UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

WESLEY DOS SANTOS MENENGUCI

IMPLEMENTAÇÃO DE MODELOS DE MECÂNICA DOS FLUIDOS COMPUTACIONAL EM SISTEMAS *MANY-CORE* USANDO C+CUDA

VITÓRIA

2011

WESLEY DOS SANTOS MENENGUCI

IMPLEMENTAÇÃO DE MODELOS DE MECÂNICA DOS FLUIDOS COMPUTACIONAL EM SISTEMAS *MANY-CORE* USANDO C+CUDA

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do grau de Mestre em Informática.

Orientadores:

Prof^{a}. Dr^{a}. Lucia Catabriga

Prof^{a}. Dr^{a}. Andréa Maria Pedrosa Valli

VITÓRIA

2011

WESLEY DOS SANTOS MENENGUCI

IMPLEMENTAÇÃO DE MODELOS DE MECÂNICA DOS FLUIDOS COMPUTACIONAL EM SISTEMAS *MANY-CORE* USANDO C+CUDA

Dissertação apresentada ao Programa de Pós-Graduação em Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do grau de Mestre em Informática.

Aprovada em 25 de agosto de 2011.

COMISSÃO EXAMINADORA

Prof^{a}. Dr^{a}. Lucia Catabriga
Universidade Federal do Espírito Santo
Orientadora

Prof^{a}. Dr^{a}. Andréa Maria Pedrosa Valli
Universidade Federal do Espírito Santo
Orientadora

Prof. Dr. Alberto Ferreira De Souza
Universidade Federal do Espírito Santo

Prof. Dr. Alvaro Luiz Gayoso de Azeredo Coutinho
Universidade Federal do Rio de Janeiro

ficha catalográfica

Aos meus pais João e Deni e minha namorada Cyntia.

Agradecimentos

Agradeço primeiramente a Deus, feitor e consumador da vida, pelo sustento durante esta caminhada.

Às minhas orientadoras: Lúcia Catabriga, pela oportunidade e confiança em mim depositada e Andrea M. P. Valli, pelo primeiro apoio e pelo auxílio, mesmo que distante.

Aos meu colegas de mestrado, especialmente a Kamila Ghidetti e ao Lucas Scotta, e do LCAD, em especial ao Lucas Veronese, ao Rafael Rodrigues e ao Jorcy Neto, pelos vários momentos de ajuda.

Aos meus pais e meus irmãos, que nunca mediram esforços para me ajudar nesta difícil tarefa.

À minha namorada Cyntia, pelo apoio, compreensão e companheirismo, principalmente nos momentos de *stress*.

Ao Fundo de Apoio à Ciência e Tecnologia do Município de Vitória (FACITEC), pela bolsa de mestrado concedida, a qual me deu tranquilidade para a realização deste trabalho.

Enfim, agradeço a todos que de forma direta ou indireta contribuiram para que este trabalho fosse realizado.

Mera mudança não é crescimento. Crescimento é a síntese de mudança e continuidade, e onde não há continuidade não há crescimento.

C. S. Lewis

Resumo

As unidades de processamento gráfico (*Graphics Processing Unit* – GPU) surgiram como um poderoso dispositivo computacional e a plataforma *Compute Unified Device Architecture* (CUDA) como um ambiente adequado para a implementação de código em GPU. Especializada inicialmente em processamento gráfico, atualmente as GPU vem sendo empregadas na otimização de cálculos lógicos e aritméticos voltados para as mais diversas áreas, beneficiando muitas áreas de pesquisa com a redução do tempo de computação. O objetivo deste trabalho é mostrar como aplicações em mecânica dos fluidos, discretizadas pelo método das diferenças finitas, podem se beneficiar com a utilização desta tecnologia. Implementações paralelas em C+CUDA para GPU das equações de Navier-Stokes e de transporte são comparadas com uma versão sequencial para CPU implementada em C. É utilizada uma formulação em diferenças finitas implícita-explícita, sendo o algoritmo explícito nas velocidades e temperatura e implícito na pressão. A resolução dos sistemas lineares resultantes é feita utilizando um esquema de coloração Red-Black das células internas da malha e o método iterativo *successive-over-relaxation* (SOR), denominado Red-Black-SOR. São discutidos neste trabalho os impactos do uso de tipos de dados *double* e *float* e também a utilização das memórias *shared* e global das GPU empregadas. O algoritmo C+CUDA é examinado para o seguinte conjunto de problemas conhecidos da literatura: cavidade com cobertura deslizante, escoamento sobre um degrau, escoamento laminar com um obstáculo cilíndrico, convecção natural e convecção de Rayleigh-Bénard, considerando casos bidimensionais e tridimensionais. O tempo de processamento é comparado com o mesmo algoritmo implementado em C. Os resultados mostraram que é possível alcançar speedups da ordem de 25 vezes para dados *float* e 21 vezes para dados *double* utilizando C+CUDA.

