnica de CFD chamada Método Lattice-Boltzmann (LBM, do inglês *Lattice-Boltzmann Method*) tem sido amplamente difundida no meio científico, pois seu algoritmo, além de simples, é caracterizado por apresentar um bom grau de paralelismo e alta intensidade computacional (razão entre o número de instruções de computação e o número de operações de leitura/escrita), o que justifica os esforços envolvidos em sua paralelização (CHEN; DOOLEN, 1998).

Por trinta anos, um dos artifícios mais significativos para o aprimoramento da performance de dispositivos de computação fora aumentar a frequência de operação do processador. Entretanto, há alguns anos as CPUs tem se aproximado do limite da sua frequência de operação devido à intensidade de calor dissipado por unidade de volume no denso aglomerado de transistors que compõe sua estrutura interna. Isso implica em custos com refrigeração que crescem vertiginosamente à medida em que se procura elevar o chamado *clock* dos CPUs. Essa barreira

foi contornada por meio da utilização de um modelo de computação baseada em vários núcleos de processadores, chamados multiprocessadores, os quais constituem praticamente a totalidade dos computadores comercializados no mundo atualmente. As CPUs são projetadas particularmente para executar um pequeno número de tarefes complexas e são adequadas em sistemas que executam tarefas discretas e desconectadas. Lidar com a execução simultânea de múltiplas tarefas implica em esforços para os programadores, especialmente quando elas tem de interagir trocando informações ou disputando espaços na memória para leitura ou escrita de dados. Além disso, o custo relacionado a maiores requisitos de fluxo de processamento e memória aumenta exponencialmente na medida em que se distancia de uma máquina de prateleira (COOK, 2013).

Em contrapartida, as placas de vídeo dedicadas equipam a maioria dos computadores atuais e estão disponíveis para as massas a preços relativamente baixos e vem sendo utilizados desde a década de 2000 para acelerar aplicações científicas e de engenharia. Essencialmente o GPU é um processador dedicado à renderização de imagens que tem a capacidade de realizar centenas de operações concorrentemente sobre grandes blocos de dados visuais (AKSNES; HESLAND, 2009). Essa funcionalidade passou a ser extendida para cálculos gerais desde 2006, quando a maior empresa fornecedora de processadores gráficos lançou no mercado os GPUs habilitados com CUDA™ (do inglês, *Compute Unified Device Architecture*), uma plataforma de computação paralela e modelo de programação que permitiu aproveitar o poder de processamento e alta largura de banda de memória desses dispositivos através de uma interface de programação simplificada (NVIDIA CORPORATION, 2016). A partir daí muitos problemas computacionalmente complexos tem sido resolvidos de maneira mais eficiente que em CPUs, aproveitando a discrepância no número de operações de ponto flutuante por segundo (FLOPs) entre ambos, como se pode constatar pela Figura 1. O mesmo comportamento é observado para a largura de banda de memória (taxa de transferência de dados para a memória por segundo). A aplicação de placas gráficas para essa finalidade é denominada computação de propósito geral em unidades de processamento gráfico (GPGPU, do inglês *General-purpose computing on Graphics Processing Units*) e surgiu como uma abordagem competitiva para acelerar problemas computacionalmente custosos em aplicações de engenharia (XIONG et al., 2012).

Figura 1 – Capacidade de processamento teórico de CPUs e GPUs, em bilhões de FLOPs.



Fonte: (NVIDIA CORPORATION, 2016).

Dessa maneira, o potencial das GPGPUs pode ser bem explorado pelo uso de métodos computacionais altamente paralelizáveis, como é o caso do LBM, já que este possui características que o tornam efetivo na computação paralela, exceto por uma restrição de sincronização local (OBRECHT et al., 2013). Apesar de necessitar de alta capacidade computacional e demandar muita memória, destaca-se por se ajustar muito bem à arquitetura de hardware das GPGPUs, sendo um método propício a ser implementado eficientemente em CUDA (KUZNIK et al., 2010). Também destaca-se pela sua capacidade de representar escoamentos em ambientes com fronteiras complexas e até mesmo em fenômenos onde há acoplamento de processos físicos e químicos complexos (ZHOU et al., 2012). Além disso as computações baseadas em LBM são bem ajustadas para simular fenômenos em escalas microscópicas como meios porosos assim como problemas que incluem escoamentos fluidos multifásicos, imiscíveis, turbulentos e efeitos de superfície (BAILEY et al., 2009).

Tölke (2009) mostrou que é possível obter um ganho de performance superior a uma ordem de grandeza em um modelo LBM bidimensional (2D) empregando uma implementação eficiente em CUDA. Fundamentado neste resultado, o presente trabalho se propôs a implementar o modelo de Tölke (2009) e imprimiu esforços no desenvolvimento de uma rotina para imposição da condição de contorno periódica na direção do escoamento. Com ela, um escoamento entre placas planas paralelas foi simulado para várias condições hidrodinâmicas para efeito de verificação do código com a solução analítica. Posteriormente o problema da cavidade quadrada com tampa móvel é resolvido e comparado com a solução numérica em malha refinada, resultado de uma simulação usando o método de diferenças finitas.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Paralelizar o modelo LBGK D2Q9 do Método Lattice-Boltzmann usando processadores gráficos com a plataforma CUDA.

1.1.2 Objetivos Específicos

Para atingir esse objetivo geral, necessita-se executar cada uma das etapas abaixo:

- Implementar o modelo de paralelização do LBGK em CUDA proposto por Tölke (2009);
- Verificar a ferramenta de simulação com a solução analítica do escoamento entre placas planas paralelas infinitas;
- Simular, com o programa desenvolvido e verificado, e um escoamento em cavidade quadrada com tampa móvel.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está divido em seis capítulos, descritos abaixo conforme suas especificidades:

- **Revisão bibliográfica**:
  - **Método Lattice-Boltzmann**: este capítulo inicia com uma breve revisão de mecânica dos fluidos, tendo em vista que o programa desenvolvido tem o propósito de simular escoamentos isotérmicos e monofásicos em determinadas configurações fluidodinâmicas. Na sequência, os fundamentos do LBM são apresentados, tais como: sua equação governante, as características do modelo microscópico, o cálculo de propriedades macroscópicas, imposição de forças de corpo e condições de contorno. Também é mostrado o modelo 2D utilizado neste trabalho e alguns aspectos relacionados à estabilidade do mesmo são mencionados.
  - **Programação de GPUs em CUDA**: a plataforma de paralelização da NVIDIA® é apresentada como um todo. São abordados aspectos da arquitetura do hardware, do modelo de programação e da sua API.
  - **Implementação do LBM em CUDA**: aqui é exposto o modelo de paralelização de Tölke (2009) e o seu algoritmo, incluindo a estrutura de armazenamento de dados e a sua programação. Também é demonstrada a função desenvolvida para a implementação da condição de contorno periódica.
- **Metodologia**: esta seção descreve os métodos e as ferramentas utilizados para implementar o código tanto quanto os programas e plataformas utilizados para elaborar as figuras, exibir e analisar os resultados.
- **Resultados e discussões**: neste capítulo são exibidos e analisados os resultados obtidos nas simulações efetuadas com o programa desenvolvido, tanto qualitativa e quanto quantitativamente. Também são feitas a verificação do código com a solução exata dos escoamento entre placas planas paralelas infinitas e a comparação numérica da simulação de um caso de escoamento em cavidade quadrada com tampa móvel.
- **Conclusão**: por fim, é encerrado o texto com algumas observações importantes acerca das contribuições deste trabalho, suas capacidades e suas limitações, bem como sugestões para trabalhos futuros.

2 MÉTODO LATTICE BOLTZMANN

2.1 REVISÃO DE MECÂNICA DOS FLUIDOS

A dinâmica do movimento de fluidos viscosos no domínio do espaço contínuo é descrita pelas equações de Navier-Stokes. A derivação matemática dessas equações resulta da aplicação da segunda lei de Newton a um elemento infinitesimal de fluido, juntamente com a hipótese de proporcionalidade entre tensão e deformação. Para um escoamento incompressível e isotérmico, as equações de Navier-Stokes são expressas como:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)=\rho\mathbf{g}-\nabla p+\mu\left(\nabla^2\mathbf{u}\right), \tag{1}$$

onde ρ é a densidade do fluido, $\mathbf{u}$ é o vetor velocidade do escoamento, ∇ é o operador diferencial *del*, g representa acelerações de corpo (tais como gravidade, campos elétricos, acelerações inerciais, etc.), p é a pressão, μ é a viscosidade dinâmica e ∇^2 é o operador laplaciano.

Para a descrição completa da fluidodinâmica, o sistema deve obedecer o princípio de conservação da massa, o qual é representado pela Equação da Continuidade, mostrada abaixo:

$$\frac{\partial \rho}{\partial t}+\nabla\cdot(\rho\mathbf{u})=0. \tag{2}$$

Em caso de escoamentos onde a transferência de calor é importante, é necessário ainda acoplar a equação da energia e, no caso de gases, também a equação de estado.

Soluções analíticas desse sistema de equações parciais foram obtidas para um pequeno conjunto de escoamentos, com propriedades físicas e condições de contorno bem comportadas. Portanto, a análise de sistemas mais complexos só se torna possível pelo emprego de métodos numéricos de CFD (PRITCHARD, 2010).

Para predizer e caracterizar diferentes padrões de escoamento, utiliza-se o parâmetro adimensional conhecido como número de Reynolds, o qual pode ser entendido como a razão entre as forças inerciais e as forças viscosas:

$$Re=\frac{UL}{\nu}, \tag{3}$$

onde U é a velocidade na corrente livre, L é o comprimento característico e ν é a viscosidade cinemática, expressa pela razão entre a viscosidade dinâmica μ e densidade ρ. A viscosidade é uma medida da resistência do fluido à deformação, consequência das forças intermoleculares e

colisões que provocam atrito entre as camadas de fluido.

2.2 EQUAÇÃO DE LATTICE BOLTZMANN

O Método de Lattice Boltzmann tem sua origem na teoria cinética dos gases desenvolvida pelo físico austríaco Ludwig Eduard Boltzmann (1844-1906), a qual explica e prevê, através da mecânica estatística, como as propriedades macroscópicas dos fluidos derivam das suas propriedades microscópicas. Gases e fluidos são descritos como constituídos por um número muito grande de partículas cujo movimento e colisões são tratados de maneira probabilística. Através do conceito de função distribuição de partículas, as moléculas constituintes não são tratadas individualmente, mas agrupadamente, sendo descritas pela mecânica clássica segundo uma probabilidade de se deslocar dentro de uma dada faixa de velocidades e posições em um dado instante de tempo (MOHAMAD, 2011). A equação governante que descreve o comportamento de um sistema termodinâmico de gases fora do equilíbrio é a chamada Equação do Transporte de Boltzmann (ETB), dada por:

$$\frac{\partial f}{\partial t} + \mathbf{c} \cdot \nabla f = \Omega(f), \tag{4}$$

onde f é a função de distribuição, a qual representa a densidade probabilística de encontrar um conjunto de partículas localizadas no volume $d\mathbf{x}$ centrado na posição $\mathbf{x}$ e com velocidades entre $\mathbf{c}$ e $\mathbf{c} + d\mathbf{c}$ no intervalo de tempo Δt; $\mathbf{c}$ é a velocidade microscópica das partículas, e $\Omega(f)$ é o chamado operador de colisão. O lado esquerdo da Equação 4 representa a advecção de f e o lado direito é um termo fonte. Pelo fato dessa ser uma equação integro-diferencial cujo termo fonte é dependente da variável que se deseja resolver, sua solução é não-trivial.

A ETB é simplificada no LBM através de um processo de discretização tanto do espaço, confinando as partículas em nós de uma rede, quanto do número de velocidades microscópicas, restringindo as direções de propagação possíveis. Qian, d'Humières e Lallemand (1992) desenvolveram uma aproximação para este termo baseada em um único tempo de relaxação, pelo qual a relaxação para a função de distribuição de equilíbrio apropriada ocorre a uma taxa constante. Essa abordagem deu origem ao método Lattice BGK (LBGK), o qual resolve a chamada Equação de Lattice Boltzmann:

$$f_i(\mathbf{x} + c\mathbf{e_i}\Delta t, t + \Delta t) - f_i(\mathbf{x}, t) = -\frac{f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)}{\tau}, \tag{5}$$

na qual i representa a i-ésima componente de um conjunto de velocidades possíveis, $\mathbf{x}$ identifica a posição discreta da partícula na malha, $c\mathbf{e_i}\Delta t = \Delta x$ representa uma distância internodal, Δt é um intervalo de tempo, medido em passos de tempo de simulação ts (do inglês, *timestep*), $c = \Delta x/\Delta t$ é a velocidade básica da rede, medida em $\mathrm{lu/ts}$, e $\mathbf{e_i}$ é a velocidade microscópica da rede na i-ésima direção no modelo de rede adotado.

2.3 FUNDAMENTOS DO LBM

De certo modo, o LBM pode ser visto como um método de diferenças finitas específico para a solução da ETB em uma malha. É fácil perceber isso escrevendo a ETB em termos da função de distribuição discreta,

$$\frac{\partial f_i}{\partial t} + c\mathbf{e_i} \cdot \nabla f_i = \Omega_i, \tag{6}$$

e, discretizando o operador diferencial e o operador de colisão da seguinte maneira:

$$\frac{f_i(\mathbf{x}, t+\Delta t) - f_i(\mathbf{x}, t)}{\Delta t} + \frac{f_i(\mathbf{x} + c\mathbf{e_i}\Delta t, t+\Delta t) - f_i(\mathbf{x}, t+\Delta t)}{\Delta t} = -\frac{f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)}{\tau}, \tag{7}$$

com a imposição de que $\Delta t = 1$ e $\Delta x = 1$, recupera-se a equação de evolução do método (BAO; MESKAS, 2011). No modelo de colisão BGK, o operador de colisão é dado por:

$$\Omega_i(f_i(\mathbf{x}, t)) = -\frac{f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)}{\tau}, \tag{8}$$

em que f_i^{eq} é a chamada função de distribuição de equilíbrio de Maxwell-Boltzmann discretizada, τ constitui um parâmetro de relaxação para o equilíbrio local da aproximação BGK e e Ω_i é o operador de colisão que atua sobre f_i (BHATNAGAR; GROSS; KROOK, 1954).

Normalmente, as equações diferenciais parciais em mecânica dos fluidos são resolvidas utilizando o método de diferenças finitas, de volumes finitos ou de elementos fluidos. Em LBM, no entanto, a solução é obtida em duas etapas denominadas colisão e propagação, com um domínio subdividido em sítios fixos. Tais sítios concentram conjuntos de partículas, as quais são propagadas para os nós adjacentes após um processo colisional.

A colisão é definida matematicamente pela soma do operador de colisão às funções de distribuição locais, conforme a Equação 9:

$$f_i(\mathbf{x}, t+\Delta t) = f_i(\mathbf{x}, t) + \Omega_i(f_i(\mathbf{x}, t)). \tag{9}$$

A propagação transporta a informação, de acordo com a direção, pela Equação 10:

$$f_i(\mathbf{x} + c\mathbf{e_i}\Delta t, t+\Delta t) = f_i(\mathbf{x}, t+\Delta t). \tag{10}$$

O número de direções de propagação é determinado pelo arranjo da rede, o qual é estabelecido pelo modelo de Lattice Boltzmann adotado. Neste trabalho será adotado o LBGK D2Q9, um dos modelos mais usados para resolver problemas de escoamento isotérmico monofásico em duas dimensões (MOHAMAD, 2011).

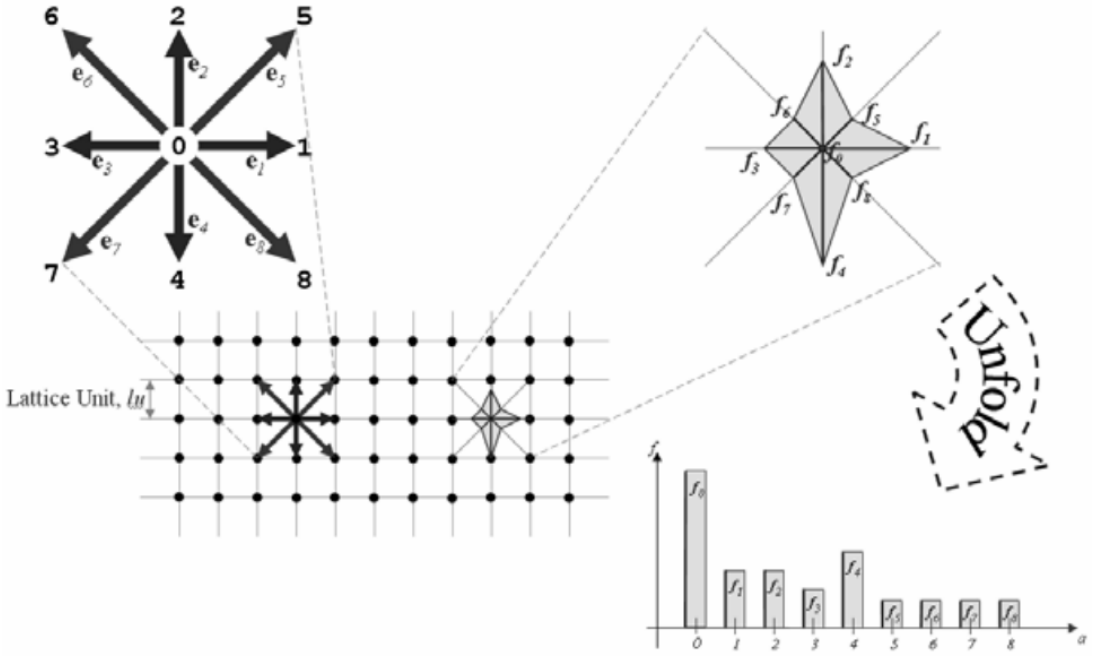
2.4 MODELO LBGK D2Q9

O LBM simplifica a solução da equação de Boltzmann reduzindo o domínio contínuo de posições espaciais e velocidades microscópicas a um pequeno número e o próprio domínio do tempo a passos de tempo discretos. No modelo de duas dimensões e nove velocidades (D2Q9) as posições das partículas estão restritas aos nós de de uma rede bidimensional de pontos igualmente espaçados por uma unidade de rede (lu, do inglês *lattice unit*). As velocidades assumem oito direções distintas e uma velocidade para aquelas em repouso, como mostra a Figura 2 (SUKOP; THORNE, 2007). Duas são as magnitudes de velocidade possíveis nesse modelo: 1, para as direções cardinais (leste, norte, oeste e sul) e $\sqrt{2}$, para as direções intercardinais (nordeste, noroeste, sudoeste e sudeste).

A densidade macroscópica do fluido, ρ, pode ser obtida por integração das funções de distribuição de partículas:

$$\rho(\mathbf{x}, t) = \sum_{i=0}^{8} f_i(\mathbf{x}, t). \tag{11}$$

Figura 2 – Modelo LBM D2Q9 explicitando as direções de velocidade e a função de distribuição associada a cada uma delas.



Fonte: Sukop e Thorne (2007, p. 34).

A velocidade macroscópica $\mathbf{u}$ é definida como uma média das velocidades microscópicas da rede, $c\mathbf{e_i}$, ponderados pelas respectivas funções de distribuição, f_i:

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{\rho} \sum_{i=0}^{8} c f_i \mathbf{e_i}, \tag{12}$$

onde $\mathbf{e_i}$, no modelo D2Q9, é definido como:

$$\mathbf{e_i} = \begin{cases} \langle 0, 0 \rangle, & i = 0; \\ \langle +1, 0 \rangle, \langle 0, +1 \rangle, \langle -1, 0 \rangle, \langle 0, -1 \rangle & i = 1, 2, 3, 4; \\ \langle +1, +1 \rangle, \langle -1, +1 \rangle, \langle -1, +1 \rangle, \langle -1, -1 \rangle, & i = 5, 6, 7, 8. \end{cases} \tag{13}$$

A função distribuição de equilíbrio discretizada é dada pela relação:

$$f_i^{eq}(\mathbf{x}, t) = \rho(\mathbf{x}, t) \omega_i \left[1 + \frac{3}{c^2} (\mathbf{e_i} \cdot \mathbf{u}) + \frac{9}{2c^4} (\mathbf{e_i} \cdot \mathbf{u})^2 - \frac{3}{2c^2} \mathbf{u}^2 \right], \tag{14}$$

onde ω_i representa o peso associado a cada velocidade microscópica no modelo D2Q9:

$$w_i = \begin{cases} 4/9, & i = 0; \\ 1/9, & i = 1, 2, 3, 4; \\ 1/36, & i = 5, 6, 7, 8; \end{cases} \tag{15}$$

onde $i = 0$ refere-se às partículas em repouso, $i = 1, 2, 3, 4$ e $5, 6, 7, 8$ às partículas propagadas nas direções cardeais e intercardinais respectivamente (SUKOP; THORNE, 2007).

O tempo de relaxação para o equilíbrio, τ, é um parâmetro adimensional que está relacionado diretamente com a viscosidade cinemática do fluido, a qual assume a fórmula:

$$\nu = \frac{2\tau - 1}{6}\frac{(\Delta x)^2}{\Delta t}, \tag{16}$$

de modo que τ não deve ser muito próximo de $1/2$ para evitar instabilidades numéricas. Seguindo uma recomendação de Sukop e Thorne (2007).

Para escoamentos envolvendo gases ideais, pode-se definir a pressão em função da densidade macroscópica como:

$$p = c_s^2 \rho, \tag{17}$$

onde c_s é a velocidade do som na rede, que vale $1/3$ lu/ts no D2Q9.

Através de uma análise multiescalar de Chapman-Enskog, pode-se recuperar as equações de Navier-Stokes no limite incompressível para baixo número de Mach, o que justifica a aplicação do método para essa classe de escoamentos (MOHAMAD, 2011).

O método possibilita a adição de forças de corpo externas sobre as partículas na forma de uma variação de velocidade. A velocidade é atualizada conforme a Equação 18, sendo a força representada pela variável $\mathbf{F}$, e é posteriormente usada no cálculo da função distribuição de equilíbrio (SUKOP; THORNE, 2007):

$$\mathbf{u}^{eq} = \mathbf{u} + \Delta\mathbf{u} = \mathbf{u} + \frac{\tau\mathbf{F}}{\rho}, \tag{18}$$

$\mathbf{u}^{eq}$ é a velocidade atualizada pela força de corpo externa, $\mathbf{F}$, que é medida em termos de força por unidade de volume.

2.5 Condições de contorno

Para delimitar a solução do problema, é preciso aplicar corretamente as condições de contorno (BCs, do inglês *Boundary Conditions*) que estejam em concordância com o fenômeno estudado. Além disso, a estabilidade e a precisão da computação são afetadas significativamente pelas condições de contorno no LBM (QIN et al., 2012).