Palavras-chave: Navier-Stokes, Equações de. Teoria do Transporte. Diferenças Finitas. C+CUDA (Linguagem de programação de computador).

Abstract

The Graphics Processing Unit (GPU) has emerged as a powerfull computing device and the Compute Unified Device Architecture (CUDA) platform is a suitable environment to develop application software that can run on GPUs. Initially specialized in graphics processing, the GPU has been designed to optimize the logical and arithmetical calculations benefiting several research areas by reducing the computation time. This work shows how applications in fluid mechanics, discretized by the finite difference method, can perform well with this technology. Parallel implementations of the Navier-Stokes and transport equations in C+CUDA for GPU are compared with sequential versions for CPU implemented in C. A finite difference formulation is considered and the algorithm is characterized as being explicit in the velocities and temperature, and implicit in the pressure. The resulting linear systems are solved using a Red-Black coloring scheme with the successive over-relaxation (SOR) iterative solver, called Red-Black-SOR. The performance of the parallel codes are discussed using double and float data types and also shared and global memories. The parallel algorithms are verified for the following set of problems: lid-driven cavity, backward-facing step, flow past a cylindrical obstacle, natural convection with heated lateral walls and Rayleigh-Bénard convection, considering two and three-dimensional cases. The time processing is compared with the algorithm implemented in C. Numerical results showed that we can achieve speedups around 25× using float data and 21× using double data.

Key words: Navier-Stokes equations. Transport theory. Finite Difference. C+CUDA.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

Avanços científicos e inovações em *hardware* e *software* permitiram um aumento exponencial no desempenho dos sistemas computacionais nos últimos 40 anos. Em grande parte, esta melhoria no desempenho foi propiciada pelo que se convencionou denominar de "Lei de Moore", (MOORE, 1965), ou seja, a capacidade da indústria de dobrar a cada dois anos (aproximadamente) o número de dispositivos (transistores) que podem ser colocados em um circuito integrado (CI), (HENNESSY; PATTERSON, 2006).

Até recentemente, a tendência na indústria de processadores era empregar transistores adicionais na implementação de CIs contendo sistemas (processador, seus *caches*, etc) com um único processador cada vez mais poderoso. No entanto, três obstáculos se tornaram aparentemente grandes demais para a continuidade desta tendência: (*i*) o consumo de energia e a consequente necessidade da dissipação do calor proveniente do chaveamento em alta frequência de um número cada vez maior de transistores (*the Power Wall*), (ASANOVIC *et al.*, 2006; MANFERDELLI, 2007; MCCALPIN *et al.*, 2007); (*ii*) a crescente latência da hierarquia de memória (*the Memory Wall*), (WULF; MCKEE, 1995; FERNANDES *et al.*, 2002); e (*iii*) as dificuldades associadas a uma maior exploração do paralelismo no nível de instrução (*the ILP Wall*), (ASANOVIC *et al.*, 2006; IRWIN; SHENA, 2005). David Patterson sumarizou estes três problemas na expressão: *the Power Wall + the Memory Wall + the ILP Wall = the Brick Wall for serial performance*, (IRWIN; SHENA, 2005).

Em face dos obstáculos mencionados, a indústria mundial de computadores mudou de curso em 2005 com o desenvolvimento de sistemas com múltiplos núcleos de processamento (*multi-core*). Transistores adicionais disponibilizados por avanços tecnológicos na fabricação de CIs passaram a ser empregados na implementação de processadores adicionais, ao invés de em um único processador mais poderoso como era feito anteriormente, (TENDLER *et al.*, 2002; KONGETIRA *et al.*, 2005).

Atualmente, processadores *multi-core* tiram proveito do número de transistores disponível em um único CI para disponibilizar *hardware* próprio para a exploração do parale-

lismo de grão grosso existente nas aplicações, (DE SOUZA, 2008). Contudo, a indústria de processadores caminha rapidamente para sistemas *many-core* (MANFERDELLI, 2007) com dezenas ou centenas de núcleos de processamento. Muito embora sistemas com múltiplos processadores estejam entre nós desde a década de 1960 e tenham sido muito estudados nas décadas de 1980 e 1990, mecanismos eficientes de programação de sistemas *many-core*, que tirem proveito do paralelismo de grão grosso e de grão fino das aplicações, até recentemente não existiam.

Neste contexto surge a *Compute Unified Device Architecture* (CUDA), (NICKOLLS *et al.*, 2008), uma nova arquitetura paralela exposta ao programador por meio de uma pequena extensão da linguagem de programação C. CUDA foi desenvolvida dentro do escopo da indústria de processadores gráficos (*Graphics Processing Unit* - GPU), (NVIDIA, 2008).

Os mecanismos de programação de GPUs disponíveis até recentemente eram por demasiado voltados para facilitar o uso de GPUs para processamento gráfico. CUDA mudou este cenário por meio da disponibilização de um modelo de programação massivamente paralela que pode facilmente ser integrado ao modelo sequencial de programação vigente - programas sequenciais em C traduzidos para C+CUDA têm alcançado desempenhos em GPUs centenas de vezes superiores aos alcançados em CPUs, (DE SOUZA, 2008).

Nas últimas décadas as técnicas de Mecânica dos Fluidos Computacional (MFC) – área da computação científica que estuda métodos computacionais para a simulação de fenômenos que envolvem escoamentos de fluidos com ou sem troca de calor – tem desenvolvido ferramentas muito importantes no que tange as áreas de projeto, otimização, pesquisa e desenvolvimento de aplicações industriais e científicas. A simulação computacional de escoamentos de fluidos requer (SAABAS, 1991): (*i*) um modelo matemático que represente o problema físico de interesse; (*ii*) um método numérico para a solução do modelo matemático em domínios complexos; e, (*iii*) uma implementação computacional do método numérico. As vantagens do método numérico são, em geral, o seu baixo custo quando comparado com o método experimental, o acesso a informações detalhadas sobre a solução e a possibilidade de realizar simulações de casos em condições ideais e reais.

Basicamente, o usuário da MFC está interessado em obter as distribuições de velocidades, pressões e temperaturas na região do escoamento para um melhor entendimento dos processos físicos envolvidos. Em geral, com essas informações é possível otimizar o projeto, reduzir seus custos operacionais e melhorar o desempenho do item projetado. Por exemplo, a redução no arrasto aerodinâmico de uma aeronave permite reduzir o seu

consumo de combustível.

Nos últimos anos a simulação numérica teve um grande desenvolvimento, tornando-se uma poderosa ferramenta na resolução de problemas em engenharia (MALISKA, 1995). O uso de métodos numéricos de forma alguma implica que a mecânica dos fluidos experimental e as análises teóricas estejam sendo postas de lado. É muito comum as três técnicas se complementarem durante um projeto que envolva escoamento de fluidos e no estudo de modelos teóricos para algum fenômeno particular.

Apesar de todo o desenvolvimento de arquiteturas de processadores e do crescente aumento do poder computacional das máquinas atuais, existem aplicações práticas que transcendem a capacidade computacional atualmente instalada nos maiores centros de processamento no mundo. Problemas como a previsão do clima em escala global, escoamentos em reservatórios de petróleo em escala real a simulação completa das estruturas turbulentas em escapamentos são exemplos de aplicações que ainda representam desafios para a comunidade científica.