2.5.1 Tipos de condição de contorno

- Períodica

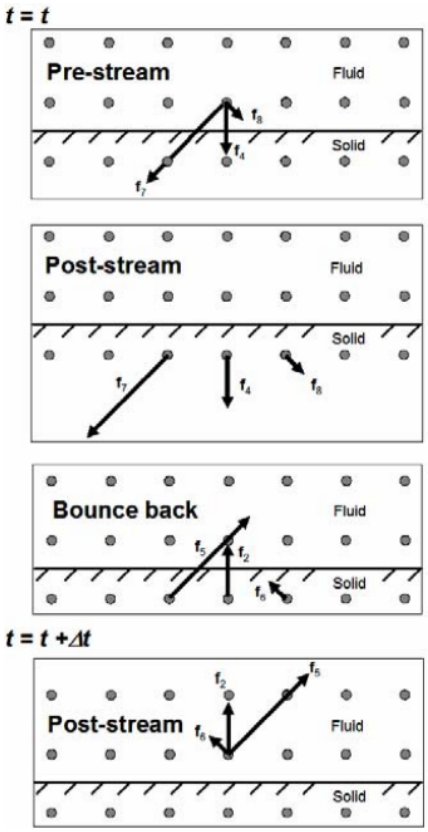
  É muito comum o uso da condição de contorno periódica que faz com que as partículas que deixam o domínio computacional por uma determinada extremidade reentrem no mesmo através da extremidade oposta. Tal condição é bastante útil quando se deseja simular um domínio com dimensão infinita em uma dada direção, pois ele se comporta como se as suas bordas opostas estivessem fisicamente conectadas.

- Não-escorregamento

  A condição de não escorregamento é alcançada pela aplicação da condição de contorno de *bounceback* nos nós sólidos. Essa abordagem torna o método bastante útil pela sua simplicidade em simular escoamentos sobre geometrias complexas, obtendo uma precisão numérica de até segunda ordem (BAO; MESKAS, 2011). Como será utilizado $\tau \approx 1$, pode-se obter bons resultados usando o esquema de *bounceback do plano médio*, pelo qual as densidades de probabilidade são temporariamente armazenadas nos nós só-

lidos para reemergirem no próximo passo de tempo, como ilustra a Figura 3 (SUKOP; THORNE, 2007).

Figura 3 – Ilustração do movimento específico de densidades conforme a direção no esquema de *bounceback do plano médio*.



Fonte: Sukop e Thorne (2007, p. 44).

- Velocidade ou pressão prescrita

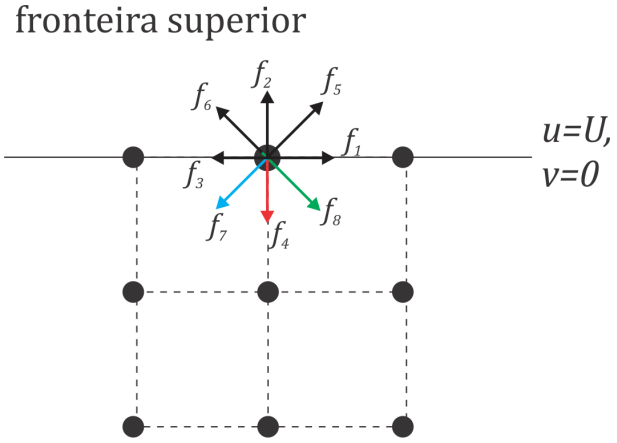
  Outra condição de contorno muito útil e de implementação relativamente simples é a de *Zou-He* (ZOU; HE, 1997). Neste tipo de BC, é possível definir uma velocidade (ou pressão) prescrita, a partir da qual são calculadas as funções distribuição que entram no domínio computacional. A densidade local do fluido e as três componentes desconhecidas da função de distribuição são determinadas pela solução de um sistema de equações lineares. Tomando-se as Equações 11 e 12, e assumindo que a regra de *bounceback* é válida para a parte de não equilíbrio da função de distribuição normal à fronteira, de modo que:

$$f_n - f_n^{eq} = f_{-n} - f_{-n}^{eq}, \tag{19}$$

  onde n representa uma função distribuição cuja direção possui componente perpendicular não nulo. Finalmente, a Equação 19 fecha o sistema de equações.

  As expressões para as quatro incógnitas variam conforme o sentido da fronteira, portanto serão apresentadas abaixo as expressões relativas ao problema da tampa móvel, que neste trabalho está localizada na extremidade superior do domínio quadrado, conforme representado na Figura 4.

Figura 4 – Ilustração da imposição da velocidade U na fronteira superior do domínio, utilizando a BC de *Zou-He*.



Fonte: Adaptado de Bao e Meskas (2011, p. 5).

Para uma velocidade qualquer $\mathbf{u} = \langle U, V \rangle$, imposta na fronteira norte do domínio, a solução do sistema linear de equações é:

$$\rho = \frac{1}{1+v}\Big(f_0 + f_1 + f_3 + 2(f_2 + f_5 + f_6)\Big) \tag{20}$$

$$f_4 = f_2 + (f_4^{eq} - f_4^{eq}) = f_2 - \frac{2}{3}\rho v \tag{21}$$

$$f_7 = f_5 + \frac{1}{2}(f_1 - f_3) - \frac{1}{6}\rho(v + 3u) \tag{22}$$

$$f_7 = f_5 + \frac{1}{2}(f_1 - f_3) - \frac{1}{6}\rho(v - 3u). \tag{23}$$

2.6 Algoritmo do LBM

O algoritmo do LBM é resumido abaixo:

1. Inicializar as variáveis ρ, $\mathbf{u}$, f_i e f_i^{eq};
2. Cálculo das *propriedades macroscópicas*, ρ e $\mathbf{u}$, a partir de f_i (Equações 11 e 12);
3. Cálculo de f_i^{eq} (Equação 14);
4. *Colisão*: cálculo da nova função distribuição $f_i' = f_i - \frac{1}{\tau}(f_i' - f_i^{eq})$ (Equação 5);
5. *Condições de contorno (BCs)*: na interface sólido-fluido (*bounceback do plano médio*); na entrada e saída (*periódica* ou *Zou-He*);
6. *Propagação*: $f_i' \rightarrow f_i$ nas direções $\mathbf{e_i}$;
7. Repetir passos 2 a 6, até que o critério de convergência seja satisfeito.

3 PROGRAMAÇÃO DE GPUS EM CUDA™

No presente capítulo, serão vistos alguns conceitos fundamentais relativos à CUDA, tais como: suas principais características de funcionamento, seu modelo de programação – delimitando-se apenas aos conceitos que mais importam à presente implementação do LBM –, aspectos específicos da arquitetura de GPUs e suas diferenças em relação a CPUs, e sucintamente também algumas das funcionalidades da sua API. Como a maior parte das informações aqui contidas deriva diretamente de NVIDIA Corporation (2016), em geral esta referência não será citada ao longo deste capítulo.

3.1 CUDA

Um processador gráfico é projetado especificamente para processar grandes conjuntos de dados gráficos (por exemplo, polígonos e pixeis) para tarefas de renderização de imagens. Recentemente as GPUs excederam o poder computacional dos CPUs, se tornado cada vez mais popular o seu uso para cálculos de propósito geral.

A CUDA é uma plataforma de computação paralela de propósito geral e modelo de programação criada pela companhia NVIDIA® para o desenvolvimento de aplicações paralelas em suas GPUs, independentemente do modelo específico de hardware. Sua finalidade é reduzir drasticamente a barreira de programação de GPGPUs, ao mesmo tempo em que permite que problemas computacionais mais complexos sejam resolvidos mais eficientemente que em uma CPU. Por essa razão e pelo alto poder computacional, essa plataforma tem ampliado cada vez mais suas possibilidades de aplicação e atraído tanto a comunidade acadêmica quanto a industrial (ZHOU et al., 2012).

O principal motivo pela discrepância na capacidade de FLOPs entre o CPU e o GPU reside em que o GPU é construído para a renderização de gráficos. Esse dispositivo é especializado em intensidade computacional e computação paralela de dados, favorecidos pela sua estrutura projetada de tal modo que mais transistores são dedicados ao processamento de dados, ao invés de *cache* de dados e controle de fluxo. Em suma, CPUs são feitas para executarem um pequeno número de tarefas complexas enquanto GPUs são projetadas para processar um grande número de tarefas simples (COOK, 2013). O seu alto ganho em performance se deve à sua capacidade de operação sobre um volume enorme de dados de maneira concorrente, sem o emprego de um grande controle de multitarefas, como é o caso dos processadores de vários núcleos (em inglês, *multicore*).

O uso de CUDA é potencializado ao máximo quando se aborda um problema dito *embaraçosamente paralelo*, ou seja, aquele em que pouco ou nenhum esforço é necessário para realizar a paralelização. Idealmente, não deve haver grande dependência de dados nem muito tráfego de comunicação entre as unidades de execução paralelas.

3.2 MODELO DE PROGRAMAÇÃO

No modelo de programação CUDA, o CPU é denominado como *host*, o qual controla toda a lógica serial e o agendador de tarefas de um programa, enquanto o GPU é considerado como um coprocessador ou *device* onde a computação paralela é realizada. A Figura 5 ilustra a execução de um programa CUDA, o qual é dividido no código do *host*, executado serialmente na CPU, e no código do *device*, o qual é paralelamente executado no GPU. A esse modelo de programação é dado o nome de computação heterogênea, pois o processamento pode ser feito de maneira independente por ambos dispositivos.

O código do *device* é organizado em um *kernel* no programa CUDA. A ele sempre é associado um *grid* de *threads*. Estas são agrupadas em conjuntos denominados blocos, nos quais as *threads* executam as mesmas instruções de código porém operam sobre porções distintas da memória, como se pode visualizar ainda na Figura 5. A *thread* é uma unidade básica de execução concorrente e é executada em uma unidade de processamento (SP, do inglês *Streaming Processor*) dentro da GPU. Os blocos são indexados conforme a posição geométrica no *grid*, sendo a cada um atribuído um único índice 3D no formato (x: coluna, y: linha, z: plano). Cada unidade de processamento é representada por uma seta ondulada e a cada chamada de *kernel* pelo *host* um novo *grid* é criado.

Neste modelo de programação, ambos *host* e *device* mantém o seu próprio espaço de memória dinâmica de acesso aleatório (DRAM, do inglês *Dynamic random-access memory*), denominados memória do *host* e memória do *device*, respectivamente. Consequentemente, um programa CUDA deve gerenciar os espaços de memória visíveis para o *kernel* através de chamadas da API para alocação e desalocação de memória bem como transferências de dados entre as memórias do *host* e do *device*.

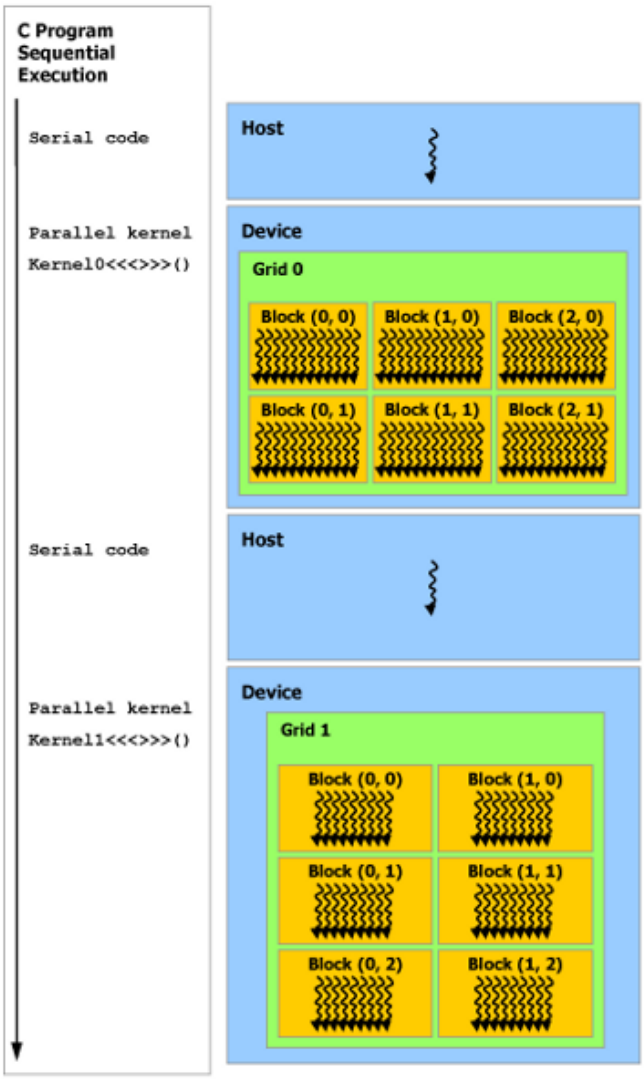
Existem vários espaços de memória na placa de vídeo e tais recursos são distribuídos conforme a hierarquia de suas estruturas. Cada tipo de memória em uma GPU possui um determinado nível de acessibilidade, o qual deve ser observado e usado a favor do programador para não sobrecarregar um determinado tipo de memória e, assim, comprometer a performance do código.

Essencialmente há três abstrações centrais nessa plataforma: uma hierarquia de grupos de *threads*, memórias compartilhadas e sincronização de barreira. Com elas se consegue aninhar um paralelismo de dados e de *threads* refinado, dentro de um paralelismo grosseiro de dados e de tarefas. Isso incentiva o programador a particionar o problema em sub-problemas grandes, os quais podem ser resolvidos independentemente e em paralelo por blocos de *threads*, onde cada sub-problema é particionado em porções menores que são resolvidas cooperativa e simultaneamente pelas *threads* dentro de um bloco (NVIDIA CORPORATION, 2016).

A CUDA estende a linguagem C permitindo ao programador definir funções chamadas *kernels*, as quais são executadas por um conjunto de *threads* simultaneamente. Estas agregadas em conjuntos chamados blocos. Os blocos são organizados em uma grade bidimensional chamada *grid* que constitui o arranjo completo das *threads* que serão criadas para um dada função no GPU, como explicita a Figura 6. A configuração da grade tem seis blocos (3×2) e cada bloco, por sua vez, tem doze *threads* (4×3).

Cada *thread* é diferenciada por um identificador único, chamado *thread ID*, dentro

Figura 5 – Ilustração do percurso de execução de um programa CUDA.

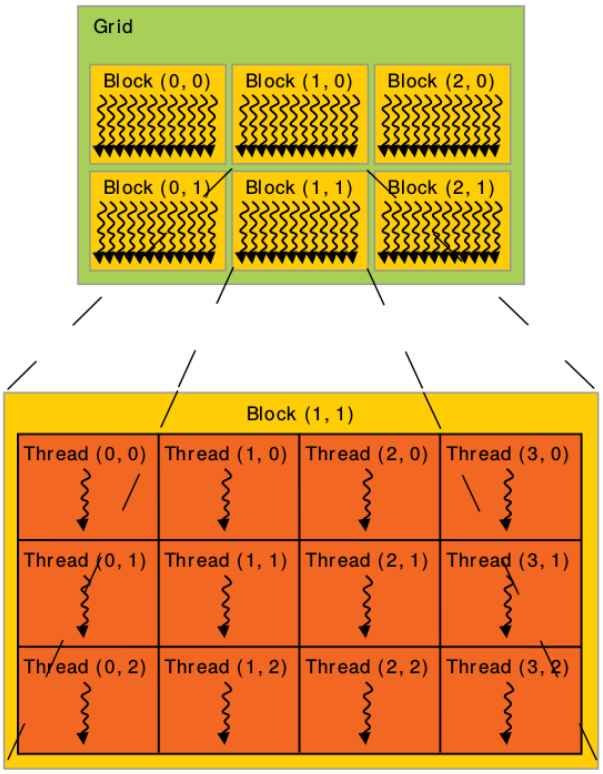


Fonte: NVIDIA Corporation (2016, p. 14).

do contexto do bloco. Esse identificador é um vetor de até três componentes (tridimensional) através do qual essas *threads* podem ser especificadas, formando um conjunto unidimensional, bidimensional ou tridimensional de blocos de *threads* (NVIDIA CORPORATION, 2016). Esses conceitos facilitam a organização das *threads* e a sua associação com os endereços de memória do hardware. Esse mapeamento é basicamente o que diferencia o processamento de uma *thread* em relação a outra, pois precisamente o mesmo código é processado por *threads* de um mesmo ramo (que não tenham sofrido divergência de fluxo por ocasião de uma instrução condicional).

O uso de estruturas de controle de fluxo, como *if* e *else*, deve ser evitado ao máximo, pois incorre em prejuízos para o paralelismo, já que as *threads* são agrupadas conforme cada condição, sendo primeiro processadas aquelas que satisfazem o *if*, e depois as restantes. Portanto não há paralelismo entre fluxos divergentes de um mesmo *kernel*.

Figura 6 – Exemplo de um *grid* com (3×2) blocos (4×3) de *threads*.



Fonte: NVIDIA Corporation (2016, p. 10).

3.3 Aspectos do hardware de GPUs

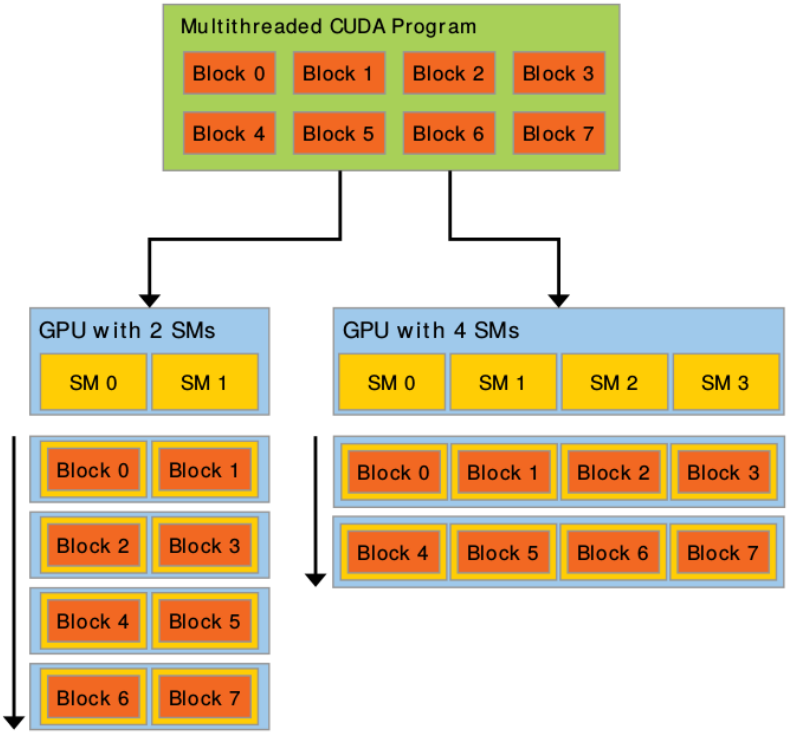
A arquitetura de GPUs da NVIDIA é constituída por um vetor escalável de multiprocessadores de fluxo (do inglês, *Streaming Multiprocessor*), tipicamente oito ou mais por dispositivo. Em outras palavras, um GPU pode ser visto como uma coleção de processadores do tipo instrução única e dados múltiplos (SIMD, do inglês *single instruction, multiple data*). O termo SIMD deriva da taxonomia de Flynn para a arquitetura de computadores. Os CPUs, por exemplo, seguem o modelo serial de programação SISD (do inglês, *single instruction, single data*), ou seja, instrução única e dado único.

A plataforma CUDA foi pensada como um mecanismo para repartir o problema em *grids* de blocos, cada qual agrupado segundo um arranjo de *threads*. Cada bloco é processado independentemente por um único SM e novos blocos são atribuídos a ele à medida em que o processamento de outros blocos é concluído. Não existe prioridade na ordem de execução dos blocos, sendo que geralmente só um subconjunto da totalidade de blocos pode ser executado de cada vez. Consequentemente, um GPU que tenha mais multiprocessadores irá processar mais blocos simultaneamente e, portanto, será executado em menos tempo que um GPU com menos SMs.

Um dos aspectos mais interessantes da CUDA é justamente o conceito de *escalabilidade transparente* que permite que a performance escale na mesma proporção em que recursos computacionais disponíveis na arquitetura do GPU são aumentados. A Figura 7 ilustra essa propriedade que é consequência da forma como se dá o processamento dos blocos no dispositivo, o qual não permite que sejam feitas barreiras de sincronização entre os blocos.

Segundo NVIDIA Corporation (2016), "O multiprocessador cria, gerencia, agenda, e executa *threads* em grupos de 32 *threads* paralelas denominados *warps*". Ao ser ordenado a

Figura 7 – Ilustração da *escalabilidade transparente* da CUDA. A seta vertical para baixo indica a passagem de tempo na execução de um programa CUDA em dois GPUs distintos, com 2 e 4 SMs respectivamente.



Fonte: Kirk e Hwu (2010, p. 7).

executar um ou mais blocos de *threads*, o multiprocessador particiona-os em *warps*, as quais são agendadas para a execução. A partição dos blocos é sempre realizada do mesmo modo, onde cada *warp* abriga *threads* com *thread IDs* crescentes e consecutivos, sendo que a primeira *warp* contém a *thread* 0. Em verdade, o processamento de blocos dentro do SM é feito em *warps*, pelas quais a leitura ou escrita de dados pode ser feita simultaneamente, desde que o acesso aos dados esteja alinhado de acordo com um padrão regular de *bytes* na memória. A esse padrão de acesso bastante eficiente dá-se o nome de coalescência, pois permite que um conjunto de dados da memória seja acessado de uma só vez. Padrões de acesso à memória que não são otimizados são denominados não-coalescidos.

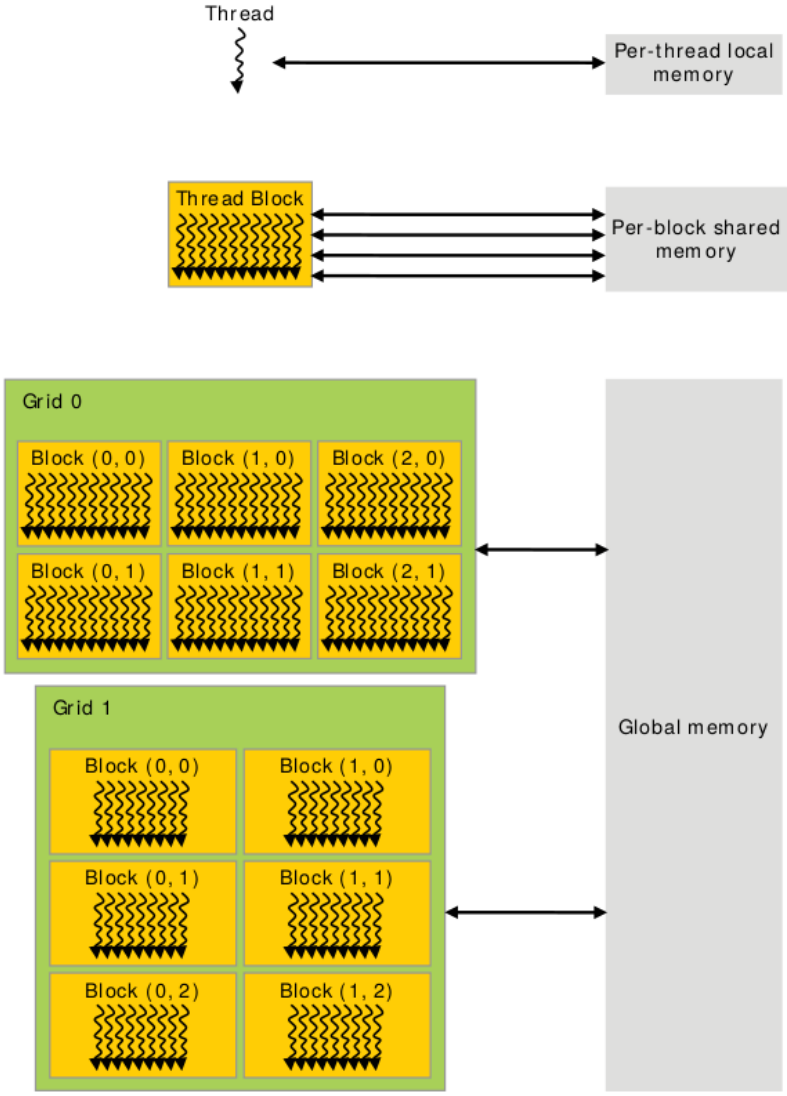
3.4 Tipos de memória

Apesar de atualmente se observar um aumento significativo dos compiladores em sua capacidade de abstrair a implementação de códigos para uma programação de alto nível, o conhecimento do funcionamento do hardware subjacente ainda é fundamental para o bom aproveitamento da performance das GPUs.