Dentre os diversos trabalhos encontrados na literatura que apresentam soluções com o uso de GPU para o método das diferenças finitas pode ser destacado o estudo desenvolvido por (SHINN; VANKA, 2009). Neste trabalho é apresentado a resolução de problemas em fluidos turbulentos tridimensionais utilizando técnicas multigrid para a resolução dos sistemas lineares. As discussões apresentadas mostram que o uso de precisão simples (*float*) não influenciou nos resultados obtidos. Os autores apresentam ainda um estudo sobre a latência da memória global da GPU e a utilização da memória compartilhada (*shared*). Outro trabalho correlato é a dissertação de mestrado de (THIBAULT, 2009) que apresenta o uso de sistemas *multi-core* (múltiplas CPUs) e *many-core* (GPU e múltiplas GPUs) em problemas de dispersão de poluentes em grandes áreas urbanas. Os resultados apresentam ganhos consideráveis com o uso de GPUs, mesmo em relação a arquiteturas multiprocessadas (*dual-core* e *quad-core*). Os resultados com GPU apresentam ganho utilizando ambas as precisões, porém o desempenho é menor quando a precisão dupla (*double*) é utilizada. Alguns objetos do trabalho de Thibault foram importantes para o presente trabalho, como a discretização tridimensional das equações de Navier-Stokes. Para o mesmo problema, também pode-se destacar os trabalhos de (SENOCAK *et al.*, 2009; THIBAULT; SENOCAK, 2009). Destacamos ainda o trabalho (MENENGUCI *et al.*, 2010) que investiga o uso de GPUs para problemas bidimensionais regidos pelas equações de Navier-Stokes e discretizados pelo método das diferenças finitas. Os resultados apresentados neste trabalho mostram que o uso da memória shared possibilita desempenho

discretamente superior, mesmo sendo utilizada em uma pequena porção do algoritmo. Em relação ao uso de multi-GPUs, pode-se destacar os recentes trabalhos de (JACOBSEN *et al.*, 2010; VERONESE *et al.*, 2010), os quais exploraram características do padrão MPI para troca de mensagens e a programação CUDA para sobrepor os dados de transferência da GPU através da MPI para acelerar os cálculos da GPU. Os resultados demonstraram que clusters multi-GPU podem acelerar substancialmente simulações em mecânica dos fluidos computacional.

Este trabalho tem por objetivo mostrar como aplicações em mecânica dos fluidos, discretizadas pelo método das diferenças finitas, podem se beneficiar com o uso de GPUs. São comparadas implementações paralelas em C+CUDA para GPU das equações de Navier-Stokes e de transporte com uma versão sequencial em C para CPU. A resolução dos sistemas lineares resultantes é feita utilizando um esquema de coloração Red-Black para as células internas da malha e o método iterativo *successive-over-relaxation* (SOR), denominado Red-Black-SOR. É analisado o impacto, numérico e no desempenho, da utilização de variáveis com precisão simples (*float*) e precisão dupla (*double*) nas GPUs. Além disso, é discutido a utilização das memórias compartilhada (*shared*) e global existentes na GPU. Os resultados mostraram que é possível alcançar *speedups* da ordem de 25 vezes para dados *float* e 21 vezes para dados *double* utilizando C+CUDA.

Este trabalho consta de mais 4 capítulos além desta introdução, que estão organizados como a seguir. No Capítulo 2 são apresentados o modelos matemáticos para as equações governantes da mecânica dos fluidos bem como o processo de discretização das mesmas pelo método das diferenças finitas. O Capítulo 3 introduz os conceitos do modelo de arquitetura e de programação em CUDA; apresentando alguns trechos do código paralelo implementado. No Capítulo 4 são apresentados e discutidos os resultados obtidos com os experimentos computacionais. Por fim, o Capítulo 5 apresenta as principais conclusões obtidas e as considerações finais do trabalho.

2 Equações Governantes - Método de Aproximação

Este capítulo apresenta um modelo matemático para as equações de Navier-Stokes e a equação do transporte, que modelam o escoamento de fluidos. Os escoamentos apresentados neste trabalho são incompressíveis e laminares e também consideram fenômenos convectivos, como a transferência de calor pelo movimento de um fluido. Também é apresentado uma breve descrição do tratamento numérico adotado, detalhando o método iterativo utilizado na resolução dos sistemas lineares resultante.

2.1 Equações Governantes

O escoamento de um fluido em uma região $\Omega \subset \mathbb{R}^N (N \in \{2, 3\})$ em um intervalo de tempo $t \in [0, t_{end}]$ é caracterizado pelas seguintes grandezas:

$$\begin{array}{rcll} \overrightarrow{\boldsymbol{u}} & : & \Omega \times [0, t_{end}] \rightarrow \mathbb{R}^N & \text{Campo de velocidades} \\ p & : & \Omega \times [0, t_{end}] \rightarrow \mathbb{R} & \text{Pressão} \\ \rho & : & \Omega \times [0, t_{end}] \rightarrow \mathbb{R} & \text{Massa Específica} \end{array}$$