Quando um *kernel* é chamado, as *threads* de um mesmo bloco executam a mesma ação sobre sua porção respectiva do problema. Durante a existência do *grid*, *threads* de um mesmo bloco podem cooperar entre si trocando dados através da memória compartilhada, também conhecida como memória do *chip*. Para qualquer coordenação de informação entre blocos, um novo *kernel* deve ser lançado para realizar a *sincronização* das *threads* (COOK, 2013). Esse processo é ilustrado pela Figura 8, onde é explicitado cada espaço de memória que é acessível

pelas *threads* durante o seu tempo de execução.

Figura 8 – Ilustração do acesso de memória pelas *threads* durante a sua execução.



Fonte: NVIDIA Corporation (2016, p. 12).

A leitura e escrita de dados na memória principal de um computador, conhecida como DRAM (do inglês, *Dynamic Random Access Memory*), é bastante lenta comparada à velocidade do processador. Por essa razão, o rendimento de instruções processadas por segundo de um CPU é limitado pela sua *largura de banda*, ou seja, pela quantidade de dados que se pode ler ou armazenar na DRAM em um dado período de tempo (COOK, 2013). Outro aspecto importante nesse contexto é a *latência*, definida como a quantidade de tempo que se leva para efetuar um pedido de leitura. Em CUDA a latência é escondida trocando *warps*, ou seja, quando uma *warp* solicita acesso à memória, outra assume seu lugar no SM enquanto a informação não chega.

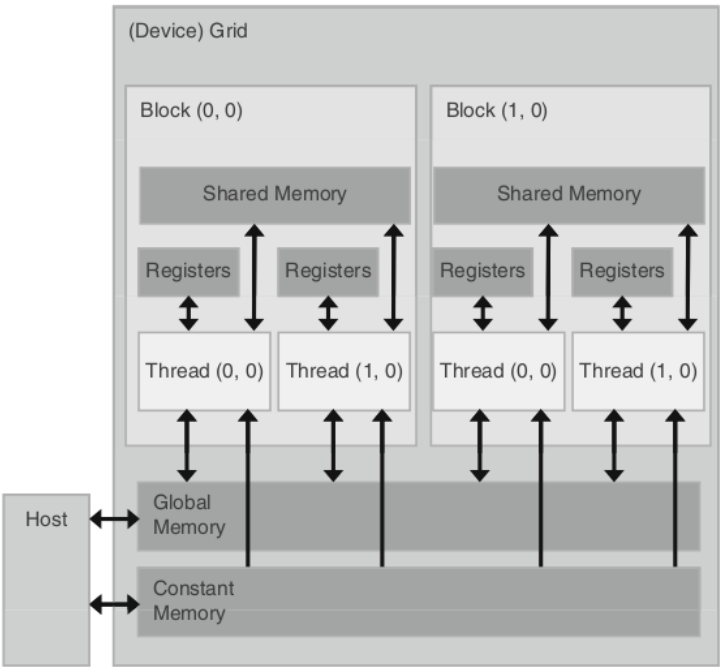
A memória global, mais lenta delas, é acessível por todas as *threads*, a compartilhada, mais rápida, apenas àquelas pertencentes a um mesmo bloco, e a local, apenas por cada *thread* individualmente da mesma forma como os ultra-rápidos registradores. Também há memória disponível para variáveis constantes e texturas, as quais podem armazenar estruturas de *arrays* de até três dimensões.

A Figura 9 demonstra a acessibilidade de dados dentro de uma GPU. A seta indica a acessibilidade dos dados, tanto para escrita quanto para leitura. Como pode ser visto, CUDA

suporta vários tipos de memória, cujos espaços podem ser lidos ou escritos por uma aplicação através de funções da API. Todos os blocos que fazem parte de um *grid* tem acesso à leitura da memória global, da memória constante e da memória de textura. A memória global é uma espécie de DRAM, portanto, possui baixa largura de banda e alta latência, sendo acessível por todas as *threads* no *grid*. A memória constante suporta baixa latência, alta largura de banda e acesso simultâneo por todas as *threads*. Os registradores e a memória compartilhada são memórias internas da placa de vídeo que podem ser acessadas a alta velocidade e de maneira altamente paralela. Os registradores são de acesso local por cada *thread* enquanto que a memória compartilhada é alocada para uso cooperativo entre as *threads* de um mesmo o bloco. Todas as memórias são acessíveis pelo GPU para acesso à escrita e leitura de dados, exceto pela memória constante, que é somente para leitura a nível do *grid* (KIRK; HWU, 2010). A hierarquia previamente explicada é reunida no esquema da Figura 9, a qual também indica por meio de setas a acessibilidade para leitura e/ou escrita de dados.

O gerenciamento de memórias no *device* é fundamental para a obtenção do desempenho máximo do dispositivo. Para tanto, faz-se necessário um balanceamento entre todos os tipos de memória. Especialmente quando se trabalha com a memória do *device* é preciso que a leitura e a escrita de dados se dê de maneira coalescida, ou seja, que as *threads* acessem os dados de maneira regular, igualmente espaçados, com o mesmo tamanho e tipo de variável. Isso permite que uma operação de leitura ou escrita de um conjunto de dados em um vetor na memória seja feita simultaneamente.

Figura 9 – Hierarquia de memórias na GPU e acessibilidade de dados.



Fonte: Kirk e Hwu (2010, p. 79).

3.5 INTERFACE DE PROGRAMAÇÃO

CUDA acompanha um ambiente de software que permite utilizar a linguagem C como uma linguagem de programação de alto nível, ocultando as particularidades do hardware aos desenvolvedores. Também outras linguagens são suportadas, tais como C++, FORTRAN, Di-

rectCompute, Java, Python, OpenACC, etc.

Para implementar um programa em CUDA, deve-se possuir uma GPU da NVIDIA habilitada para tanto e é necessário munir o computador de um conjunto de ferramentas de software, conhecido como CUDA Toolkit, o qual compreende várias bibliotecas, um *driver* e o compilador, denominado *NVIDIA CUDA Compiler* (NVCC), bem como alguns exemplos prontos.

3.5.1 CUDA API

As variáveis e funções em CUDA recebem qualificadores específicos de acordo com o local em que são armazenadas e ao seu nível de execução. O local de armazenamento de uma variável é especificado por qualificadores de tipos de variável, que em CUDA são três: *__device__*, *__shared__*, e *__constant__*.

A Figura 10 ilustra essa classificação de qualificadores, indicando o tipo de memória, o escopo de acessibilidade e o tempo de vida a que cada um se refere.

Quando nenhum dos especificadores é informado, a variável automática reside em um registrador, exceto em alguns casos em que o compilador decide colocá-la na memória local, o que pode incorrer em perda de performance.

- *__device__*

  Este qualificador declara uma variável no *device*. Pode ser usada em conjunto com os outros dois tipos de qualificadores para especificar o espaço de memória. Se nenhum dos outros dois é informado, então a variável:

  - Reside em memória global.
  - Tem o tempo de vida do *kernel*.
  - É acessível por todas as *threads* no *grid* e a partir do *host*.

- *__shared__*

  Este qualificador é usado para declarar uma variável que:

  - Reside na memória compartilhada de um bloco de *threads*.
  - Tem o tempo de vida do bloco.
  - É acessível somente por todas as *threads* do bloco.

- *__constant__*

  Qualificador usado para declarar uma variável que:

  - Reside no espaço de memória constante.
  - Tem o tempo de vida do *kernel*.
  - É acessível por todas as *threads* no *grid*.

Quando um programa CUDA é iniciado, todo o volume de dados que será processado pela placa gráfica está contido na memória do *host* e deve ser copiado para a memória do *device*, pois o *kernel* opera a partir desse espaço de memória. Portanto são providas funções para alocação, desalocação e cópia de dados na memória do *device* assim como para transferência de dados entre ambos dispositivos.

Figura 10 – Qualificadores de tipo de variáveis CUDA.

Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

Fonte: Kirk e Hwu (2010, p. 80).

- Memória global

  A memória pode ser alocada como uma memória linear ou como vetores CUDA, estes últimos sendo adequados para lidar de modo otimizado com texturas gráficas. A memória linear é alocada usando a função $cudaMalloc()$ e liberada usando $cudaFree()$ e as transferências de dados entre ambos espaços DRAM é efetuada com a função $cudaMemcpy()$. A essas funções são passados parâmetros como ponteiros para o endereço onde será alocada ou liberada a memória, o tipo e o tamanho do espaço de memória reservado, e, no caso de transferências, o sentido de cópia da memória (que pode ser do *device* para o próprio *device*, deste para o *host* e vice-versa). A alocação de vetores CUDA é feita através de $cudaMallocPitch()$ e $cudaMalloc3D()$, que são funções adequadas para alocação de vetores 2D e 3D, pois garantem que a alocação é preenchida apropriadamente para atender aos requisitos de alinhamento de memória. A declaração e a alocação deste tipo de memória é feita no código do *host*.

- Memória compartilhada

  A memória compartilhada, do inglês *shared memory*, é alocada usando simplesmente o qualificador $\_\_shared\_\_$ dentro do escopo do *kernel*. Como foi dito, esse tipo de memória é muito mais rápido que a memória global, portanto deve ser explorado sempre que haja uma oportunidade para substituir acessos à memória global. Em geral, a informação é armazenada neste tipo de memória a partir de uma leitura à memória global pelas *threads* seguida de uma atribuição à memória compartilhada declarada previamente. Esse tipo de memória não tem necessidade de ser desalocada, pois é liberada ao final da execução do bloco.

- Memória local

  Esta memória não tem necessidade de alocação nem de um qualificador específico. Consequentemente, pode ser tratada dentro do *device* do mesmo modo que é feita tipicamente a declaração de variáveis no *host* em um programa serial.

- Memória constante

  Tal memória é imutável e não pode ser modificada pelo *device* (somente leitura) e o seu valor deve ser explicitado antes do lançamento do *kernel* no código do *host*.

Para uma descrição detalhada de cada parâmetro e mais informações sobre particularidades da programação em CUDA, é recomendada a leitura no manual da NVIDIA Corporation (2016).

3.5.2 Acessos à memória do device

É preciso que haja grande intensidade computacional em um programa CUDA para se aproximar do seu limite teórico de processamento. Intensidade computacional aqui é entendida como a razão entre o número de operações de ponto flutuante realizadas para cada acesso à memória global.

O acesso à memória endereçável no GPU pode precisar ser efetuado múltiplas vezes dependendo da distribuição em que esses acessos acontecem dentro de uma *warp*. O desempenho de cada tipo de memória é afetado de maneira diferente de acordo com essa distribuição. A memória global é acessada por transações de memória de 32, 64 ou 128 *bytes*. Tais transações devem ser alinhadas, pois somente segmentos de memória de 32, 64 ou 128 *bytes* que estão alinhados ao seu tamanho (ou seja, o primeiro endereço de memória é um múltiplo do tamanho da palavra) podem ser lidos ou escritos através dessas transações. Quando uma *warp* executa uma instrução de acesso à memória global, os acessos das *threads* dentro dela são coalescidos em uma ou mais dessas transações, dependendo do tamanho da palavra acessada por cada *thread* e da distribuição dos endereços de memória ao longo das *threads*. Usualmente o aproveitamento da eficiência de leitura cai à medida em que mais transações são necessárias, já que mais palavras não utilizadas são transferidas juntamente com as palavras efetivamente acessadas pelas *threads*. O número de transações necessárias e o quanto o desempenho é afetado pelo modo de acesso varia conforme a capacidade do dispositivo. Em geral essas restrições tem sido gradativamente relaxadas conforme novas versões são lançadas e a tecnologia avança, portanto não cabe aqui entrar neste nível de detalhamento, já que é expressamente dependente do hardware com o qual se trabalha.

4 IMPLEMENTAÇÃO DO LBM EM CUDA

4.1 MODELO PROPOSTO POR TÖLKE

Segundo Tölke (2009), uma implementação otimizada do LBM em GPUs deve ser projetada de maneira distinta do que em CPUs, como consequência da diferença entre essas arquiteturas. A largura de banda da memória deve ser explorada nas GPUs, pois elas não possuem uma hierarquia de *cache* como aquela presente em núcleos de CPU. Essa natureza distinta pode ser aproveitada para esconder a latência dos acessos à memória pela subdivisão do domínio computacional em uma extensa quantidade de blocos e pelo aumento do número de operações aritméticas em relação às operações de leitura e escrita de dados. Assim, enquanto um bloco espera pela leitura de um dado na memória global, outro bloco assume o seu lugar vacante no multiprocessador. Sendo assim, é fundamental que as informações sejam armazenadas de uma maneira em que seja aproveitada a estrutura de execução paralela das *threads*.

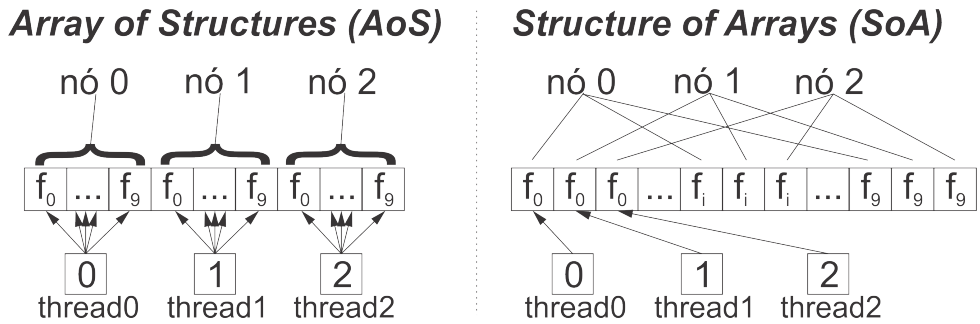
4.1.1 Estrutura de armazenamento

A taxa de transferência de dados na memória dos GPUs pode ser maximizada quando os acessos a ela ocorrem de maneira *coalescida*. A coalescência aqui se refere à leitura e escrita de dados realizada quando *threads* de um mesmo bloco acessam dados localizados em posições consecutivas na memória. Neste caso, os acessos são combinados em uma única solicitação pelo hardware (DELBOSC et al., 2014).

O padrões de acesso à memória são caracterizados segundo a forma como os dados são armazenados. Basicamente dois são os padrões mais utilizados: vetor de estruturas (AoS, do inglês *Array of Structures*) e estrutura de vetores (SoA, do inglês *Structure of Arrays*). Ambos padrões estão representados na Figura 11 enfatizando o arranjo das funções de distribuição na memória e a sequência de acesso das *threads* a esse espaço.

Para cada ponto da rede no modelo D2Q9, oito das nove velocidades devem ser propagadas para pontos não adjacentes da rede. Tipicamente na implementação em CPU, as funções de distribuição de partículas são armazenadas em um AoS. Todavia esse padrão de acesso não permite a coalescência já que as *threads* de um mesmo bloco não acessam endereços de memória consecutivos. Os acessos por cada *thread* à uma determinada função de distribuição são regularmente espaçados e não sequenciais, portanto não podem ser condensados em um único pedido de acesso à memoria. Ao invés disso os acessos são serializados reduzindo a perfor-

Figura 11 – Esquemas de armazenamento na memória.

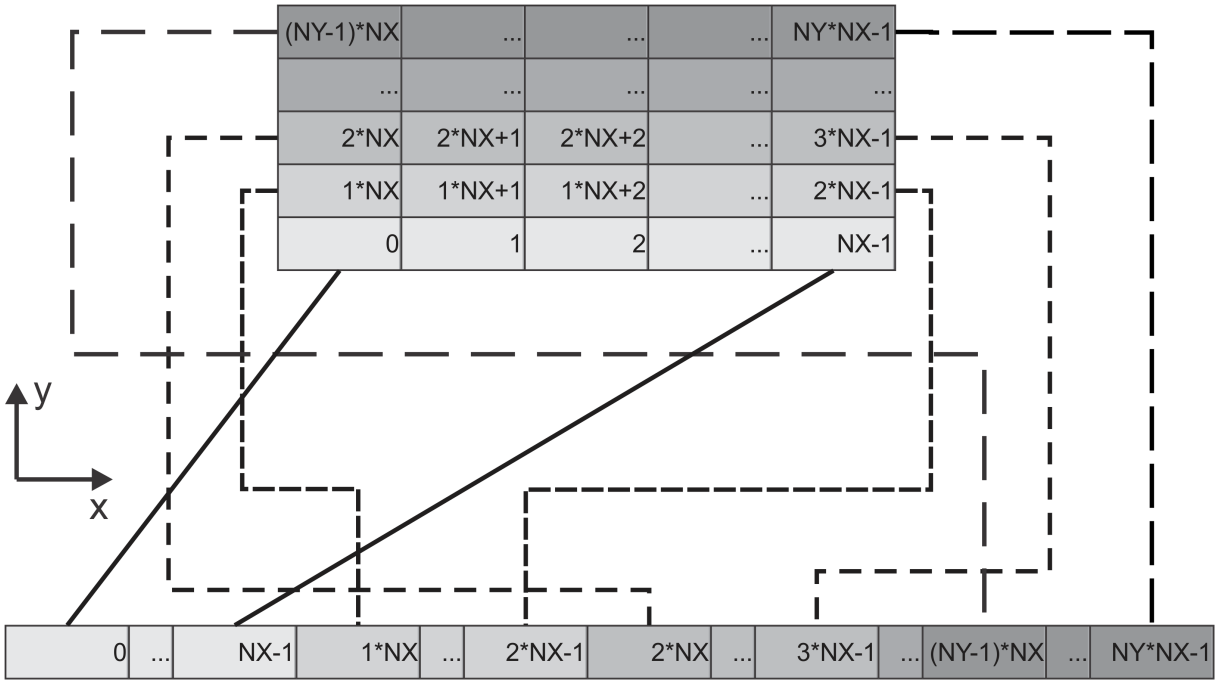


Fonte: Adaptado de (DELBOSC et al., 2014, p. 10-11).

mance do *kernel*. Por outro lado, a SoA é particularmente interessante para a implementação em GPU, pois o conjunto de funções de distribuição não está mais agrupado segundo os nós físicos a que correspondem, mas de acordo com a direção da velocidade microscópica. Portanto, os acessos à memória podem ser coalescidos em um único acesso por um *warp*. A Figura 11 ressalta as diferenças entre ambos esquemas de armazenamento.

Ao compatibilizar o arranjo das funções de distribuição na memória ao arranjo das *threads* no *grid* se garante que o acesso à memória será coalescido. Consequentemente os vetores de cada uma das direções da funções de distribuição são armazenados sequencialmente em um vetor unidimensional, conforme mostrado pela Figura 12.

Figura 12 – Ilustração da linearização dos vetores utilizados no código.

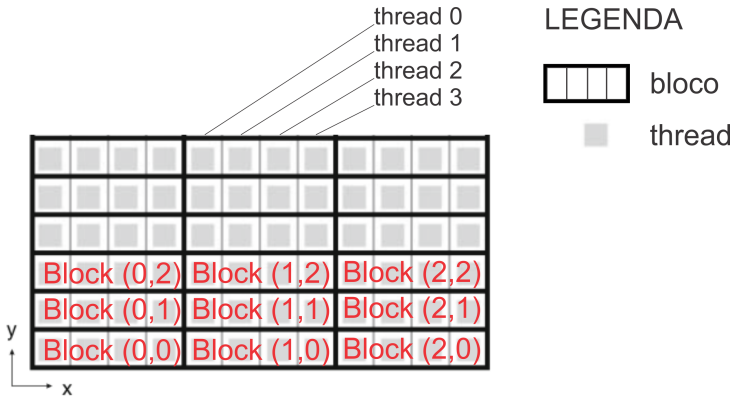


Fonte: Autor (2016).

As *threads* no *grid* 2D são organizadas em blocos unidimensionais de N_x elementos, posicionados um após o outro, como ilustra a Figura 13. Como cada *thread* está associada a um ponto no domínio físico, torna-se necessário fazer uma associação entre o *thread ID* de uma determinada *thread* ao endereço em memória global acessível por ela. Isso é feito utilizando as variáveis internas ao *kernel* que tornam disponíveis a cada *thread* do *grid* informações como o

índice do bloco ao qual ela pertence, as dimensões em do bloco (em X e Y), etc.

Figura 13 – Ilustração da indexação das *threads* e dos blocos.



Fonte: Autor (2016).

As funções de distribuição da rede serão armazenadas na memória segundo uma SoA, na qual a cada direção de velocidade do modelo D2Q9 corresponde um vetor de funções de distribuição de todos os nós do domínio.

4.1.2 Implementação dos kernels

No LBM D2Q9 as funções de distribuição devem ser propagadas para 8 direções distintas (o repouso, fR, não é propagado). São alocados no total dois conjuntos de 9 vetores unidimensionais para as funções de distribuição de partículas, um para o passo de tempo atual e outro para o passo de tempo seguinte, totalizando 18 vetores. Uma restrição é que as dimensões do domínio devem ser múltiplos de 16, para evitar que sejam processadas *warps* incompletas. Do contrário, torna-se necessário o uso de *padding*, que consiste em alocar vetores com dimensões múltiplas de 16, porém com memória de preenchimento que não será utilizada no processamento. A API do CUDA disponibiliza funções que atendem especificamente esse propósito.

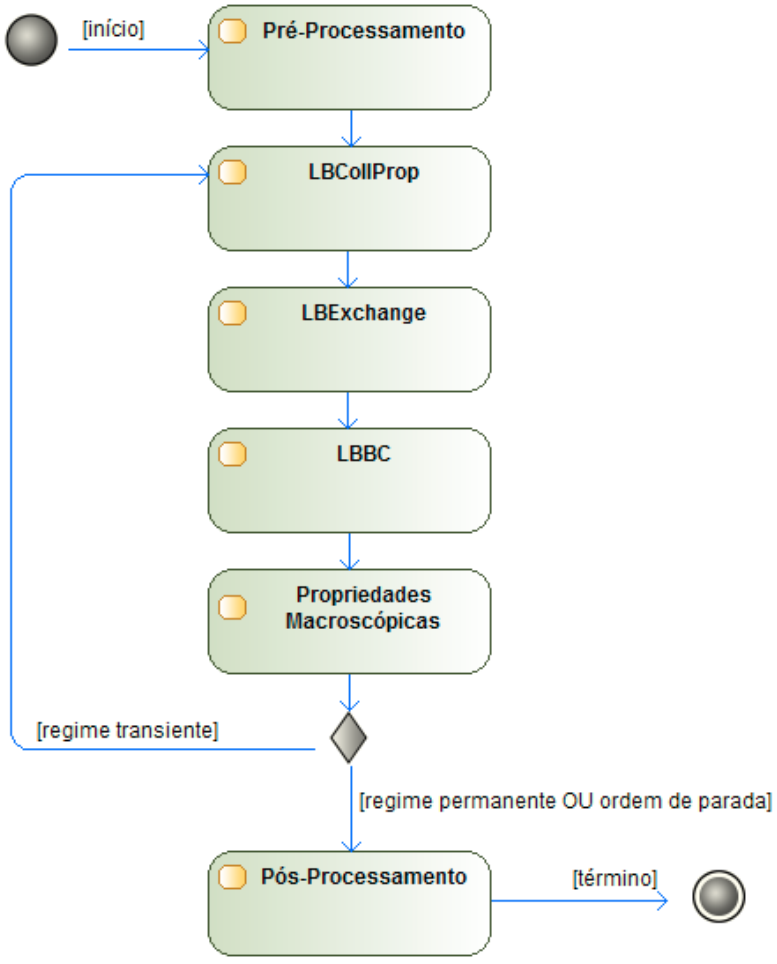
As funções propagadas ao norte (fN) e ao sul (fS) são propagadas em memória global, pois implicam em respectivamente um acréscimo e um decréscimo de uma linha completa da matriz. Já as distribuições de partículas que tem componentes ao leste (fE, fNE, fSE) e ao oeste (fW, fNW, fSW) são propagadas com um pouco mais de sofisticação usando a memória compartilhada.

Essencialmente um programa em LBM é estruturado conforme mostra o fluxograma da Figura 14 e suas etapas principais serão descritas abaixo com ênfase para a implementação nos GPUs.

- *Pré-processamento*: leitura das condições de entrada e dos arquivos de geometria. Alocações dos espaços de memória, no *host* e no *device*. inicialização das variáveis do programa (viscosidade do fluido, campo de densidade e campo de velocidades) e transferência das variáveis do CPU para o GPU. Configuração dos *kernels* e, finalmente, a entrada no laço principal, no qual são executadas as funções abaixo descritas, até que o critério de convergência seja atendido:

- $LBCollProp$: esquema de colisão e propagação parcial em memória compartilhada e global. Nesta etapa os nós fluidos realizam a colisão enquanto os nós sólidos aplicam a

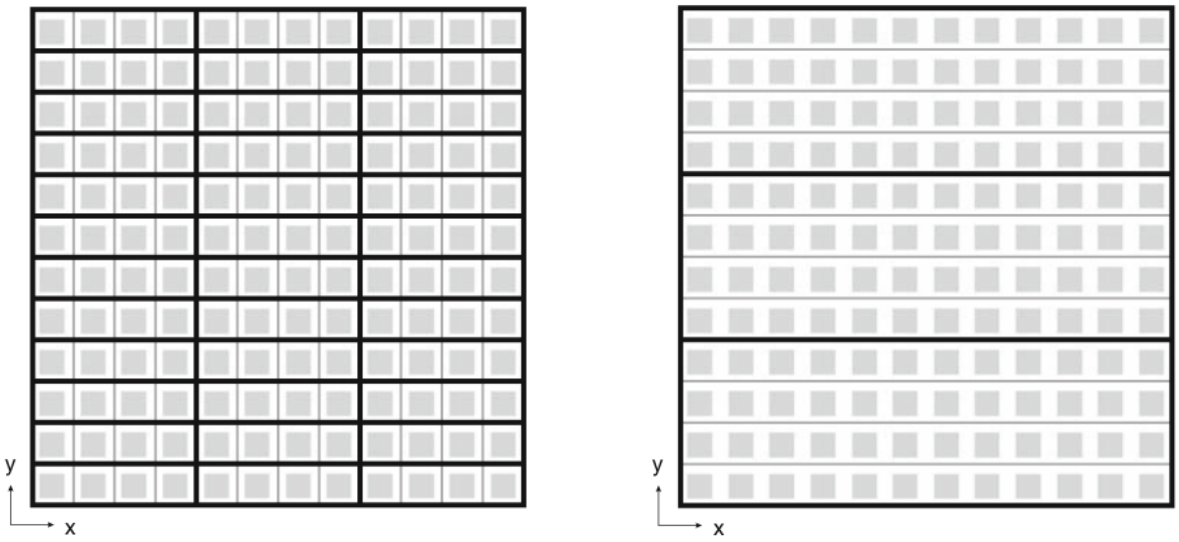
Figura 14 – Fluxograma resumido do algoritmo implementado.



Fonte: Autor (2016).

condição de contorno de *bounceback*. Na colisão as funções de distribuição não trocar informações com as células vizinhas, portanto a etapa é complemente local. A propagação é primeiramente feita em memória compartilhada, onde as funções de distribuição são transportadas para os nós vizinhos. As funções distribuição que tem componente ao oeste e ao leste são armazenadas em memória compartilhadas e deslocadas internamente para a posição adjacente conforme a sua direção. Aquelas que deveriam cruzar a fronteira de cada bloco são armazenadas na extremidade oposta do bloco e transportadas posteriormente entre blocos no próximo *kernel*. Por fim, as funções distribuição que tem componente ao norte (fN, fNW, fNE) são propagadas para a linha de nós superior e as funções distribuição que tem componente ao sul (fS, fSW, fSE) são propagadas para a linha de nós inferior.

- $LBExchange$: neste esquema, a etapa de propagação é completada em memória global. Como foi mencionado previamente, a função deste *kernel* é realizar o transporte das funções de distribuição que deveriam ter cruzado as fronteiras dos blocos na etapa de propagação.



(a) *Grid* subdivido em (3×12) blocos unidimensionais de 4 *threads* em X.

(b) *Grid* subdivido em (3×12) blocos de 4 *threads* em Y

Figura 15 – Configuração do *grid* (12×12 *threads*) em $LBCollProp$ (à esquerda) e $LBExchange$ (à direita).

- $LBBC$: este *kernel* não foi apresentado por Tölke (2009) em seu paper. Nesta implementação ocorre a imposição da condição de contorno periódica.

- *Pós-processamento*: Escrita dos arquivos de saída para os campos de densidade e velocidade bem como para o histórico do erro global do domínio. Também são gerados alguns arquivos de texto com a configuração dos *grids*, o seu tempo de execução na última iteração e o tempo de execução total do programa.

O programa foi implementado em CUDA segundo o modelo descrito nesta seção, com a inclusão de algumas funcionalidades, e consta integralmente no Apêndice A deste texto.

4.2 CONTROLE DE CONVERGÊNCIA

O erro quadrático médio normalizado ($\overline{RMSD}$, do inglês *Root Mean Square Deviation*) foi utilizado para controlar a convergência da solução usando a Equação 24 abaixo, com uma tolerância de 1×10^{-6}. Essencialmente esse erro é dado pela razão entre o desvio quadrático médio ($RMSD$) e a velocidade média do escoamento ao longo do domínio fluido:

$$\overline{RMSD} = \frac{RMSD}{\bar{u}} = \frac{\sqrt{N \sum_{i=0}^{N-1} \left[u_i(t+\Delta t) - u_i(t)\right]^2}}{\sum_{i=0}^{N-1} u_i(t+\Delta t)}, \tag{24}$$

onde o $RMSD$ é dado por:

$$RMSD = \sqrt{\frac{\sum_{i=0}^{N-1} \left[u_i(t+\Delta t) - u_i(t)\right]^2}{N}}, \tag{25}$$

e $\bar{u}$ é calculado como:

$$\bar{u} = \frac{1}{N} \sum_{i=0}^{N-1} u_i(t+\Delta t). \tag{26}$$

onde $\bar{u}$ é a velocidade média do domínio, i representa o i-ésimo ponto fluido da malha para o qual o erro está sendo calculado, x_i representa a posição desse ponto, N é o número total de nós fluidos para os quais o erro é calculado.

5 METODOLOGIA

Uma revisão bibliográfica foi realizada com o objetivo de fazer um levantamento das implementações existentes e, na sequência, escolher uma delas. Também, foram utilizados os textos de (SANDERS; KANDROT, 2010), (KIRK; HWU, 2010) e (COOK, 2013) como base para o estudo das particularidades do CUDA.

Tomou-se Tölke (2009) como referência principal para a aplicação de seu modelo de paralelização devido à simplicidade e eficiência de seu método. Seu texto se consolidou como uma das primeiras e mais importantes referências na área de estudo, pois além ser um dos primeiros a tratar especificamente da implementação de um *kernel* Lattice-Boltzmann, também apresenta em seu corpo a estrutura geral do programa por ele desenvolvido.

Por se tratar de uma abordagem numérica, neste trabalho se desenvolveu um código computacional em linguagem de programação C, com as extensões pertinentes da API do CUDA. O programa foi desenvolvido no ambiente de desenvolvimento integrado (IDE, do inglês *Integrated Development Environment*) da plataforma de desenvolvimento NVIDIA® Nsight™, o qual possui ferramentas de *debug* e *profiling* que fornecem recursos para encontrar erros e otimizar a performance do CPU e do GPU.

Tipicamente os métodos computacionais consistem de três etapas: o pré-processamento, onde os dados são preparados para a solução; o processamento, pelo qual um método numérico é aplicado para resolver o sistema de equações; e o pós-processamento, no qual os resultados do programa são salvos em arquivos para tratamento e análise posterior.

Para verificação do código foi estudado como problema um escoamento entre placas planas paralelas infinitas em regime permanente utilizando as condições de contorno periódicas previamente citadas. Para chegar na solução numérica o programa é inicializado em condição estática e um gradiente de pressão por força de corpo é imposto em todo o domínio fluido. O algoritmo é executado até que uma condição de convergência de erro quadrático médio é atingida para um critério escolhido. Para essa configuração, a condição de contorno periódica foi imposta nas bordas laterais de um domínio retangular, pelas quais o fluido escoará, e nos sítios sólidos, que definem as superfícies internas das placas (bordas superior e inferior), foi aplicada a condição de contorno de não-escorregamento. Essa abordagem fornece resultados satisfatórios, especialmente quando se utiliza um tempo de relaxação próximo de 1, valor escolhido para uma primeira estimativa (SUKOP; THORNE, 2007).

Na etapa de processamento a CPU chama as funções que resolvem a equação do LBM na GPU, denominadas kernels nesse caso. Neste trecho do código é basicamente aplicado o modelo de Tölke (2009) com algumas alterações no que se refere às condições de contorno

do problema e à maneira como os vetores de distribuição serão passados como parâmetro e referenciados dentro de cada escopo.

No pós-processamento os resultados das simulações são devolvidos para a CPU, onde são armazenados em arquivos.

Uma análise de convergência de malha foi feita para cada problema fluidodinâmico simulado. Partiu-se de uma malha quadrada 16×16, dobrando sucessivamente o número de pontos por lado a cada refinamento realizado.

O desvio médio quadrático normalizado pela média foi usado como critério de parada, com uma tolerância de 1×10^{-6}, conforme a Equação 24.

O problema do escoamento entre placas planas paralelas infinitas foi simulado utilizando as seguintes condições:

- Velocidade inicial igual a zero em todo o domínio;
- Tempo de relaxação constante: $\tau = 1.0$;
- Densidade inicial: $\rho = 1.0$
- Viscosidade constante: $\nu = 1/6$;

A Tabela 5 reúne o conjunto de simulações realizadas para Reynolds igual a 4.0.

Tabela 1 – Acelerações impostas para um escoamento em $Re = 4.0$ em função do refinamento da malha, variando de 16 a 256 nós em Y.

	Reynolds = 4.0				
L	16	32	64	128	256
g_x	4.859×10^{-4}	4.938×10^{-5}	5.595×10^{-6}	6.6654×10^{-7}	8.136×10^{-8}

Fonte: Autor (2016).

As simulações foram conduzidas com 4, 8, 16 e 32 *threads* por *bloco*. A configuração do *grid* variava de acordo com o tamanho da malha e foi estabelecida do seguinte modo:

Tabela 2 – Configuração do *grid* para os *kernel* $LBCollProp$ conforme o tamanho da malha, para blocos unidimensionais de 4 *threads*.

	$LBCollProp$				
L	16	32	64	128	256
$blocos$	$(4, 1, 1)$	$(4, 1, 1)$	$(4, 1, 1)$	$(4, 1, 1)$	$(4, 1, 1)$
$grid$	$(4, 4)$	$(8, 8)$	$(16, 16)$	$(32, 32)$	$(64, 64)$

Fonte: Autor (2016).

Na Tabela 3 abaixo estão reunidas as configurações para os *kernels* de acordo com o número de *threads* por *bloco*, N_x. É importante notar que todo o esquema de propagação da implementação de Tölke (2009) está baseado em *blocos* unidimensionais com número de *threads* múltiplos de quatro.

Para garantir que o mesmo problema está sendo resolvido pelas diferentes malhas, é preciso o tempo adimensional, $\tilde{t}$, seja constante entre as simulações. Foram realizadas as simulações acima até que o critério de convergência fosse satisfeito, e dentre elas foi escolhida

Tabela 3 – Configuração do *grid* para os *kernels* conforme o tamanho da malha.

	$LBExchange/LBBC\_xPeriodic/LBBC\_xBackup$				
L	16	32	64	128	256
bloco	$(N_x, 1, 1)$				
grid	$\left(1, \frac{16}{N_x}\right)$	$\left(1, \frac{32}{N_x}\right)$	$\left(1, \frac{64}{N_x}\right)$	$\left(1, \frac{128}{N_x}\right)$	$\left(1, \frac{256}{N_x}\right)$

Fonte: Autor (2016).

aquela com o maior tempo adimensional. O número de iterações executadas para cada uma das malhas (L) foi calculado com base no $\tilde{t}$ escolhido assim como a velocidade característica imposta pela força de corpo g_x.

Por fim, foi determinado o número de passos de tempo a serem dados como entrada ao programa de simulação usando a Equação 27:

$$N_{it} = \tilde{t}T_o = \tilde{t}\frac{L}{U} \tag{27}$$

onde N_{it} é o número de passos de tempo de simulação (dado em ts), T_o é um tempo característico da simulação, calculado como a razão entre a dimensão característica do problema, L (dada em lu), e a velocidade característica, U (dada em $\mathrm{lu/ts}$).

As simulações em tempo adimensional constante foram então comparadas com a solução analítica para a visualização da evolução do erro médio quadrático da solução numérica em relação à solução analítica, tanto em função do número de iterações, quanto em função do tamanho da malha.

Utilizaram-se *scripts* em Python juntamente com a biblioteca *matplotlib* para produzir os gráficos a partir dos arquivos de texto processados em *Python*. Os dados pós-processados da simulação foram comparados com o resultado analítico para um perfil de escoamento bidimensional entre placas planas paralelas infinitas (PRITCHARD, 2010).

Para a realização das simulações no problema da cavidade quadrada com placa móvel, utilizou-se como referência o trabalho de Ghia, Ghia e Shin (1982). Foi utilizada uma malha quadrada de 256^2, com $\tau = 1$ para $Re = 400$ e $\tau = 0.8$ para $Re = 1000$. As saídas do programa foram usadas para visualizar o escoamento no software *ParaView* (AYACHIT, 2015), traçar as linhas de corrente e o campo vetorial de velocidades.

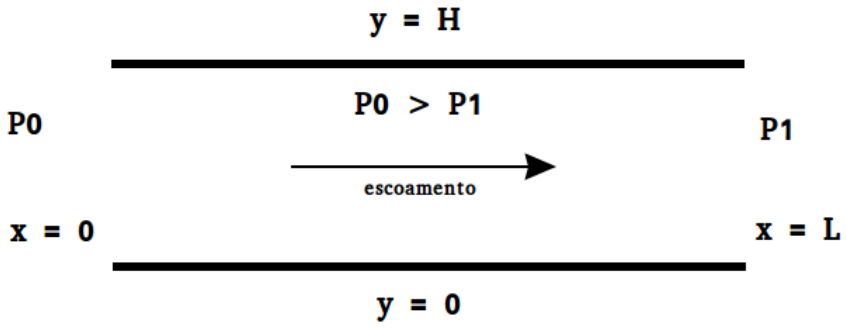
6 RESULTADOS E DISCUSSÕES

Nesta seção são apresentados os resultados numéricos obtidos com o programa implementado. Primeiramente é resolvido o escoamento entre placas planas paralelas infinitas para um número de Reynolds igual a 4.0, de modo a verificar a ferramenta desenvolvida. Também é mostrada uma solução para o problema da cavidade quadrada com tampa móvel e feita uma avaliação qualitativa a partir das linhas de corrente do escoamento.

6.1 ESCOAMENTO ENTRE PLACAS PLANAS PARALELAS INFINITAS

Escolheu-se o escoamento entre placas planas paralelas infinitas, apresentado na Figura 16, para verificação do funcionamento do programa. Essa escolha se deve à sua simplicidade bem como à existência de solução analítica, a qual é apresentada abaixo e foi usada por Bao e Meskas (2011) para comparação com os resultados numéricos do campo de velocidades.

Figura 16 – Escoamento entre placas planas paralelas infinitas com distância H e comprimento L, submetidas a um gradiente de pressão.



Fonte: Autor (2016).

As soluções analíticas são expressas a partir das Equações de Navier-Stokes, com as simplificações de que o escoamento é completamente desenvolvido, em regime permanente, isotérmico (propriedades constantes) e incompressível, de tal modo que a sua equação governante pode ser reduzida para:

$$\mu \frac{\partial^2 u}{\partial y^2} = \frac{\partial p}{\partial x} \tag{28}$$

$$\begin{aligned} u &= \frac{G}{2\mu} y\,(y-H) & \frac{\partial p}{\partial x} &= -G \\ v &= 0 & \frac{\partial p}{\partial y} &= 0 \end{aligned} \tag{29}$$

onde G pode ser um gradiente de pressão imposto, como $(P0 - P1)/L$, ou ainda gravitacional, como é o caso de placas planas verticais.

As condições de fronteira deste problema são:

$$\begin{aligned} u(x,y,0) = v(x,y,0) &= 0 \\ u(x,0,t) = v(x,0,t) &= 0 \\ u(x,H,t) = v(x,H,t) &= 0 \end{aligned} \qquad \begin{aligned} p(0,y,t) &= P_0 \\ p(L,y,t) &= P_1 \end{aligned}$$

onde P_0 e P_1 são as pressões na entrada e na saída respectivamente. No escoamento entre placas planas paralelas infinitas, foi aplicada a condição de contorno periódica nas extremidades perpendiculares à direção do escoamento, que neste caso é horizontal. Sendo assim, as bordas laterais (esquerda e direita) se comportam como se estivessem fisicamente conectadas.

6.2 Análise dos resultados

Como era esperado, os resultados obtidos variando o número de *threads* por bloco são idênticos, já que o modo pelo qual as *threads* estão agrupadas interferirá unicamente no tempo de simulação. Os resultados de todas as simulações variando o tamanho do bloco devem ser iguais, pois as mesmas operações são realizadas. A única diferença está no número de blocos programados para a execução e no modo como espaço de memória é acessado pelas *warps*. Por conseguinte, neste capítulo são apresentados apenas aqueles obtidos para um único tamanho de bloco, sendo omitidos todos os outros.

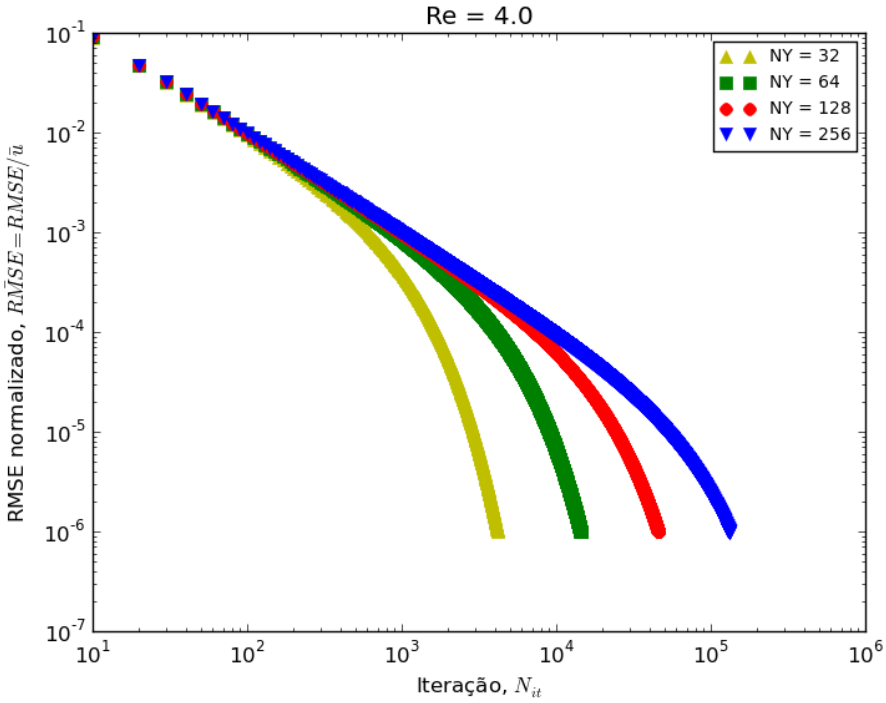
As Figuras 17 a 19 foram geradas com o critério de convergência $\overline{RMSD} \leq 1 \times 10^{-6}$. A Figura 17 tem ambos eixos em escala logarítmica e mostra a dependência do erro com o número de iterações para cada malha simulada. Ela salienta o fato de que o $\overline{RMSD}$ é proporcional ao número de pontos na dimensão vertical do domínio (NY) quando o número de iterações é mantido constante. De maneira inversa, pode-se depreender que para se conseguir um determinado nível de erro o programa precisará de mais iterações para convergir em malhas maiores.

Olhando para a Figura 18 é possível perceber que os resultados numéricos em todas as malhas reproduzem bem o perfil de velocidade da solução analítica para a condição simulada de $Re = 4$. Todavia não se pode com esses gráficos ter uma noção da magnitude relativa do erro de cada malha. Por essa razão foi construída a Figura 21, a qual apresenta os resultados da velocidade adimensionalizada do perfil numérico em relação à solução analítica adimensionalizada.

Um aspecto a se observar é que a velocidade adimensional na direção do escoamento foi normalizada nos gráficos supracitados pela velocidade média de toda a seção no domínio fluido. Por essa razão, a velocidade máxima, que deve ocorrer na linha de centro entre as placas, tem valor de 1.5, conforme dado pela Equação 29 no ponto em que $y = H/2$.