Nesta dissertação o escoamento utilizado é incompressível, sendo assim a massa específica ρ é assumida como constante. O escoamento é descrito por um sistema de equações diferenciais que representam o modelo matemático para os seguintes princípios físicos: conservação da massa, conservação do momento (segunda lei de Newton) e conservação da energia (primeira lei da termodinâmica) (FORTUNA, 2000). Desta forma, o escoamento pode ser representado pelas equações de momento, continuidade e transporte, apresentadas aqui na seguinte forma adimensional:

$$\frac{\partial}{\partial t}\vec{\boldsymbol{u}} + \left(\vec{\boldsymbol{u}} \cdot \nabla\right)\vec{\boldsymbol{u}} + \nabla p = \frac{1}{Re}\nabla^2\vec{\boldsymbol{u}} + \vec{\boldsymbol{f}} \qquad \text{(Momento)} \qquad (2.1)$$

$$\nabla \cdot \vec{\boldsymbol{u}} = 0 \qquad \text{(Continuidade)} \qquad (2.2)$$

$$\frac{\partial T}{\partial t} + \vec{\boldsymbol{u}} \cdot \nabla T = \frac{1}{Re}\frac{1}{Pr}\nabla^2 T + q \qquad \text{(Transporte)} \qquad (2.3)$$

onde: $\vec{\boldsymbol{u}}$ é o vetor de componentes da velocidade, p é a pressão, $Re = (\rho_\infty u_\infty L_\infty)/\mu$ é o número adimensional de Reynolds; ρ_∞, u_∞ e L_∞ são constantes escalares, sendo, respectivamente: massa específica do fluido, velocidade característica e comprimento característico, e μ é a viscosidade dinâmica, T é a temperatura, $Pr = \mu/(\rho\alpha)$ é o número adimensional de Prandtl, sendo α a constante de difusividade térmica, $\vec{\boldsymbol{f}}$ e q são as componentes das forças de corpo.

A representação na forma adimensional é necessária para garantir que as propriedades de um escoamento em um modelo hipotético sejam válidas em um modelo em escala real. Com isso, o conceito de *similaridade* de escoamento é verificado, ou seja, garante-se que as características do modelo e do objeto real são similares, diferindo apenas na escala (FORTUNA, 2000; GRIEBEL, 1998).

Mudanças na temperatura de um fluido fazem com que suas propriedades, como viscosidade e massa específica, variem no espaço e/ou no tempo. A forma exata dessas variações é uma propriedade particular de cada tipo de fluido (FORTUNA, 2000). Ao incluir a temperatura T no modelo matemático deve-se levar em consideração essas propriedades termodinâmicas dos fluidos. Para garantir as propriedades termodinâmicas do fluido utilizamos a *aproximação de Boussinesq*, geralmente considerada para pequenas variações de temperatura (GRAY; GIORGINI, 1976; GRIEBEL, 1998). Esta aproximação adiciona um termo-fonte de *flutuação*, ou de empuxo, nas equações de momento. Toma-se a massa específica do fluido como constante, exceto nos termos de flutuação, e considera-se todas as demais propriedades constantes (GRIEBEL, 1998). Maiores detalhes podem ser encontrados em (BULGARELLI *et al.*, 1984; CASULLI, 1981).

O termo de empuxo pode ser obtido pela expansão da massa específica ρ em série de Taylor ao redor de $\rho_0 = \rho(T_0)$, que é a massa específica para uma temperatura de referência T_0:

$$\rho(T) = \rho_0 + \left.\frac{\partial \rho}{\partial T}\right|_{T_0}(T - T_0) + \Theta(T - T_0)^2 \qquad (2.4)$$

Desprezando os termos de segunda ordem e superiores, tem-se:

$$\rho(T) \approx \rho_0 + \left.\frac{\partial \rho}{\partial T}\right|_{T_0} (T - T_0) \tag{2.5}$$

A Equação (2.5) normalmente é escrita, na forma compacta, como:

$$\rho(T) \approx \rho_0 \left[1 - \beta(T - T_0)\right] \tag{2.6}$$

onde

$$\beta = -\left.\frac{1}{\rho_0}\frac{\partial \rho}{\partial T}\right|_{T_0} \tag{2.7}$$

é o termo de *expansão térmica* do fluido.