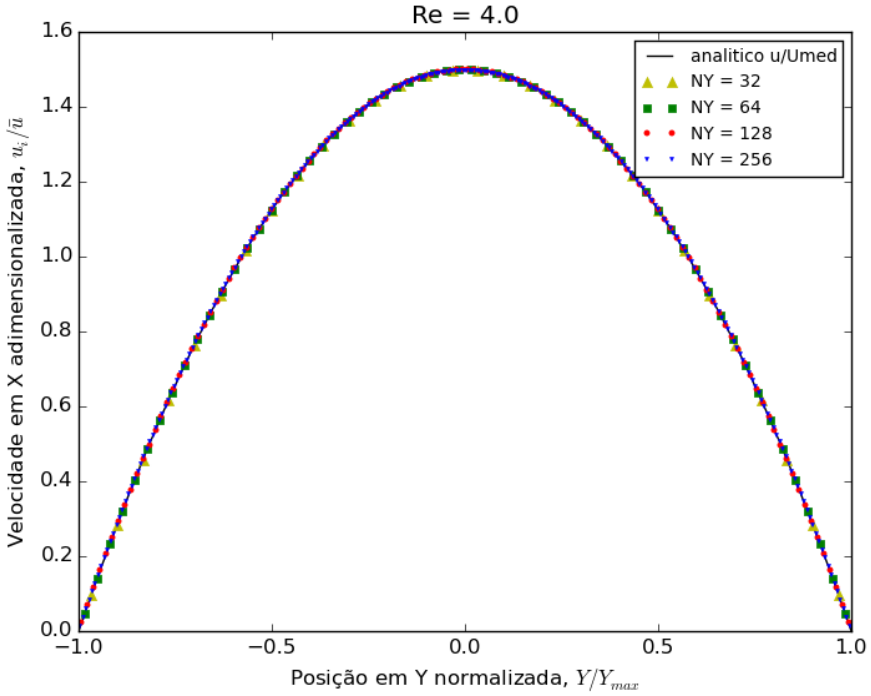
Vê-se que o erro relativo é maior nas proximidades das fronteiras sólidas para todas as malhas. Isso se deve não somente ao fato de que a condição de contorno de *bounceback* apresenta uma precisão numérica de segunda ordem, mas também ao gradiente de velocidade que é máximo nestes extremos do domínio. É fácil constatar que as curvas tem formato distinto conforme o tamanho da malha e que essa diferença se torna mais acentuada entre as curvas em que NY é igual a 32 e 256. Isso se deve à constante de tempo adimensional, $\tilde{t}$, não ser igual

Figura 17 – Evolução do $RMSD$ normalizado pela velocidade média do domínio em função do número de iterações com uma tolerância de 1×10^{-6}.



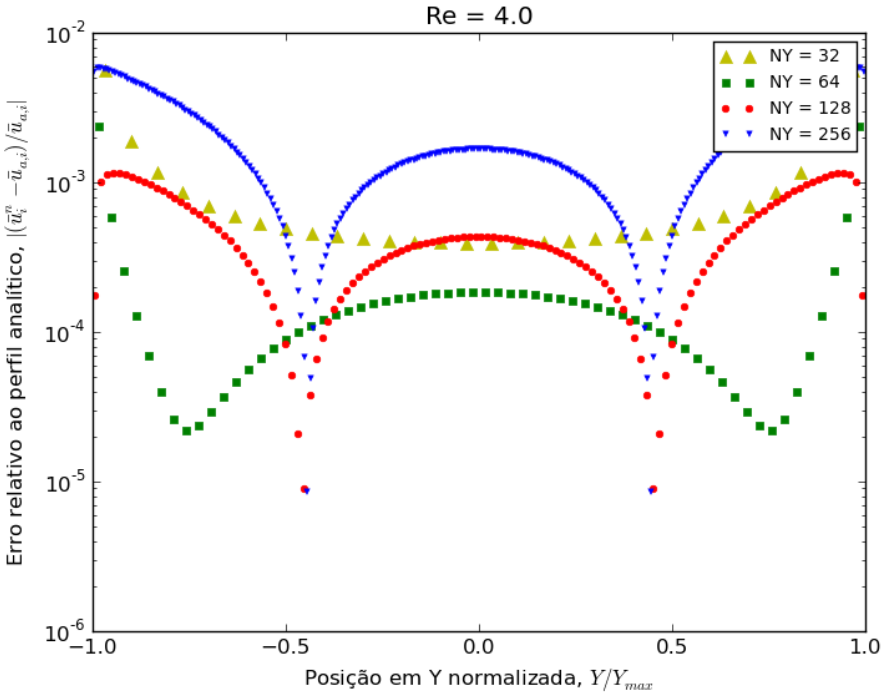
Fonte: Autor (2016).

Figura 18 – Evolução do perfil de velocidade normalizado pela velocidade média da seção em função do refinamento da malha com uma tolerância de 1×10^{-6} para o $\overline{RMSD}$.



Fonte: Autor (2016).

Figura 19 – Evolução do erro do campo de velocidades da solução numérica em relação à analítica em função do refinamento da malha com uma tolerância de 1×10^{-6} para o $\overline{RMSD}$.



Fonte: Autor (2016).

para esses casos. Como premissa para assegurar que o mesmo problema está sendo simulado computacionalmente para cada malha, é necessário que o $\tilde{t}$ não varie de uma condição para a outra.

A Tabela 4 ilustra esse fato ao apresentar no campo referente ao tempo de processamento, t_P, valores diferentes de acordo com o número de *threads* escolhido. É possível perceber que todas as variáveis apresentadas nesta tabela são dependentes do tamanho do domínio simulado. Nota-se que a constante de tempo adimensional diminui conforme o domínio é aumentado. A razão disso é que o tempo característico da simulação, T_o, que adimensionaliza o número de passos de tempo de simulação, N_{it}, é inversamente proporcional ao quadrado da velocidade, já que: $T_0 = L/U = \nu Re/U^2$. Sendo assim, é possível estimar o número de passos de tempo necessários para que o $\overline{RMSD}$ seja inferior ao critério de convergência em todas as simulações e a partir desse ponto verificar o grau de fidelidade de cada simulação em relação ao *benchmark* analítico do problema de placas planas paralelas infinitas.

Também pode ser constatado que o desempenho da configuração de 32 *threads* é consideravelmente superior uma vez que o seu tempo de execução chega a ser inferior a um terço do tempo de processamento em blocos de 4 *threads*. Esse ganho em capacidade de processamento tem origem no fato de que os acessos à memória estão ocorrendo de modo coalescido para a configuração maior em virtude do agrupamento de *warps* conter mais *threads* ativas.

Para avaliar a influência do tamanho dos blocos sobre o desempenho computacional do LBGK em CUDA, foi mensurado o tempo de execução do algoritmo do LBM em cada simulação executada. À titulo de comparação, foram registrados o número de iterações, o tempo de computação e o desempenho relativo entre as configurações de 4 e 32 *threads* por bloco. Os resultados estão reunidos na Tabela 5, que apresenta o tempo de processamento, t_P, dado em segundos.

Tabela 4 – Relação de parâmetros resultantes da execução do programa para o cálculo do tempo adimensional, em função do número de *threads* e do tamanho do domínio, com uma tolerância de 1×10^{-6} para o $\overline{RMSD}$.

Reynolds = 4.0					
Parâmetro	*threads*	L			
		32	64	128	256
U ($\mathrm{lu\,ts^{-1}}$)	$4, 8, 16, 32$	0.0222	0.0108	0.0053	0.0026
N_{it}	$4, 8, 16, 32$	4160	14380	45750	131930
$T_0 = L/U$	$4, 8, 16, 32$	1350	5766	23814	96774
$\tilde{t}$	$4, 8, 16, 32$	3.0815	2.4939	1.9211	1.3633
t_P (s)	4	1.6042	17.4341	204.5359	2354.0474
	8	1.6353	11.1018	125.2817	1415.6997
	16	0.8185	7.7176	87.3540	977.3821
	32	0.7430	6.3393	69.4195	781.1488

Fonte: Autor (2016).

Tabela 5 – Tempo computacional e avaliação da performance entre blocos de 4 e 32 *threads* para $\tilde{t} = 4$ em função do refinamento da malha .

Reynolds = 4.0					
Parâmetro	*threads*	L			
		32	64	128	256
N_{it}	$4, 32$	5400	23064	95256	387096
t_P (sec)	4	2.0786	27.9459	424.9053	6815.7936
	32	1.0313	10.0829	143.5861	2251.0360
Desempenho percentual $\left(\frac{t_P^{32}}{t_P^4}\%\right)$		49.62	36.08	33.79	33.03

Fonte: Autor (2016).

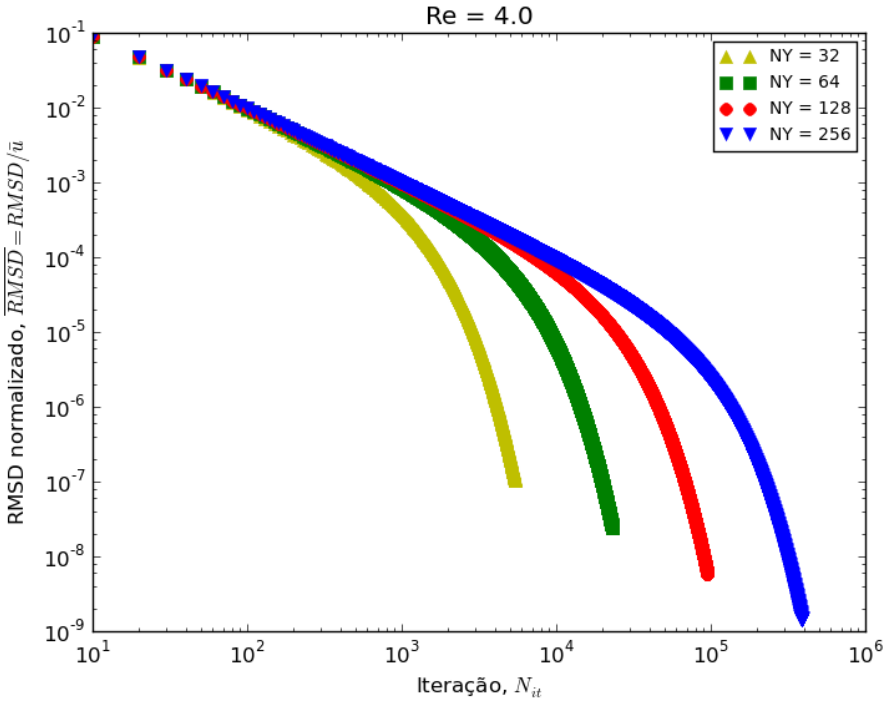
Pode-se perceber facilmente que o tempo de processamento é proporcional ao tamanho da malha. É importante notar que t_P é menor para o caso em que o número de *threads* é igual a 32. Esse é um indicativo da importância de garantir o alinhamento de dados e blocos suficientemente grandes para que as *warps* sejam totalmente ocupadas. No caso em que o número de *threads* é igual a 4 há uma grande perda em performance, pois as *warps* são executadas em grupos de 32 *threads* e isso implica que haverão 28 *threads* inativas para cada instância de bloco processado no GPU. A consequência disso é que o tempo de processamento é aumentado drasticamente, acompanhado pelo maior número de transações de memória, consequência do maior número de blocos que deve ser processado.

Outro aspecto interessante a se observar é que o desempenho na configuração de 32 *threads* aumenta na medida em que a malha é refinada, como pode ser visto pelo desempenho percentual dessa configuração de bloco em relação à de blocos de 4 *threads* apenas. Tal comportamento é expresso pela diminuição de $(t_P^{32}/t_P^4)\%$ de 49.62% na malha 32^2 pontos para 33.03% na malha de 256^2 pontos. Isso alerta para a gravidade que a escolha de uma configuração de blocos adequada representa para o desempenho global do método paralelizado em CUDA e deve ser levado em consideração para garantir que o potencial computacional do hardware está sendo aproveitado.

6.3 TESTE DE CONVERGÊNCIA DE MALHA

As Figuras 20 a 22 foram produzidas com $\tilde{t} = 4$, executando o número de iterações proporcionalmente ao tamanho da malha conforme a Equação 27.

Figura 20 – Evolução do $\overline{RMSD}$ em relação ao passo de tempo anterior em função do refinamento da malha em $\tilde{t} = 4$.

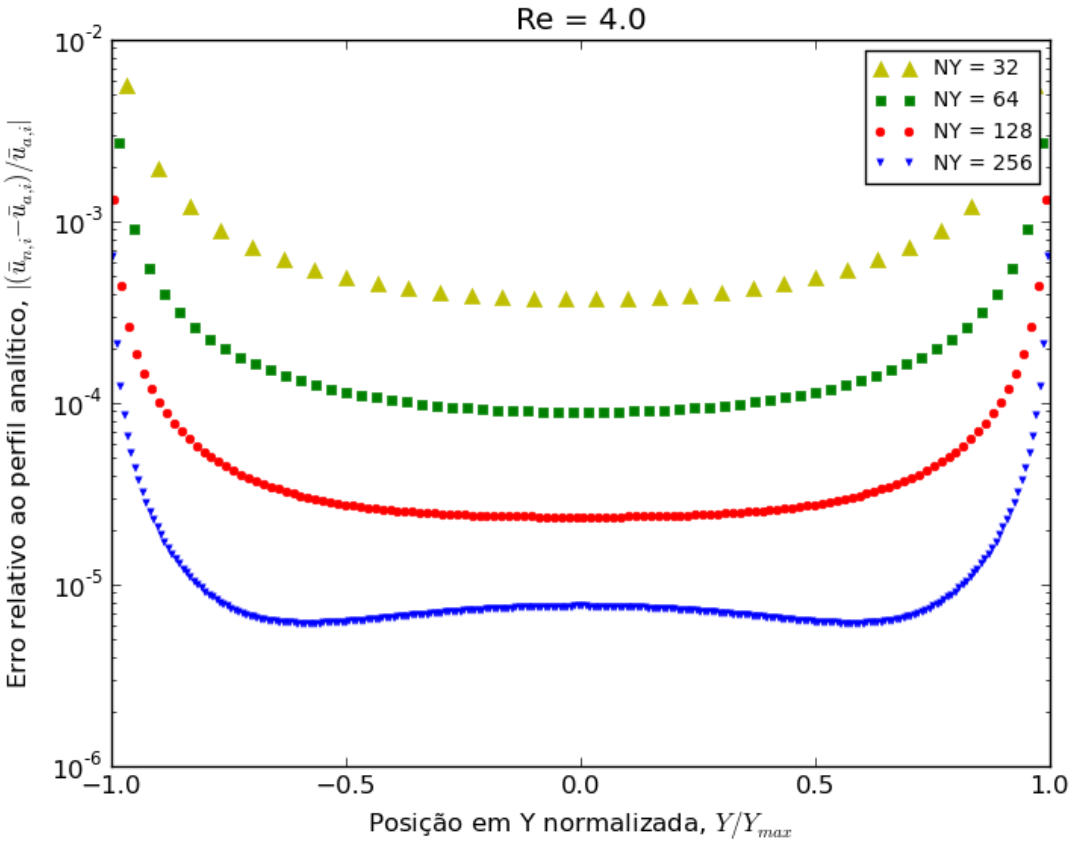


Fonte: Autor (2016).

Observou-se a concordância qualitativa do perfil de velocidade numérico para todas

as malhas com o perfil de velocidade da solução analítica. Do mesmo modo como foi feito anteriormente, foram traçados os erros relativos ao perfil analítico como mostra a Figura 21.

Figura 21 – Evolução do erro do campo de velocidades da solução numérica em relação à analítica em função do refinamento da malha em $\tilde{t} = 4$.



Fonte: Autor (2016).

A série de dados que menos se ajusta ao traço analítico é justamente aquela para a qual $NY = 32$, ou seja, a menor delas. Tal fato se justifica pela discretização mais grosseira do domínio computacional, que incorre em prejuízos para a aproximação das variações da função distribuição e é a razão pela qual normalmente se faz o estudo de convergência de malha. A velocidade u é independente da seção que se toma para a análise, visto que se trata de uma solução em regime permanente unidimensional e com condição periódica, portanto foi escolhida a primeira seção de cada simulação para a análise.

Para avaliar globalmente o grau de precisão da solução numérica relativamente à analítica, foi escolhido o erro quadrático médio (RMSE, do inglês *Root Mean Square Error*) normalizado, designado $\overline{RMSE}$, o qual é expresso como:

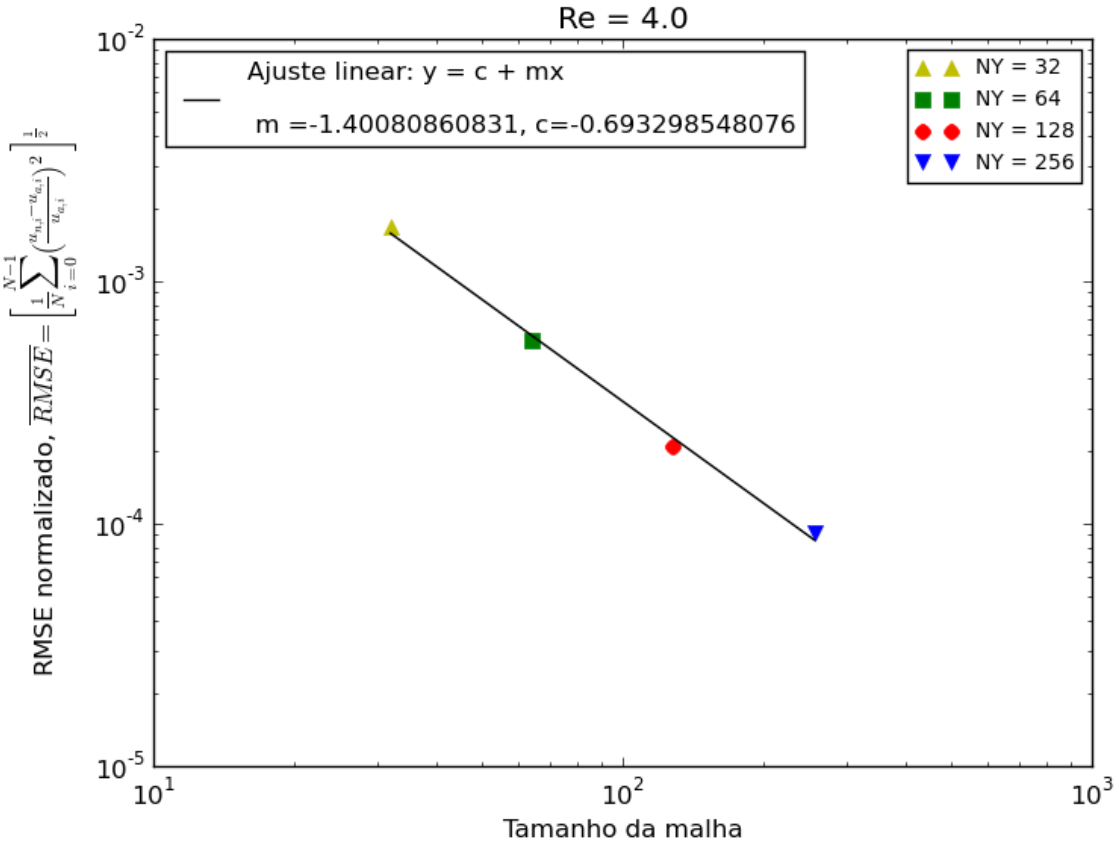
$$\overline{RMSE} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{u_{n,i} - u_{a,i}}{u_{a,i}}\right)^2}, \tag{30}$$

onde i representa o i-ésimo ponto fluido da malha para o qual o erro relativo é calculado, N é o número total de nós fluidos do domínio, $u_{n,i}$ representa a aproximação numérica para a componente X da velocidade no i-ésimo ponto e $u_{a,i}$ a componente X da velocidade na solução

analítica para este mesmo ponto.

Da Figura 22 é imediata a conclusão de que refinamento da malha corresponde a uma diminuição do $\overline{RMSE}$ relativo ao perfil analítico. Porém, como a curva está graficada em escala log-log, pela qual se verifica que o erro cai inversamente com o tamanho da malha elevado a uma determinada potência que depende do tamanho da malha.

Figura 22 – Evolução do $\overline{RMSE}$ em relação à solução analítica em função do refinamento da malha em $\tilde{t} = 4$.



Fonte: Autor (2016).

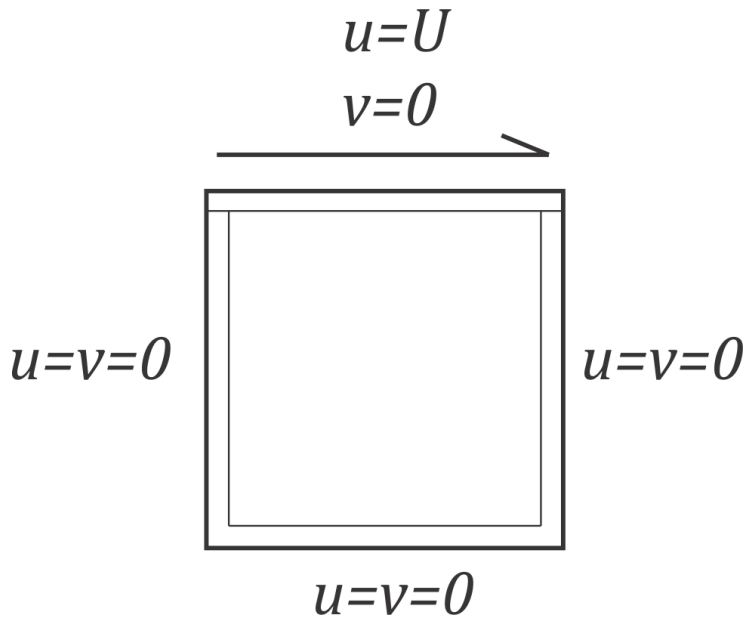
Isso pode ser observado pela dependência em potência negativa que o $\overline{RMSE}$ tem com o refinamento de malha, conforme está indicado na Figura 22. Foi feita uma regressão linear por mínimos quadrados que ajusta os dados das simulações para as figuras supracitadas. Obteve-se um coeficiente linear da reta em torno de -1.401 e uma constante da reta de aproximadamente -0.693. Tais valores indicam que o $\overline{RMSE}$ tenderá a diminuir por um fator de $10^{-1.4}$ para um aumento correspondente $\sqrt{10}$ na dimensão característica do domínio, considerando-se uma rede quadrada como as que foram utilizadas. Isso é um indicativo do fato de que a condição de contorno de *bounceback do plano médio* incorpora um erro levemente maior que de segunda ordem ao método, expresso pela potência de ordem 1.4 obtida da regressão linear. Ou seja, o programa tem comportamento de redução do erro de ordem superior à primeira ordem com o refinamento da malha. Também se constatou que um $\overline{RMSE}$ inferior a 1×10^{-4} pode ser obtido se a malha é refinada além de 256^2 pontos. Tal aproximação pode ser portanto considerada satisfatória para uma aplicação de engenharia, pois incorre em um erro inferior a 0.01%.

6.4 PROBLEMA DA CAVIDADE QUADRADA COM TAMPA MÓVEL

O problema da cavidade quadrada com tampa móvel é bastante difundido na literatura e várias soluções com resultados numéricos em malhas bastante refinadas estão disponíveis. Trata-se de um ambiente rico para se testar códigos de CFD e possui a vantagem do ponto de vista da implementação de que suas BCs são de fácil aplicação.

Basicamente ele consiste em uma calha com um tampo movido a velocidade constante U e resulta em um escoamento 2D com formação de um vórtice primário próximo do eixo longitudinal central, conforme o esquema da Figura 23. A cavidade neste problema é infinita na direção perpendicular ao plano em que o escoamento é analisado.

Figura 23 – Esquema da cavidade quadrada com tampa móvel.



Fonte: Autor (2016).

As três paredes da calha tem a condição de *bounceback* (não-escorregamento). À placa móvel foi atribuída a BC de *Zou-He* (ZOU; HE, 1997). As condições iniciais são de velocidade nula em todo o fluido e as funções de distribuição inicialmente tem valores iguais aos pesos da rede, tal que $f_i = \omega_i$. Por conseguinte a densidade inicial vale 1, conforme a Equação 11. O movimento da placa é reproduzido pela velocidade constante imposta nos nós fluidos na fronteira norte do domínio. Ambos pontos de rede nos cantos superiores são considerados parte dessa placa móvel (BAO; MESKAS, 2011). Para que o número de Reynolds se mantenha fixo, a velocidade da placa é ajustada conforme o tamanho do domínio é alterado.

As Figuras 24 e 26 apresentam as linhas de corrente obtidas para as simulações em $Re = 400$ e $Re = 1000$ respectivamente. Pode-se observar que há a formação de um vórtice na metade superior da cavidade e que esse vórtice tende a se deslocar para a direita (sentido em que a tampa é movimentada) à medida que o número de Reynolds aumenta. Além disso há um leve achatamento do vórtice central, que tem sentido de rotação horário.

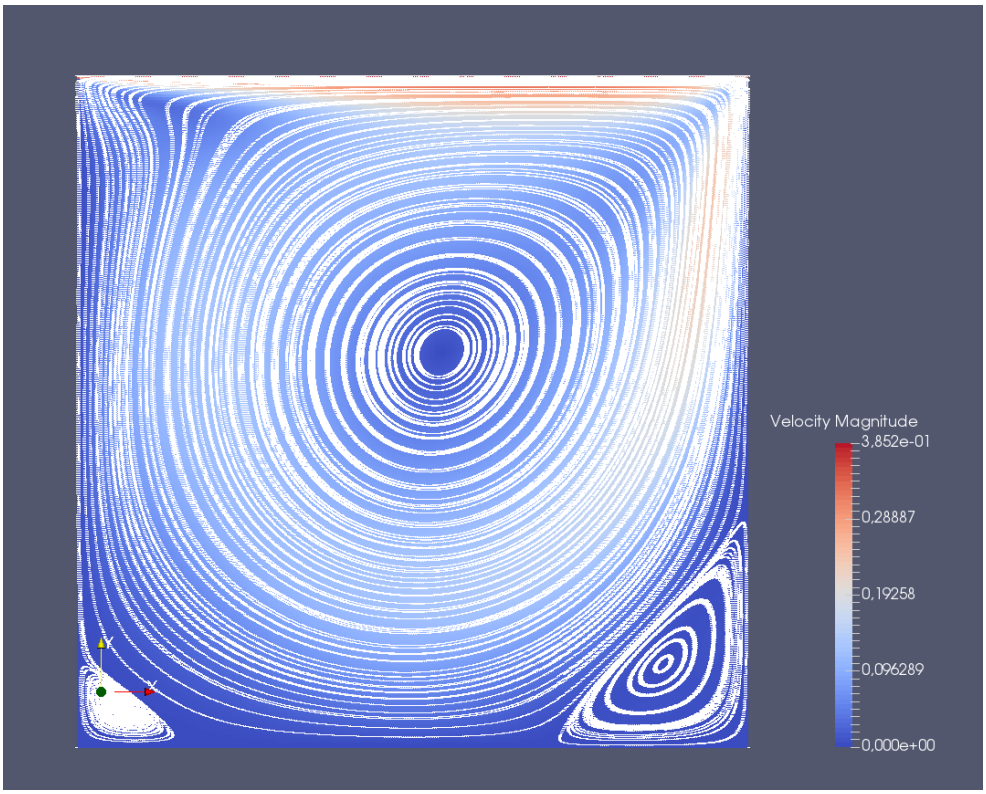
Tais resultados exibem comportamento compatível ao obtido pelas soluções numéricas de Ghia, Ghia e Shin (1982) para o mesmo problema. Em seu trabalho, Ghia, Ghia e Shin (1982) aplicam e demonstram a efetividade do método *coupled strongly implicit multigrid* para simulações de alto Reynolds em malhas refinadas. Seus resultados são apresentados nas Figuras

25 e 27, sendo a primeira um escoamento em $Re = 400$ com uma malha de 257×257 pontos e a segunda um escoamento em $Re = 1000$ com uma malha de 129×129 pontos.

A Figura 24 gerada para $Re = 400$ (com $\tau = 1$) exibe um uma configuração de linhas de corrente concordante com a solução de Ghia, Ghia e Shin (1982), conforme é apresentado na Figura 25. Além do vórtice principal há duas regiões de recirculação secundárias, uma em cada canto inferior da cavidade. Estas fluem em sentido oposto ao vórtice principal e tem raios distintos, sendo a zona da direita maior que a da esquerda. Próximo ao canto superior esquerdo as linhas de corrente tendem a se afastar em contraposição estreitamento das mesmas nas extremidades próximas ao canto superior direito. Tais gradientes de velocidades foram bem capturados bem como a proporção dos vórtices em relação às dimensões da cavidade. A recirculação central está levemente acima do centro da cavidade e é ligeiramente deformada em direção ao canto superior direito.

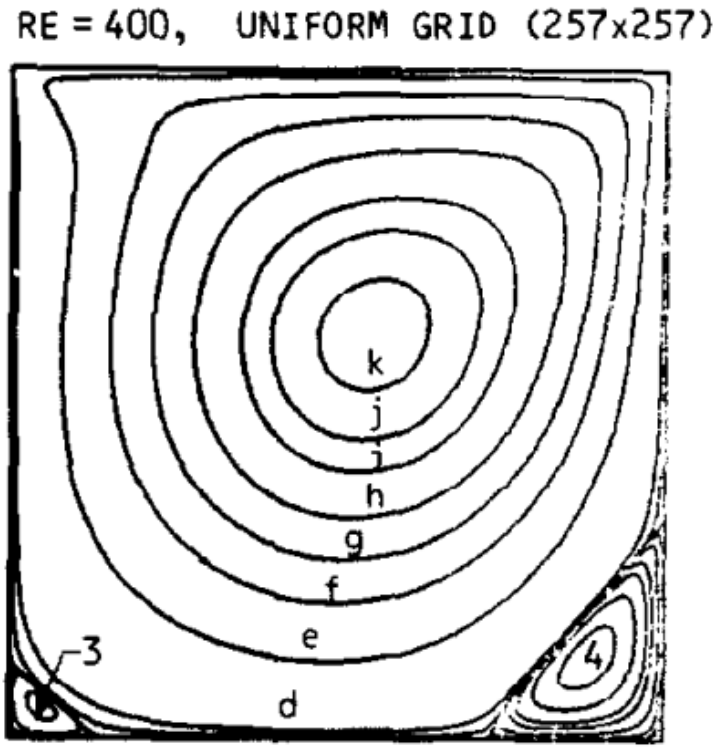
O regime de Reynolds mais alto propicia um achatamento das linhas de corrente em direção à tampa, indicando um maior gradiente de velocidades junto à mesma. Essa característica pode ser constatada pela mudança abruta do vermelho (velocidade máxima) para o branco(velocidade intermediária) na borda superior da Figura 26, obtida para $Re = 1000$ (com $\tau = 0.8$). Ainda é possível perceber que ambas as zonas de recirculação secundárias aumentam de tamanho quando o número de Reynolds é intensificado, mais notadamente aquela do canto esquerdo. Ademais, um aspecto interessante a ser notado é a constrição do vórtice principal, que passa a ser consideravelmente menor e mais arredondado que os outros dois, como exibido pela Figura 27. Por último também percebe-se que o mesmo é transladado para mais próximo do ponto central da cavidade e que as linhas de corrente acompanham o arredondamento nele observado.

Figura 24 – Linhas de corrente na seção transversal do escoamento na cavidade quadrada com tampa móvel, para $Re = 400$ e $\tau = 1$.



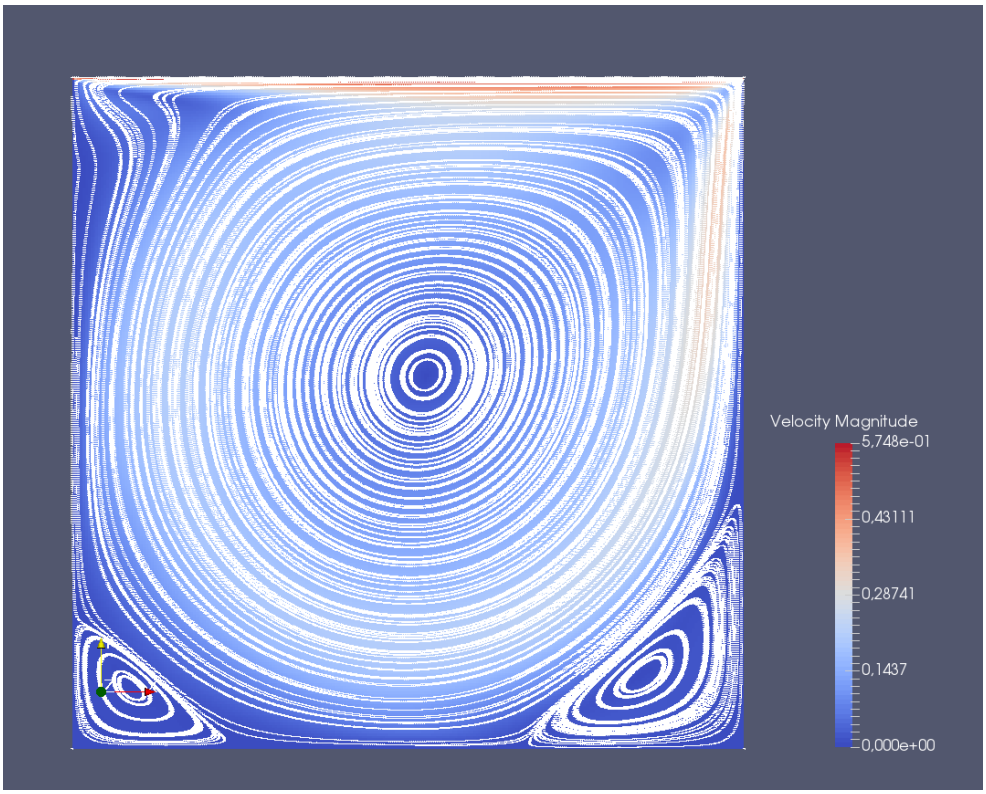
Fonte: Autor (2016).

Figura 25 – Resultado da simulação de Ghia, Ghia e Shin (1982) para o problema da cavidade quadrada com tampa móvel em $Re = 400$.



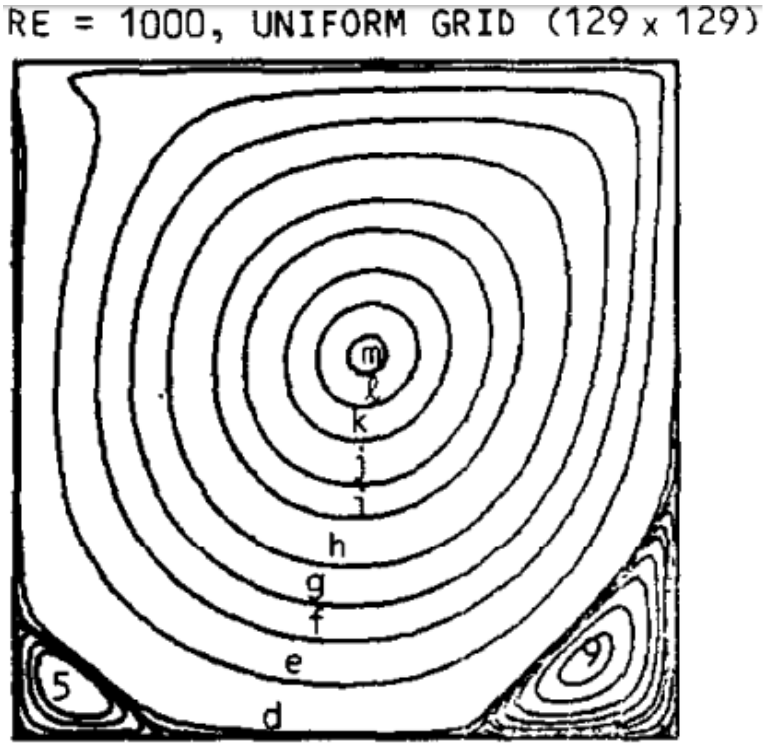
Fonte: Ghia, Ghia e Shin (1982, p. 400).

Figura 26 – Linhas de corrente na seção transversal do escoamento na cavidade quadrada com tampa móvel, para $Re = 1000$ e $\tau = 0.8$.



Fonte: Autor (2016).

Figura 27 – Resultado da simulação de Ghia, Ghia e Shin (1982) para o problema da cavidade quadrada com tampa móvel em $Re = 1000$.



Fonte: Ghia, Ghia e Shin (1982, p. 401).

7 CONCLUSÃO

Foi realizada uma implementação do esquema numérico LBGK em um modelo de rede D2Q9 usando a plataforma CUDA a fim de paralelizar o processamento do Método Lattice-Boltzmann em GPUs da NVIDIA. A tarefa paralelização deste método numérico tem o propósito de possibilitar que problemas maiores e mais complexos sejam resolvidos em relação ao que seria possível unicamente com uma abordagem CPU *multicore*. Os GPUs dispõem de uma arquitetura interna que viabiliza um ganho de processamento proporcional ao aumento de recursos computacionais investidos, o que justifica a sua popularização atualmente.

Esta implementação apresentou resultados com um bom grau de precisão para malhas testadas com até 256 nós ao longo da dimensão característica, demonstrando convergência para a solução analítica com um $\overline{RMSE}$ máximo de 4×10^{-3}. O erro relativo local se apresentou tanto maior quanto maior fosse a proximidade do nó da fronteira sólida devido ao gradiente de velocidades ser mais intenso localmente bem como ao erro associado à segunda ordem de precisão da condição de *bounceback do plano médio* utilizada. Isso explica parcialmente o porque a regressão linear mostra que o $\overline{RMSE}$ cai de maneira inversa ao tamanho da malha na potência 1.4.

Embora fosse suficiente simular um problema unidimensional em regime permanente como o de placas planas paralelas infinitas em apenas uma coluna de nós da rede, neste trabalho sua solução foi realizada em um domínio quadrado para que se pudesse averiguar a correta implementação do esquema de propagação proposto por Tölke (2009).

A presente implementação está restrita a processadores gráficos da NVIDIA, ainda que a maior fração das GPUs comercializadas no mundo advenha atualmente desta empresa. Não obstante, a presente abordagem se mostrou bastante efetiva pelo que permite que a maior parte do trabalho seja processada pela GPU, deixando a CPU livre para realizar outras tarefas.

Foi constatado que o aumento do número de *threads* por bloco foi acompanhado por uma redução no tempo de processamento, indicando a importância de garantir a coalescência de acesso à memória pelo agrupamento em blocos múltiplos de 16 *threads*, em razão do modo pelo qual é efetuada a leitura e escrita de dados pelas *warps*. Também foi visto que a performance da configuração com mais *threads* aumenta em relação à que tem menos na medida em que a malha é refinada.

Além disso, o presente trabalho se restringiu a implementar o modelo de *lattice* D2Q9, o que simplificou a programação do método e a estruturação do código. A implementação de um modelo 3D implicaria em um aumento considerável da complexidade.

O código desenvolvido neste trabalho foi verificado com o escoamento bidimensional

entre placas planas infinitas. As simulações executadas para a cavidade quadrada com tampa móvel apresentaram um comportamento satisfatório com o observado no trabalho de Ghia, Ghia e Shin (1982) para este *benchmark* de CFD e reforçam o fato de que a ferramenta é confiável para ser eventualmente aplicada em configurações de escoamentos semelhantes.

7.1 TRABALHOS FUTUROS

Com alguns melhoramentos, esta implementação pode ser bastante útil para simular cenários da engenharia aeroespacial, como escoamentos subsônicos sobre aerofólios e outras formas geométricas complexas imersas na atmosfera.

Há muito o que pode ser melhorado nesta implementação. Portanto, abaixo são sugeridas apenas algumas linhas de desenvolvimento em vistas à continuidade deste trabalho:

- Um comparativo quantitativo com o resultados de (GHIA; GHIA; SHIN, 1982) proporcionaria uma confirmação mais substancial para confirmar a confiabilidade e determinar a acurácia do programa.
- Seria muito útil a inclusão de rotinas para o cálculo de propriedades físicas do escoamento, como a tensão cisalhante, a vorticidade, etc.
- Inclusão de outras condições de contorno. Tal inclusão implicaria apenas a programação de novos *kernels* para alteração das funções de distribuição de acordo com a face de aplicação.
- Uma rotina para o cálculo do arrasto pode ser incorporada no algoritmo, dentro da etapa de *bounceback* naqueles nós sólidos que fazem interface com os nós fluidos, pela computação da variação de quantidade de movimento, que no LBM é feito sem muita dificuldade.
- Uma estudo comparativo da performance do código paralelizado em relação a um código serial, preservando as estruturas principais do método.
- Investigação da convergência e da estabilidade do código bem como um estudo a propagação dos erros numéricos e sua relação com as condições de contorno.
- No que diz respeito à otimização computacional, um estudo da ocupância da GPU poderia ser realizado bem como sucessivas alterações para melhoria da performance computacional do código numérico implementado.
- O programa ainda poderia contemplar funções para o cálculo de propriedades importantes em outros ramos da engenharia, como, por exemplo, a permeabilidade de meios porosos, parâmetros que interferem diretamente no processo extrativo na engenharia de petróleo, e até mesmo nas ciências biológicas para o estudo do efeito da porosidade sobre o escoamento medular na estrutura óssea.
- Idealmente deveria ser feita uma reestruturação do código e uma expansão para a implementação de um modelo 3D (como o D3Q19, por exemplo), já que muitos efeitos na mecânica dos fluidos são predominantemente tridimensionais.

REFERÊNCIAS

AKSNES, E. O.; HESLAND, H. *GPU Techniques for Porous Rock Visualization*. Dissertação (Mestrado) — Norwegian University of Science and Technology, Trondheim, NOR, 2009. Disponível em: <http://www.idi.ntnu.no/~elster/master-studs/aksnes-hesland-MSproj.pdf>. Acesso em: 25 nov.2016.

AYACHIT, U. *The ParaView guide: a parallel visualization application*. Nova Iorque, EUA, 2015. Disponível em: <http://www.paraview.org/>. Acesso em: 10 nov.2016.

BAILEY, P. et al. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In: IEEE. *2009 International Conference on Parallel Processing*. 2009. p. 550–557. Disponível em: <https://www.cs.arizona.edu/~pbailey/Accelerating_GPU_LBM.pdf>. Acesso em: 25 nov.2016.

BAO, Y. B.; MESKAS, J. *Lattice Boltzmann method for fluid simulations*. [S.l.], 2011. Disponível em: <http://www.cims.nyu.edu/~billbao/report930.pdf>. Acesso em: 25 nov.2016.

BHATNAGAR, P. L.; GROSS, E. P.; KROOK, M. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, American Physical Society, v. 94, p. 511–525, May 1954. Disponível em: <http://link.aps.org/doi/10.1103/PhysRev.94.511>. Acesso em: 24 nov.2016.

CHEN, S.; DOOLEN, G. D. Lattice boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, Annual Reviews, v. 30, n. 1, p. 329–364, 1998. Disponível em: <http://www.annualreviews.org/doi/abs/10.1146/annurev.fluid.30.1.329>. Acesso em: 24 nov.2016.

COOK, S. *CUDA Programming*: A developer's guide to parallel computing with gpus. 1. ed. San Francisco, EUA: Morgan Kaufmann Publishers Inc., 2013.

DELBOSC, N. et al. Optimized implementation of the lattice boltzmann method on a graphics processing unit towards real-time fluid simulation. *Computers & Mathematics with Applications*, Elsevier, v. 67, n. 2, p. 462–475, 2014.

GHIA, U.; GHIA, K.; SHIN, C. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of Computational Physics*, v. 48, n. 3, p. 387 – 411, 1982. ISSN 0021-9991. Disponível em: <http://www.sciencedirect.com/science/article/pii/0021999182900584>.

KIRK, D.; HWU, W. *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington, EUA: Morgan Kaufmann Publishers Inc., 2010.

KUZNIK, F. et al. Lbm based flow simulation using gpu computing processor. *Computers & Mathematics with Applications*, Elsevier, v. 59, n. 7, p. 2380–2392, 2010. Disponível em: <http://dl.acm.org/citation.cfm?id=1451680>. Acesso em: 24 nov.2016.

MOHAMAD, A. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. Londres, GBR: Springer, 2011.

NVIDIA CORPORATION. *CUDA C Programming Guide, versão 8.0*. 2016. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acesso em: 24 out.2016.

OBRECHT, C. et al. Efficient gpu implementation of the linearly interpolated bounce-back boundary condition. *Computers & Mathematics with Applications*, v. 65, n. 6, p. 936 – 944, 2013. Mesoscopic Methods in Engineering and Science. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0898122112004130>. Acesso em: 18 nov.2016.

PRITCHARD, P. *Fox and McDonald's Introduction to fluid mechanics*. 8. ed. Nova Iorque, EUA: John Wiley & Sons, 2010. Disponível em: <https://books.google.com.br/books?id=RdIbAAAAQBAJ>. Acesso em: 17 nov.2016.

QIAN, Y.; D'HUMIÈRES, D.; LALLEMAND, P. Lattice bgk models for navier-stokes equation. *EPL (Europhysics Letters)*, IOP Publishing, v. 17, n. 6, p. 479, 1992. Disponível em: <http://iopscience.iop.org/article/10.1209/0295-5075/17/6/001/meta>.

QIN, Z. et al. Implementation and optimization of lattice boltzmann method for fluid flow on gpu with cuda. *International Journal of Digital Content Technology & its Applications*, v. 6, n. 13, 2012. Disponível em: <http://www.globalcis.org/jdcta/ppl/JDCTA%20Vol6%20No13%20Binder1_part4.pdf>. Acesso em: 24 out.2016.

SANDERS, J.; KANDROT, E. *CUDA by Example*: an introduction to general-purpose gpu programming. 1. ed. Boston, EUA: Addison-Wesley Professional, 2010.

SUKOP, M. C.; THORNE, D. T. *Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers*. 1. ed. University Park, Miami FL 33199, USA: Springer Publishing Company, Incorporated, 2007. Disponível em: <https://books.google.com.br/books/about/Lattice_Boltzmann_Modeling.html?id=35SWbKnPb5IC&redir_esc=y>. Acesso em: 24 nov.2016.

TÖLKE, J. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. *Comput. Vis. Sci.*, Springer-Verlag, Berlin, Heidelberg, v. 13, n. 1, p. 29–39, nov 2009. Disponível em: <http://dx.doi.org/10.1007/s00791-008-0120-2>. Acesso em: 16 out.2016.

XIONG, Q. et al. Efficient parallel implementation of the lattice boltzmann method on large clusters of graphic processing units. *Chinese Science Bulletin*, v. 57, n. 7, p. 707–715, 2012. Disponível em: <http://dx.doi.org/10.1007/s11434-011-4908-y>. Acesso em: 23 out.2016.

ZHOU, H. et al. Gpu implementation of lattice boltzmann method for flows with curved boundaries. *Computer Methods in Applied Mechanics and Engineering*, v. 225–228, p. 65 – 73, 2012. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0045782512000874>. Acesso em: 20 out.2016.

ZOU, Q.; HE, X. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. *Physics of Fluids*, AIP Publishing, v. 9, n. 6, p. 1591–1598, 1997. Disponível em: <http://scitation.aip.org/content/aip/journal/pof2/9/6/10.1063/1.869307>. Acesso em: 24 nov.2016.

APÊNDICE A - CÓDIGO COMPUTACIONAL DESENVOLVIDO EM LINGUAGEM C COM API CUDA

```
/*
 ================================================================================

 Name        : cuda_LBGK_D2Q9.cu
 Author      : Denis Leite Gomes <denislgplus@hotmail.com>
 Version     : 1.0 (last modified on: 29/11/2016)
 Copyright   : Your copyright notice
 Description : This program implements the LBM BGK 2DQ9 in CUDA
 *               based on the implementation of Tôlke (2009), and also
 *               includes Periodic and Zou-He BC kernels.
 ================================================================================

*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>

#ifndef MAX                // Macro for returning the maximum value
#define MAX(a, b) (a > b ? a : b)
#endif

#define xstr(x)   str(x)   // Stringification after macro expansion
#define str(x)    #x       // Stringification macro
//
   ================================================================================

//  Configuration section - Set geometry, block size, error, interval
    & problem
#define WIDTH      256    // Channel width:                    16  / 32
     / 64  / 128 / 256
#define NX         WIDTH // Number of grid nodes in X direction
#define NY         WIDTH // Number of grid nodes in Y direction
#define N_THREADS 32     // Num  ber of threads per block
#define MAX_ERROR 1e-6//5.e-9// Maximum error tolerated
#define MAX_STEPS 1e7// Maximum number of allowed steps:
   1176/5400/23064/95256/387096
#define DEL_TXT    10000 // TXT file printing interval
#define DEL_VTK    1000  // VTK file printing interval
#define DEL_ERR    100   // ERR file printing interval

#define RE     1000.0// Reynolds Number
#define RHO    1.0    // Density for solids and fluids
#define TAU    0.8    // Relaxation Time
// Body acceleration for Poiseuille Flow (from analytical solution)
#define GX     0    //1./3.*RE*(2.*TAU-1.)*(2.*TAU-1.)/((NY-2.)*(NY-2)
   *(NY-2.))

#define UX0    RE*(2.*TAU-1.)/6./(NY-2) // Velocidade horizontal da
```

```
    placa
#define UY0    0    // Velocidade vertical da placa

#define FILENAME   xstr(NX) "x" xstr(NY) ".dat"  // Geometry filename
//
    ================================================================================

#define FLUID 1       // Fluid node identifier
#define SOLID 0       // Solid node identifier
#define TRASH 3       // Thrash node identifier
#define PAD   2       // Number of padded rows
#define YP    1       // Pad thickness (in number of rows)
#define N     (NX*NY)      // Fluid domain for Poiseuille Flow
#define NP    (N+PAD*NX)   // Total number of nodes (including thrash)
#define INI   (NX)   // Index for the beginning of the physical domain
#define END   (NX+N)// Index for the end of the physical domain

// Distribution Function indexes - SoA arrays of NP size:
// [F]=[FR,FE,FN,FW,FS,FNE,FNW,FSW,FSE]    |''''''''''''''''''''''''''|
#define FR  NP*0  // Rest particles        | FNW[6]  FN[2]    FNE[5] |
#define FE  NP*1  // East particles        |    |\`    /|\   ´/|     |
#define FN  NP*2  // North particles       |         \   |   /        |
#define FW  NP*3  // West particles        |           \ | /          |
#define FS  NP*4  // South particles       | FW[3]<---FR[0]--->FE[1] |
#define FNE NP*5  // North-east particles  |           / | \          |
#define FNW NP*6  // North-west particles  |         /   |   \        |
#define FSW NP*7  // South-west particles  |      |/_   \|/   _\|     |
#define FSE NP*8  // South-east particles  | FSW[7]   FS[4]   FSE[8] |
#define NV  9     // Number of velocities  |_________________________|

// Macro function for checking error code returned from CUDA API
#ifndef CUDA_CHECK
#define CUDA_CHECK(value) {                                          \
  cudaError_t _m_cudaStat = value;                                   \
  if (_m_cudaStat != cudaSuccess) {                                  \
    fprintf(stderr, "\nError %s at line %d in file %s\n",     \
        cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__); \
    exit(1);}}
#endif // CUDA_CHECK

//--- Utilities ---
// Array structure and functions for working with arrays
typedef struct {double *array; size_t used; size_t size;} Array;
__host__ void initArray(Array*, size_t);    // Initialize array of
    certain size
__host__ void insertArray(Array*, double); // Insert data into the
    array
__host__ void freeArray(Array*);           // Free array memory
    space
__host__ int countFluid(int*);             // Count fluid nodes
    inside the mesh
```

```
//--- File utilities ---
// DAT file for simulation outputs
__host__ void simulationOutputs(int, dim3, dim3, dim3);
// Execution time of CUDA Kernels
__host__ void timeKernel(clock_t*, clock_t*, double*);
// Geometry information storage in Host memory from input "YxX.dat"
   file
__host__ void store_geo(int*, char*);
// VTK file generator
__host__ void vtk(int, int*, double*, double*, double*);
// TXT file generator
__host__ void txt(int, int*, double*, double*, double*);
// ERROR file generator
__host__ void errorFile(Array*, Array*);

//--- Initial Configuration and Conditions ---
__host__ void setCudaMaxDevice(void); // Set the best available GPU
// Initial density initialization in Host memory
__host__ void ini_rho(int*, double, double*);
// Initial velocity field initialization in Host memory
__host__ void ini_vel_field(int*, double*, double*, double*, double*)
   ;

//--- Macroscopic Properties ---
// Equilibrium distribution function calculation
__host__ void f_eq(int*, const double*, double*, double*, double*,
    const double*, double*);
// Macroscopic variables recovery
__host__ void macroscopic(int*, double*, double*, double*, double*);

//--- LBM kernel functions ---
// Kernel for collision, bounceback and partial propagation (1st,
   shared memory, 2nd, global memory)
__global__ void LBCollProp(int*, double*, double*);
// Kernel for correction of previous propagation across blocks (in
   global memory)
__global__ void LBExchange(int, double*);
// Kernel for backing up information that will be overwritten in
   LBExchange
__global__ void LBBC_xBackup(double*, double *);
// Kernel for dealing with periodic boundaries
__global__ void LBBC_xPeriodic(double*, double*, double*);
// Kernel for applying ZouHe BCs, on the negative X plane
__global__ void LBBC_ZouHeNorth(double*, double, double);

//--- Constant memory declaration ---
__constant__ double a_D[2], w_D[3]; // Acceleration and weights

int main(int argc, char** argv) {
  setCudaMaxDevice();    // GPU Device selection
```

```
  // Weights for LBCollProp() collision and copy to DEVICE
const double w_H[3] = { 4./9., 1./9., 1./36. };
CUDA_CHECK(cudaMemcpyToSymbol(w_D, w_H, sizeof(double)*3));
// Body accelaration - used after collision in LBCollProp
const double a_H[2] = { GX, 0.0 };
CUDA_CHECK(cudaMemcpyToSymbol(a_D, a_H, sizeof(double)*2));

char geo_name[] = FILENAME;
// Geometry allocation and pointing - HOST memory
int *geo_H = (int*) malloc(sizeof(int)*NP);
store_geo(geo_H, geo_name); // Save to geometry file
int NF = countFluid(geo_H); // Count number of fluid nodes

// Geometry allocation and pointing - DEVICE memory
int *geo_D;
CUDA_CHECK(cudaMalloc((void**)&geo_D, sizeof(int)*NP));
CUDA_CHECK(cudaMemcpy(geo_D, geo_H, sizeof(int)*NP,
    cudaMemcpyHostToDevice));

// Old and new Distribution Functions - HOST memory allocation
double *f0_H = (double*) malloc(sizeof(double)*NP*NV);
double *f1_H = (double*) malloc(sizeof(double)*NP*NV);
double *fT_H = (double*) malloc(sizeof(double)*NP*NV);

// Old and new distribution function - DEVICE memory allocation
double *f0_D, *f1_D, *fT_D;
CUDA_CHECK(cudaMalloc((void**)&f0_D, sizeof(double)*NP*NV));
CUDA_CHECK(cudaMalloc((void**)&f1_D, sizeof(double)*NP*NV));
CUDA_CHECK(cudaMalloc((void**)&fT_D, sizeof(double)*NP*NV));

// Auxiliary pointers: each element will point to each of the 9
    arrays
double *fOld[NV], *fNew[NV];

// Distribution Function used in LBBC_xBackup - DEVICE memory
    allocation
double *fB_D;
CUDA_CHECK(cudaMalloc((void**)&fB_D, sizeof(double)*NY*6));

// Density field - HOST memory allocation
double *rho_H = (double*) malloc(sizeof(double)*NP);
// Current Velocity field - HOST memory allocation
double *vx_H = (double*) calloc(NP, sizeof(double));
double *vy_H = (double*) calloc(NP, sizeof(double));
// Previous Velocity field - HOST memory allocation
double *vx_o_H = (double*) calloc(NP, sizeof(double));
double *vy_o_H = (double*) calloc(NP, sizeof(double));

// Density and Velocity field - DEVICE memory allocation
double *rho_D, *vx_D, *vy_D;
CUDA_CHECK(cudaMalloc((void**)&rho_D, sizeof(double)*NP));
```

```
CUDA_CHECK(cudaMalloc((void**)&vx_D, sizeof(double)*NP));
CUDA_CHECK(cudaMalloc((void**)&vy_D, sizeof(double)*NP));

// HOST arrays for Velocity Profile - Velocidade nula
double *vx_prof = (double*) calloc(NY, sizeof(double));
double *vy_prof = (double*) calloc(NY, sizeof(double));

// Density field initialization - HOST memory
ini_rho(geo_H, RHO, rho_H);
// Copy Density field: HOST --> DEVICE
CUDA_CHECK(cudaMemcpy(rho_D, rho_H, sizeof(double)*NP,
    cudaMemcpyHostToDevice))

// Velocity field initialization - HOST memory
double ux_lid = UX0, uy_lid = UY0;
printf("%f \t %f", ux_lid, uy_lid);

// Distribution function field initialization - Initial condition:
    equilibrium at rest
f_eq(geo_H, a_H, vx_o_H, vy_o_H, rho_H, w_H, f0_H);

// Initially: Current Distribution Function = Previous Distribution
     Function
memcpy(f1_H, f0_H, sizeof(double)*NP*NV);
memcpy(fT_H, f0_H, sizeof(double)*NP*NV);

// Copy Distribution function field: HOST --> DEVICE
CUDA_CHECK(cudaMemcpy(f0_D, f0_H, sizeof(double)*NP*NV,
    cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(f1_D, f1_H, sizeof(double)*NP*NV,
    cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(fT_D, fT_H, sizeof(double)*NP*NV,
    cudaMemcpyHostToDevice));

// Auxiliary variables for convergence
Array error, error_;  // Arrays for RMS and normalized RMS errors
double relError, rmse, rmse_;   // Auxiliary variables for error
    calculations

// GRID Configuration
dim3 block(N_THREADS, 1, 1);     // 1D block of N_THREADS threads
dim3 grid(NX / N_THREADS, NY);   // 2D grid of NX/N_THREADS by NY
    blocks
dim3 grid1(1, NY / N_THREADS);   // 2D grid of 1 by NX/N_THREADS
    blocks

bool converged = false; // Logical variable for loop execution
    control
int step = 0;           // Starting time step

double t_sec[5];  // Varible for storing time in milliseconds
```

```
  clock_t start[5], stop[5]; // CUDA events for execution time

  // Error arrays initialization
  initArray(&error, 1); initArray(&error_, 1);
  // Time array initialization
  start[0] = clock(); // Start timer
  double v_sum = 0;

  while (!converged && step <= MAX_STEPS) { // Converges with error
     or stops

    memcpy(f0_H, f1_H, sizeof(double)*NP*NV);
    // Current vector of this step becomes the old vector for the
       next step
    if (step % 2 == 0) {// Even Step
      fOld[0] = &f0_D[FR]; fOld[1] = &f0_D[FE]; fOld[2] = &f0_D[FN];
      fOld[3] = &f0_D[FW]; fOld[4] = &f0_D[FS]; fOld[5] = &f0_D[FNE];
      fOld[6] = &f0_D[FNW];fOld[7] = &f0_D[FSW];fOld[8] = &f0_D[FSE];
      fNew[0] = &f1_D[FR]; fNew[1] = &f1_D[FE]; fNew[2] = &f1_D[FN];
      fNew[3] = &f1_D[FW]; fNew[4] = &f1_D[FS]; fNew[5] = &f1_D[FNE];
      fNew[6] = &f1_D[FNW];fNew[7] = &f1_D[FSW];fNew[8] = &f1_D[FSE];
    } else {      // Odd Step
      fNew[0] = &f0_D[FR]; fNew[1] = &f0_D[FE]; fNew[2] = &f0_D[FN];
      fNew[3] = &f0_D[FW]; fNew[4] = &f0_D[FS]; fNew[5] = &f0_D[FNE];
      fNew[6] = &f0_D[FNW];fNew[7] = &f0_D[FSW];fNew[8] = &f0_D[FSE];
      fOld[0] = &f1_D[FR]; fOld[1] = &f1_D[FE]; fOld[2] = &f1_D[FN];
      fOld[3] = &f1_D[FW]; fOld[4] = &f1_D[FS]; fOld[5] = &f1_D[FNE];
      fOld[6] = &f1_D[FNW];fOld[7] = &f1_D[FSW];fOld[8] = &f1_D[FSE];
    }

    start[1] = clock();
    // Collision+Propagation calculus for SOLID+FLUID domain
    LBCollProp<<<grid, block>>>(geo_D, fOld[0], fNew[0]);
    CUDA_CHECK(cudaDeviceSynchronize());// Synchronize the device (
       all blocks)
    stop[1] = clock();  start[2] = clock();
//    LBBC_xBackup<<<grid1, block>>>(fNew[0], fB_D);// Backup for
   EAST/WEST borders in a x-periodic BC
//    CUDA_CHECK(cudaDeviceSynchronize());  // Synchronize the device
    (all blocks)
//    CUDA_CHECK(cudaMemcpy(fOld[0], fNew[0], sizeof(double)*NP*NV,
   cudaMemcpyDeviceToDevice));
    stop[2] = clock();  start[3] = clock();
    // Propagation correction for the whole domain
    LBExchange<<<grid1, block>>>(NX/N_THREADS, fNew[0]);
    CUDA_CHECK(cudaDeviceSynchronize());  // Synchronize the device (
       all blocks)
    stop[3] = clock();  start[4] = clock();
//    LBBC_xPeriodic<<<grid1, block>>>(fOld[0], fNew[0], fB_D);//
   Impose x-periodic BC
    stop[4] = clock();
```

```
    LBBC_ZouHeNorth<<<grid1, block>>>(fNew[0], ux_lid, uy_lid);
    CUDA_CHECK(cudaDeviceSynchronize());
    CUDA_CHECK(cudaMemcpy(f1_H, fNew[0], sizeof(double)*NP*NV,
        cudaMemcpyDeviceToHost));

    // Macroscopic variables calculation
    macroscopic(geo_H, f1_H, rho_H, vx_H, vy_H);

    //Code for printing TXT files
    if (step % DEL_TXT == 0) {
      txt(step, geo_H, rho_H, vx_H, vy_H);
    }
    if (step % DEL_VTK == 0) {
      vtk(step, geo_H, rho_H, vx_H, vy_H);
    }
    if (step % DEL_ERR == 0 || step == MAX_STEPS) {
      printf("\n__________ Step %d __________", step);
      relError = 0.0;
      v_sum = 0.0;
      rmse = 0.0;
      // Convergence test - Root Mean Square Error (RMSE)
      for (int i = INI; i < END; i++) { // Calculates error for FLUID
          nodes only
        if (geo_H[i]) { // Fluid node
          v_sum+= abs(vx_H[i]);
          rmse += (vx_H[i] - vx_o_H[i]) * (vx_H[i] - vx_o_H[i]);
          relError += abs(vx_H[i] - vx_o_H[i]);
//              MAX(abs((vx_H[i] - vx_o_H[i])/vx_H[i]), relError);
        }
      }
      // Normalized Root Mean Square Error (RMSE) calculation
      rmse_ = NF * rmse;
      rmse_ = sqrt(rmse_);
      rmse_ /= v_sum;
      // RMS Error
      rmse = sqrt(rmse /= NF);

      // Relative Error
      relError /= v_sum;

      // Store both Errors into the arrays
      insertArray(&error, rmse);  insertArray(&error_, rmse_);

      printf("\nREL_E = %.12f\n", relError);// Print maximum relative
          error
      printf("\nRMSE_ = %.12f\n", rmse_);    // Print Norm-RMSE

      // Convergence Criterion: Normalized RMSE
      if (relError < MAX_ERROR || step == MAX_STEPS) { // rmse_
        stop[0] = clock(); // Start timer
        converged = true; //vtk(step, geo_H, rho_H, vx_H, vy_H);
```

```
            txt(step, geo_H, rho_H, vx_H, vy_H);
            vtk(step, geo_H, rho_H, vx_H, vy_H);
            simulationOutputs(step, grid, grid1, block);
            errorFile(&error, &error_);      // Write Error files
            timeKernel(start, stop, t_sec);
          }
        }
        // Update previous Velocity Field
        memcpy(vx_o_H, vx_H, sizeof(double)*NP);
        memcpy(vy_o_H, vy_H, sizeof(double)*NP);
        step++; // advance iteration
      }

      // Free DEVICE Memory
      CUDA_CHECK(cudaFree(f0_D)); CUDA_CHECK(cudaFree(f1_D));
      CUDA_CHECK(cudaFree(geo_D)); CUDA_CHECK(cudaFree(rho_D));
      CUDA_CHECK(cudaFree(vx_D)); CUDA_CHECK(cudaFree(vy_D));
      cudaDeviceReset();// Causes all profile data to be flushed before
          the application exits

      // Free HOST Memory
      for (int i = 0; i < NV; i++) { fOld[i] = fNew[i] = NULL;}
      free(geo_H); free(rho_H);
      free(vx_H); free(vy_H);
      free(f0_H); free(f1_H);
      freeArray(&error); freeArray(&error_);

      printf("\nReynolds: %f", RE);
      printf("\nKinematic viscosity: %f", (2.*TAU-1.)/6.);
      printf("\nAverage velocity: %f", RE*(2.*TAU-1.)/6./(NY-2.));
      printf("\nBody acceleration: %.20f", (double)GX);
      printf("\nNumber of time steps: %d", step);
      printf("\n\n========== END OF SIMULATION ==========\n");
      return EXIT_SUCCESS;
    }
    __global__ void LBCollProp(int* geo_D, double* f0_D, double* f1_D) {
      // number of threads
      int num_threads = blockDim.x;
      // local thread index
      int tx = threadIdx.x;
      // Block index in x
      int bx = blockIdx.x;
      // Block index in y
      int by = blockIdx.y;
      // Global x-IndexcudaPrintfDisplay
      int xStart = tx + bx * num_threads;
      // Global y-Indfw_Dex
      int yStart = by + YP;
      // Index k in 1D-arrays inside global memory
      int k = NX * yStart + xStart;
```

```
// Shared memmory for propagations within blocks
__shared__ double F_OUT_E[N_THREADS];
__shared__ double F_OUT_W[N_THREADS];
__shared__ double F_OUT_NE[N_THREADS];
__shared__ double F_OUT_NW[N_THREADS];
__shared__ double F_OUT_SW[N_THREADS];
__shared__ double F_OUT_SE[N_THREADS];

//Load fr,fe,fn,fw,fs,fne,fnw,fsw,fse residing in global memory to
    local 'F' variables
double F_IN_R  = f0_D[FR + k];
double F_IN_E  = f0_D[FE + k];
double F_IN_N  = f0_D[FN + k];
double F_IN_W  = f0_D[FW + k];
double F_IN_S  = f0_D[FS + k];
double F_IN_NE = f0_D[FNE + k];
double F_IN_NW = f0_D[FNW + k];
double F_IN_SW = f0_D[FSW + k];
double F_IN_SE = f0_D[FSE + k];

double ux = 0.0, uy = 0.0, dens = 0.0;
// Collision Step
if (geo_D[k] == FLUID) { // Fluid Node

  // First and second moments of the distribution function
  dens += (F_IN_E) + (F_IN_N) + (F_IN_W) + (F_IN_S) + (F_IN_NE)
      + (F_IN_NW) + (F_IN_SW) + (F_IN_SE) + F_IN_R;
  ux += 1 * (F_IN_E) + 0 * (F_IN_N) + -1 * (F_IN_W) + 0 * (F_IN_S)
      + 1 * (F_IN_NE) + -1 * (F_IN_NW) + -1 * (F_IN_SW)
      + 1 * (F_IN_SE);
  uy += 0 * (F_IN_E) + 1 * (F_IN_N) + 0 * (F_IN_W) + -1 * (F_IN_S)
      + 1 * (F_IN_NE) + 1 * (F_IN_NW) + -1 * (F_IN_SW)
      + -1 * (F_IN_SE);

  // Body acceleration application
  ux += a_D[0] * (TAU); uy += a_D[1] * (TAU);

  // Macroscopic Velocities in X and Y directions
  ux /= dens; uy /= dens;

  // Auxiliary variables
  double uSQ_x =   ux * ux, uSQ_y =   uy * uy,
      uEQ_ne =   ux + uy, uEQ_nw = - ux + uy,
      uEQ_sw = - ux - uy, uEQ_se =   ux - uy,
      C = 1.5 * (uSQ_x + uSQ_y);

  // Equilibrium Distribution Function Calculus
  double fEQr = dens * w_D[0] * (1. + -C);
  double fEQe = dens * w_D[1] * (1. + 3. * ux + 4.5 * uSQ_x - C);
  double fEQn = dens * w_D[1] * (1. + 3. * uy + 4.5 * uSQ_y - C);
  double fEQw = dens * w_D[1] * (1. + 3. * -ux + 4.5 * uSQ_x - C);
```

```
    double fEQs = dens * w_D[1] * (1. + 3. * -uy + 4.5 * uSQ_y - C);
    double fEQne = dens * w_D[2]* (1. + 3. * uEQ_ne + 4.5 * uEQ_ne *
       uEQ_ne - C);
    double fEQnw = dens * w_D[2]* (1. + 3. * uEQ_nw + 4.5 * uEQ_nw *
       uEQ_nw - C);
    double fEQsw = dens * w_D[2]* (1. + 3. * uEQ_sw + 4.5 * uEQ_sw *
       uEQ_sw - C);
    double fEQse = dens * w_D[2]* (1. + 3. * uEQ_se + 4.5 * uEQ_se *
       uEQ_se - C);

    // BGK Collision
    F_IN_R  += (fEQr - F_IN_R) / TAU;
    F_IN_E  += (fEQe - F_IN_E) / TAU;
    F_IN_N  += (fEQn - F_IN_N) / TAU;
    F_IN_W  += (fEQw - F_IN_W) / TAU;
    F_IN_S  += (fEQs - F_IN_S) / TAU;
    F_IN_NE += (fEQne - F_IN_NE) / TAU;
    F_IN_NW += (fEQnw - F_IN_NW) / TAU;
    F_IN_SW += (fEQsw - F_IN_SW) / TAU;
    F_IN_SE += (fEQse - F_IN_SE) / TAU;
  }
  else if (geo_D[k] == SOLID) { // Solid Node
    //BOUNCE BACK:
    double temp;
    temp = F_IN_E;  F_IN_E = F_IN_W;   F_IN_W = temp;
    temp = F_IN_N;  F_IN_N = F_IN_S;   F_IN_S = temp;
    temp = F_IN_NE; F_IN_NE = F_IN_SW; F_IN_SW = temp;
    temp = F_IN_NW; F_IN_NW = F_IN_SE; F_IN_SE = temp;
  }
//   Partial Propagation Step
  // Propagation using shared memory for distributions
  // having a shift in east or west direction
  if (tx == 0) {
    //store distribution leaving
    //the domain across the east border
    F_OUT_E[tx + 1] = F_IN_E;
    F_OUT_NE[tx + 1] = F_IN_NE;
    F_OUT_SE[tx + 1] = F_IN_SE;
    //store distribution leaving
    //the domain across the west border
    F_OUT_W[num_threads - 1] = F_IN_W;
    F_OUT_NW[num_threads - 1] = F_IN_NW;
    F_OUT_SW[num_threads - 1] = F_IN_SW;
  } else if (tx == num_threads - 1) {
    //store distribution leaving
    //the domain across the east border
    F_OUT_E[0] = F_IN_E;
    F_OUT_NE[0] = F_IN_NE;
    F_OUT_SE[0] = F_IN_SE;
    F_OUT_W[tx - 1] = F_IN_W;
    F_OUT_NW[tx - 1] = F_IN_NW;
```

```
    F_OUT_SW[tx - 1] = F_IN_SW;
  } else {
    //store distribution leaving
    //the domain across the west border
    F_OUT_E[tx + 1] = F_IN_E;
    F_OUT_NE[tx + 1] = F_IN_NE;
    F_OUT_SE[tx + 1] = F_IN_SE;
    F_OUT_W[tx - 1] = F_IN_W;
    F_OUT_NW[tx - 1] = F_IN_NW;
    F_OUT_SW[tx - 1] = F_IN_SW;
  }
  // Synchronize threads
  __syncthreads();
  // Device memory writing
  f1_D[FR + k] = F_IN_R;
  f1_D[FE + k] = F_OUT_E[tx];
  f1_D[FW + k] = F_OUT_W[tx];
  // North Propagation
  k = NX * (yStart + 1) + xStart;
  f1_D[FN + k] = F_IN_N;
  f1_D[FNE + k] = F_OUT_NE[tx];
  f1_D[FNW + k] = F_OUT_NW[tx];
  // South Propagation
  k = NX * (yStart - 1) + xStart;
  f1_D[FS + k] = F_IN_S;
  f1_D[FSW + k] = F_OUT_SW[tx];
  f1_D[FSE + k] = F_OUT_SE[tx];
}
__global__ void LBExchange(int nbx, double *f1_D) {
  // Number of threads per block in LBCollProp
  int num_threads = N_THREADS;
  // Number of threads per block in LBExchange - one per line
  int num_threads1 = blockDim.x;
  // Block index in 'y'
  int by = blockIdx.y;
  // local thread index
  int tx = threadIdx.x;

  int xStart, yStart, bx; // geometric position (xStart ,yStart) and
     number of blocks in 'x' in LBCollProp
  int xStartW, xTargetW;  // Startpoint & Targetpoint for W
     distribution function
  int xStartE, xTargetE;  // Startpoint & Targetpoint for E
     distribution function
  int kStartW, kTargetW;  // Global index for (xStartW) and (kTargetW
     ) in W distribution functions
  int kStartE, kTargetE;  // Global index for (xStartE) and (kTargetE
     ) in W distribution functions

  yStart = by * num_threads1 + YP + tx; // vertical coordinate of
     current thread inside the grid
```

```
  // Global memory transfer - WEST Distribution functions
  for (bx = 0; bx < nbx - 1; bx++) {
    xStart = bx * num_threads;       // beginning of current block
    // Set Startpoint
    xStartW = xStart + 2*num_threads - 1; // end of current block
    // Set Targetpoint
    xTargetW = xStartW - num_threads;// end of the previous block (on
        the left side)
    // Set global memory indexes
    kStartW = NX * yStart + xStartW;// global memory index in
       position (xStartW ,yStart)
    kTargetW = NX * yStart + xTargetW;// global memory index in
       position (xTargetW,yStart)
    // Global Memory Copy: Startpoint -> Targetpoint
    f1_D[FW  + kTargetW] = f1_D[FW  + kStartW];    // west
       distribution function
    f1_D[FNW + kTargetW] = f1_D[FNW + kStartW];    // northwest
       distribution function
    f1_D[FSW + kTargetW] = f1_D[FSW + kStartW]; // southwest
       distribution function
  }
  // Global memory transfer - EAST Distribution functions
  for (bx = nbx - 2; bx >= 0; bx--) {
    xStart = bx * num_threads;       // beginning of current block
    // Set Startpoint
    xStartE = xStart;          // beginning of current block
    // Set Targetpoint
    xTargetE = xStartE + num_threads;// beginning of the previous
       block (on the right side)
    // Set global memory indexes
    kStartE = NX * yStart + xStartE;// global memory index in
       position (xStartE ,yStart)
    kTargetE = NX * yStart + xTargetE;// global memory index in
       position (xTargetE,yStart)
    // Global Memory Copy: Startpoint -> Targetpoint
    f1_D[FE  + kTargetE] = f1_D[FE  + kStartE]; // east distribution
       function
    f1_D[FNE + kTargetE] = f1_D[FNE + kStartE]; // northeast
       distribution function
    f1_D[FSE + kTargetE] = f1_D[FSE + kStartE]; // southeast
       distribution function
  }
}
__global__ void LBBC_xBackup(double *f1_D, double *fB) {
  int num_threads = N_THREADS;
  // Number of threads per block in LBExchange - one per line
  int num_threads1 = blockDim.x;
  // Block index in 'y'
  int by = blockIdx.y;
  // local thread index
```

```
  int tx = threadIdx.x;

  int yStart = by * num_threads1 + YP + tx;

  int xStartE = NX - 1 - num_threads;      // F's Leste na borda oeste
      -> bloco seguinte
  int xStartW = num_threads - 1; // F's Oeste na borda leste -> bloco
      anterior
  int kStartE = NX * yStart + xStartE;
  int kStartW = NX * yStart + xStartW;
  int idx = by * num_threads1 + tx;

  double F_IN_E  = f1_D[FE  + kStartE];// pointer to fe=[0*NY]
  double F_IN_W  = f1_D[FW  + kStartW];// pointer to fw=[1*NY]
  double F_IN_NE = f1_D[FNE + kStartE];// pointer to fne=[2*NY]
  double F_IN_NW = f1_D[FNW + kStartW];// pointer to fnw=[3*NY]
  double F_IN_SW = f1_D[FSW + kStartW];// pointer to fsw=[4*NY]
  double F_IN_SE = f1_D[FSE + kStartE];// pointer to fse=[5*NY]

  __syncthreads();
  fB[0 * NY + idx] = F_IN_E;
  fB[1 * NY + idx] = F_IN_W;
  fB[2 * NY + idx] = F_IN_NE;
  fB[3 * NY + idx] = F_IN_NW;
  fB[4 * NY + idx] = F_IN_SW;
  fB[5 * NY + idx] = F_IN_SE;
}
__global__ void LBBC_xPeriodic(double *fT_D, double *f1_D, double *fB
    ) {
  // Number of threads per block in LBCollProp
  int num_threads = N_THREADS;
  // Number of threads per block in LBExchange - one per line
  int num_threads1 = blockDim.x;
  // Block index in 'y'
  int by = blockIdx.y;
  // local thread index
  int tx = threadIdx.x;

  int yStart = by * num_threads1 + YP + tx;

  int xTargetE = 0;
  int xTargetW = NX - 1;

  int kTargetE = NX * (yStart) + xTargetE;
  int kTargetW = NX * (yStart) + xTargetW;

  int idx = by * num_threads1 + tx;
  double F_IN_E  = fB[0 * NY + idx];
  double F_IN_W  = fB[1 * NY + idx];
  double F_IN_NE = fB[2 * NY + idx];
  double F_IN_NW = fB[3 * NY + idx];
```

```
  double F_IN_SW = fB[4 * NY + idx];
  double F_IN_SE = fB[5 * NY + idx];

  __syncthreads();
// Boundary Condition (BC) Implementation
  f1_D[FE  + kTargetE] = F_IN_E;
  f1_D[FW  + kTargetW] = F_IN_W;
  f1_D[FNE + kTargetE] = F_IN_NE;
  f1_D[FNW + kTargetW] = F_IN_NW;
  f1_D[FSW + kTargetW] = F_IN_SW;
  f1_D[FSE + kTargetE] = F_IN_SE;
}
__global__ void LBBC_ZouHeNorth(double* f1_D, double ux0, double uy0)
   {
  // Number of threads per block in LBExchange - one per line
  int num_threads1 = blockDim.x;
  // Block index in 'y'
  int by = blockIdx.y;
  // local thread index
  int tx = threadIdx.x;

  int yStart = by * num_threads1 + YP + tx;

  if (yStart - YP == NY-1){ // North border
    for (int i = 0; i < NX; i++){
      int k = NX*yStart + i;

      double fR  = f1_D[FR+k] , fE = f1_D[FE+k] , fN = f1_D[FN+k] ,
             fW  = f1_D[FW+k] , fNE = f1_D[FNE+k],fNW = f1_D[FNW+k];

      double rho0 = (fR + fE + fW + 2.*(fN + fNE + fNW))/(1.+uy0);
      double rux0 = rho0*ux0, ruy0 = rho0*uy0;

      f1_D[FS+k]  = fN - (2./3.)*ruy0;
      f1_D[FSW+k] = fNW + .5*(fE-fW) - .5*rux0 - (1./6.)*ruy0;
      f1_D[FSE+k] = fNE + .5*(fW-fE) + .5*rux0 - (1./6.)*ruy0;
    }
  }
}
__host__ void f_eq(int *geo, const double a[], double *vx, double *vy
   ,
    double *rho, const double w[], double *f) {
  for (int i = INI; i < END; i++) {
    double ux = vx[i];
    double uy = vy[i];

    // Squared velocities
    double uSQ_x = ux * ux, uSQ_y = uy * uy; //uSQ = ux * ux + uy *
       uy;

    double uEQ_ne = ux + uy, uEQ_nw = -ux + uy, uEQ_sw = -ux + -uy,
```

```
        uEQ_se =
         ux + -uy;

    double C = 1.5 * (uSQ_x + uSQ_y);

    // Equilibrium Distribution Function Calculus
    f[i + FR ] = rho[i] * w[0]* (1. + -C);
    f[i + FE ] = rho[i] * w[1]* (1. + 3. * ux + 4.5 * uSQ_x - C);
    f[i + FN ] = rho[i] * w[1]* (1. + 3. * uy + 4.5 * uSQ_y - C);
    f[i + FW ] = rho[i] * w[1]* (1. + 3. * -ux + 4.5 * uSQ_x - C);
    f[i + FS ] = rho[i] * w[1]* (1. + 3. * -uy + 4.5 * uSQ_y - C);
    f[i + FNE] = rho[i] * w[2]* (1. + 3. * uEQ_ne + 4.5 * uEQ_ne *
       uEQ_ne - C);
    f[i + FNW] = rho[i] * w[2]* (1. + 3. * uEQ_nw + 4.5 * uEQ_nw *
       uEQ_nw - C);
    f[i + FSW] = rho[i] * w[2]* (1. + 3. * uEQ_sw + 4.5 * uEQ_sw *
       uEQ_sw - C);
    f[i + FSE] = rho[i] * w[2]* (1. + 3. * uEQ_se + 4.5 * uEQ_se *
       uEQ_se - C);
  }
}
__host__ void setCudaMaxDevice(void) {
  int num_devices, device;
  cudaGetDeviceCount(&num_devices);
  if (num_devices > 1) {
    int max_multiprocessors = 0, max_device = 0;
    for (device = 0; device < num_devices; device++) {
      cudaDeviceProp properties;
      cudaGetDeviceProperties(&properties, device);
      if (max_multiprocessors < properties.multiProcessorCount) {
        max_multiprocessors = properties.multiProcessorCount;
        max_device = device;
      }
    }
    cudaSetDevice(max_device);
  }
}
__host__ void store_geo(int* geo,
    char* filename/*, unsigned *nx, unsigned *ny, unsigned *nz,
       unsigned *np*/) {
  int temp[3];
  printf("Filename: %s\n", filename);
  FILE *media;

  if ((media = fopen(filename, "r")) == NULL) {
    //fprintf(stderr, "\n# Failure opening: %s!", filename);
    perror("Error opening the chosen file!\n");
    exit(1);
  } else {
    fseek(media, 0, 0);
    fscanf(media, "%d %d %d", &temp[0], &temp[1], &temp[2]); // NX NY
```

```
        NZ
    if (temp[2] == 1) {
      for (unsigned i = 0; i < NP; i++) {
        fscanf(media, "  %d", &geo[i]);
      }
      fprintf(stdout, "\nMedia file reading has been successful!\n");
      fclose(media);
    } else {
      fprintf(stderr,
          "\nProgram is unable to compute 3D geometry! (NZ != 1)");
      exit(1);
    }
  }
}
__host__ void ini_rho(int *geo, double rho_ini, double *rho) {
  for (int k = 0; k < NP; k++) {
    if (geo[k] == FLUID || geo[k] == SOLID) {
      rho[k] = rho_ini;
    } else {
      rho[k] = 0.0;
    }
  }
}
__host__ void ini_vel_field(int *geo, double *vx, double *vx_prof,
   double *vy,
    double *vy_prof) {
  // Velocity field initilization (based on prof_uniform [vx(y),vy(y)
     ] profile)
  int k;
  for (int j = 0; j < NY; j++) {
    for (int i = 0; i < NX; i++) {
      k = NX * (YP + j) + i;
      if (geo[k] == 1) {
        vx[k] = vx_prof[j];
        vy[k] = vy_prof[j];
      } else {
        vx[k] = 0.0;
        vy[k] = 0.0;
      }
    }
  }
}
__host__ void macroscopic(int *geo, double *f, double *rho, double *
   vx, double *vy) {
  for (int i = INI; i < END; i++) {
    if (geo[i]){
      double dens = 0.0, px = 0.0, py = 0.0;
      dens += f[i + FR] + f[i + FE] + f[i + FN] + f[i + FW] + f[i +
         FS]
          + f[i + FNE] + f[i + FNW] + f[i + FSW] + f[i + FSE];
      // Momentum
```

```
      px += 1 * f[i + FE] + 0 * f[i + FN] + -1 * f[i + FW] + 0 * f[i
         + FS]
          + 1 * f[i + FNE] + -1 * f[i + FNW] + -1 * f[i + FSW]
          + 1 * f[i + FSE];
      py += 0 * f[i + FE] + 1 * f[i + FN] + 0 * f[i + FW] + -1 * f[i
         + FS]
          + 1 * f[i + FNE] + 1 * f[i + FNW] + -1 * f[i + FSW]
          + -1 * f[i + FSE];

      rho[i] = dens;      // Macroscopic Densitiy in a lattice node
      vx[i] = px / dens; // Macroscopic Velocity in x direction in a
         lattice node
      vy[i] = py / dens; // Macroscopic Velocity in y direction in a
         lattice node
    } else {
      rho[i]= 0.0;
      vx[i] = 0.0;
      vy[i] = 0.0;
    }
  }
}
__host__ void vtk(int step, int *geo_H, double *rho, double *vx,
   double *vy) { // Initializes macroscopic properties for each mesh
   node
  // Density VTK File
  char name_rho[28];
  sprintf(name_rho, "dens_%04d.vtk", step);
  FILE *rho_vtk = fopen(name_rho, "w");

  // Velocity VTK File
  char name_vel[28];
  sprintf(name_vel, "vel_%04d.vtk", step);
  FILE *vel_vtk = fopen(name_vel, "w");

  if (vel_vtk == NULL || rho_vtk == NULL) {
    printf("\n#Failure opening vel.vtk or rho.vtk!"); exit(1);
  } else {
    // PRINT Density VTK file
    fprintf(rho_vtk, "# vtk DataFile Version 2.0\n");
    fprintf(rho_vtk, "Density\n");
    fprintf(rho_vtk, "ASCII\n");
    fprintf(rho_vtk, "DATASET STRUCTURED_POINTS\n");
    fprintf(rho_vtk, "DIMENSIONS %d %d %d\n", NX, NY, 1);
    fprintf(rho_vtk, "ASPECT_RATIO 1 1 1\n");
    fprintf(rho_vtk, "ORIGIN 0 0 0\n");
    fprintf(rho_vtk, "POINT_DATA %d\n", N);
    fprintf(rho_vtk, "SCALARS Density double\n");
    fprintf(rho_vtk, "LOOKUP_TABLE default");
    // PRINT Velocity VTK file
    fprintf(vel_vtk, "# vtk DataFile Version 2.0\n");
    fprintf(vel_vtk, "Velocidade\n");
```

```
      fprintf(vel_vtk, "ASCII\n");
      fprintf(vel_vtk, "DATASET STRUCTURED_POINTS\n");
      fprintf(vel_vtk, "DIMENSIONS %d %d %d\n", NX, NY, 1);
      fprintf(vel_vtk, "ASPECT_RATIO 1 1 1\n");
      fprintf(vel_vtk, "ORIGIN 0 0 0\n");
      fprintf(vel_vtk, "POINT_DATA %d\n", N);
      fprintf(vel_vtk, "VECTORS Velocity double");
      for (int i = INI; i < END; i++) {
        if (i % NX == 0) {
          fprintf(vel_vtk, "\n");
          fprintf(rho_vtk, "\n");
        }
        fprintf(vel_vtk, "%.8f %.8f %.8f ", vx[i], vy[i], 0.0); // 2D -
            vz[i]=0.0
        fprintf(rho_vtk, "%.8f ", rho[i]);
      }
    }
    fclose(vel_vtk);
    fclose(rho_vtk);
}
__host__ void txt(int step, int *geo_H, double *rho, double *vx,
    double *vy) { // TXT file generator
    // Density TXT File
    char name_vel[28];
    //sprintf(name_vel, "v-%dx%d-Re.3%f-t%06d-E%f.txt", NX, NY, RE,
        step, MAX_ERROR);
    sprintf(name_vel, "v-t_%04d.txt", step);
    FILE *vel_txt;

    // Velocity TXT File
    char name_rho[28];
    //sprintf(name_rho, "r-%dx%d-Re.3%f-t%06d-E%f.txt", NX, NY, RE,
        step, MAX_ERROR);
    sprintf(name_rho, "r-t_%04d.txt", step);
    FILE *rho_txt;

    if ((vel_txt = fopen(name_vel, "w")) == NULL
        || (rho_txt = fopen(name_rho, "w")) == NULL) {
      //fprintf(stderr, "\n# Failure opening: %s!", filename);
      printf("\n#Failure opening vel.txt or rho.txt!"); exit(1);
    } else {
      for (int j = 0; j < NY; j++) {
        for (int i = 0; i < NX; i++) {
          int k = NX * (YP + j) + i;
          fprintf(vel_txt, "%.20f %.20f  ", vx[k], vy[k]);
          fprintf(rho_txt, "%.20f  ", rho[k]);
        }
        fprintf(vel_txt, "\n");
    }
    fclose(vel_txt);
    }
```

```
}
__host__ void errorFile(Array *rmse, Array *rmse_) { // ERROR file
    generator
  char name_f[28];
  sprintf(name_f, "errors.txt");
  FILE *error = fopen(name_f, "w");

  if (error == NULL) {
    printf("Failure opening error file for writing!");  exit(1);
  } else {
    for (int i = 0; i < rmse->used; i++) {
      fprintf(error, "%.20f %.20f\n", rmse->array[i], rmse_->array[i
          ]);
    }
  }
  fclose(error);
}
__host__ void simulationOutputs(int step, dim3 grid, dim3 grid1, dim3
    block){
  char name_f[40];
  sprintf(name_f, "sim-Re%f-%dx%d.txt", RE, NX, NY);
  FILE *f = fopen(name_f, "w");

  if (f == NULL) {
      printf("Failure opening error file for writing!");  exit(1);
  } else {
    fprintf(f, "Poiseuille Problem\n\n");
    fprintf(f, "Domain: (NX = %d, NY = %d)\n\n", NX, NY);
    fprintf(f, "------- Kernel Configurations -------\n\n");
    fprintf(f, "LBCollProp<<<grid, block>>>():\n");
    fprintf(f, "LBExchange<<<grid1, block>>>():\n");
    fprintf(f, "LBBC_xBackup<<<grid1, block>>>():\n");
    fprintf(f, "LBBC_xPeriodic<<<grid1, block>>>():\n");
    fprintf(f, "\ngrid  = (%03d,%03d,%03d)\n", grid.x, grid.y, grid.z
        );
    fprintf(f, "grid1 = (%03d,%03d,%03d)\n", grid1.x, grid1.y, grid1.
        z);
    fprintf(f, "block = (%03d,%03d,%03d)\n", block.x, block.y, block.
        z);
    fprintf(f, "---------------------------------------\n\n");
    fprintf(f, "Reynold = %f\n", RE);
    fprintf(f, "Kinematic viscosity = %f\n", (2.*TAU-1.)/6.);
    fprintf(f, "Average velocity = %.20f\n", RE*(2.*TAU-1.)/6./(NY
        -2.));
    fprintf(f, "Body acceleration = %.20f\n\n", (double)GX);
    fprintf(f, "Total simulation steps = %d\n", step);
  }
  fclose(f);
}
__host__ void timeKernel(clock_t start[5], clock_t stop[5], double t
    [5]){
```

```
  char name_f[40];
  sprintf(name_f, "times.txt");
  FILE *f = fopen(name_f, "w");

  for(int i = 0; i < 5; i++)
    t[i] = (double)(stop[i]-start[i])/CLOCKS_PER_SEC;

  if (f == NULL) {
      printf("Failure opening time file for writing!"); exit(1);
  } else {
    fprintf(f, "Poiseuille Problem\n\n");
    fprintf(f, "Domain: (NX = %d, NY = %d)\n\n", NX, NY);
    fprintf(f, "-------- Kernel Execution Time ----------------\n\n");
    fprintf(f, "LBCollProp<<<grid, block>>>():      %f\n", t[1]);
    fprintf(f, "LBExchange<<<grid1, block>>>():     %f\n", t[2]);
    fprintf(f, "LBBC_xBackup<<<grid1, block>>>():   %f\n", t[3]);
    fprintf(f, "LBBC_xPeriodic<<<grid1, block>>>(): %f\n", t[4]);
    fprintf(f, "-----------------------------------------------\n\n");
    fprintf(f, "Total processing time:              %f\n", t[0]);
  }
  fclose(f);
}
__host__ int countFluid(int *geo) {
  int counter = 0;
  for (int i = INI; i < END; i++)
    if (geo[i])
      counter++;
  return counter;
}
__host__ void initArray(Array *a, size_t initialSize) {
  a->array = (double *) malloc(initialSize * sizeof(double));
  a->used = 0;
  a->size = initialSize;
}
__host__ void insertArray(Array *a, double element) {
  if (a->used == a->size) {
    a->size *= 2;
    a->array = (double *) realloc(a->array, a->size * sizeof(double))
       ;
  }
  a->array[a->used++] = element;
}
void freeArray(Array *a) {
  free(a->array);
  a->array = NULL;
  a->used = a->size = 0;
}
```