A diferença entre as massas específicas de um elemento do fluido e de seu elemento vizinho, na presença de um campo gravitacional, introduz uma força de empuxo sobre o fluido mais leve, isso é, menos denso, expresso pelo termo de empuxo (FORTUNA, 2000):

$$\overrightarrow{\boldsymbol{f}} = \rho \overrightarrow{\boldsymbol{F}}_{\text{temp}} = \rho_0 \left[1 - \beta(T - T_0)\right] \overrightarrow{\boldsymbol{g}} \tag{2.8}$$

onde β é o termo de expansão térmica do fluido e $\overrightarrow{\boldsymbol{g}} = G\overrightarrow{\boldsymbol{e}}_g$, onde G é a constante gravitacional universal e $\overrightarrow{\boldsymbol{e}}_g$ é o vetor unitário na direção da atuação da gravidade. O produto $\rho\overrightarrow{\boldsymbol{F}}_{\text{temp}}$ é um exemplo de força de campo que aparece na equação (2.1) como um termo de fonte ($\overrightarrow{\boldsymbol{f}}$). Desta forma, assumindo a aproximação (2.8) e a condição de continuidade (2.2), podemos reescrever as equações de momento para as componentes de velocidade e para a temperatura da seguinte forma:

$$\begin{aligned}
\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} &= \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) - \frac{\partial u^2}{\partial x} - \frac{\partial uv}{\partial y} - \frac{\partial uw}{\partial z} \\
&- \left[1 - \beta(T - T_0)\right] g_x
\end{aligned} \tag{2.9}$$

$$\begin{aligned}
\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} &= \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) - \frac{\partial uv}{\partial x} - \frac{\partial v^2}{\partial y} - \frac{\partial vw}{\partial z} \\
&- \left[1 - \beta(T - T_0)\right] g_y
\end{aligned} \tag{2.10}$$

$$\begin{aligned}
\frac{\partial w}{\partial t} + \frac{\partial p}{\partial z} &= \frac{1}{Re}\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) - \frac{\partial uw}{\partial x} - \frac{\partial vw}{\partial y} - \frac{\partial w^2}{\partial z} \\
&- \left[1 - \beta(T - T_0)\right] g_z
\end{aligned} \tag{2.11}$$

$$\frac{\partial T}{\partial t} = \frac{1}{Re}\frac{1}{Pr}\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) - \frac{\partial uT}{\partial x} - \frac{\partial vT}{\partial y} - \frac{\partial wT}{\partial z} + q \tag{2.12}$$

2.2 Condições Iniciais e de Contorno

Para completar a formulação matemática do problema é necessário impor condições iniciais e condições de contorno do domínio de definição do problema Ω.

Na modelagem são impostas, inicialmente, condições iniciais ($t = 0$) sobres as velocidades, a pressão e a temperatura, $u = u_0(x, y, z)$, $v = v_0(x, y, z)$, $w = w_0(x, y, z)$, $p = p_0(x, y, z)$ e $T = T_0(x, y, z)$ de tal forma que as velocidades satisfaçam a Equação (2.2).

As condições de contorno para pressão e a temperatura são nulas ($p = T = 0$); para as velocidades, no caso tridimensional, considerando domínios com alguma superfície paralela à um plano coordenado, obtêm-se dois componentes para a velocidade tangencial (u_{t1}) e (u_{t2}) e uma velocidade normal u_n, conforme visto na Figura 1. Para os pontos ao longo do contorno do domínio e para os pontos que definem contornos de obstáculos internos ao domínio, caso hajam, consideram-se as seguintes condições de contorno:

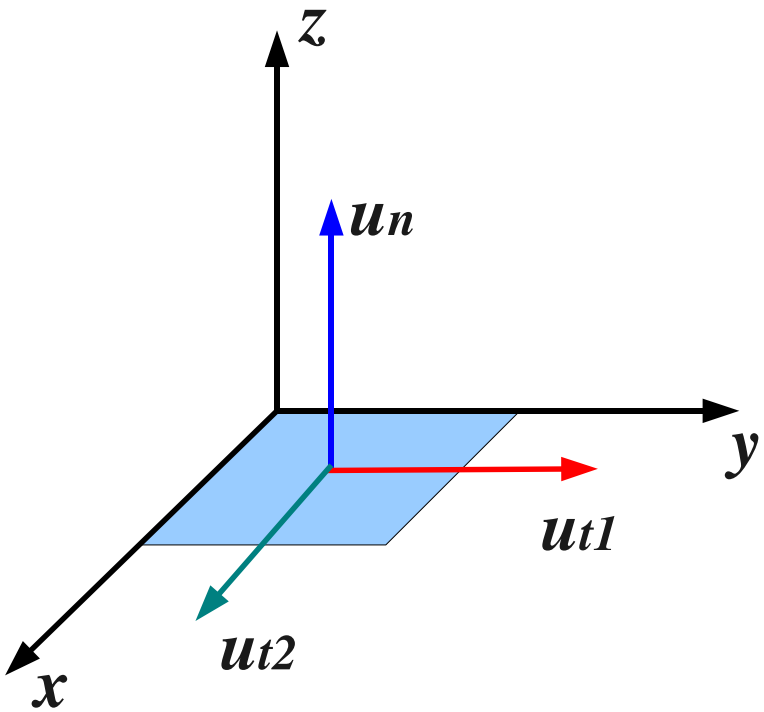


Figura 1: Componentes das velocidades normal (u_n) e tangenciais (u_{t1} e u_{t2}) para o plano xy.

Condição tipo Não-deslizamento : condição de contorno onde a velocidade normal é nula ($u_n = 0$), uma vez que o fluido não pode penetrar no contorno do domínio ou do obstáculo, e a velocidade tangencial também é nula ($u_t = 0$), o que reflete o fato do fluido, imediatamente adjacente à superfície do contorno, estar em repouso em relação ao mesmo.

Condição tipo Deslizamento : igualmente à condição tipo não-deslizamento, a velocidade normal é nula ($v_n = 0$). No entanto, a derivada normal da velocidade tangencial é nula, ou seja, ($\partial u_t/n = 0$), o que significa que a velocidade tangencial não muda na direção normal. Essa condição é muitas vezes imposta ao longo de uma linha ou de um plano de simetria no problema, assim reduzindo o tamanho do domínio do problema.

Condição tipo Entrada : condição de contorno onde as velocidades em uma região de entrada de fluido são explicitamente fornecidas.

Condição tipo Saída : condição de contorno onde as derivadas normais das velocidades normal (u_n) e tangencial (u_t) são nulas, ou seja, $\partial u_n/\partial n = \partial u_t/\partial n = 0$. Isso reflete o fato de que a velocidade total não muda na direção normal do contorno.

A Figura 2 resume as diferentes condições de contorno apresentadas para um domínio geral que representa um escoamento em um duto.

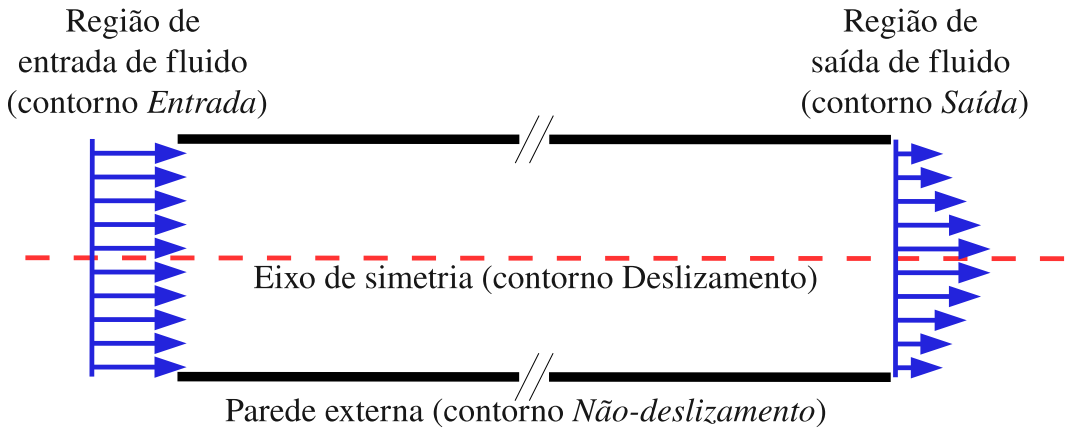


Figura 2: Exemplificação das condições de contorno de um escoamento em um duto.

2.3 Algoritmo Numérico – Aproximações

O tratamento numérico das equações de Navier-Stokes e de transporte é baseado em um esquema de diferenças finitas como sugerido em (GRIEBEL, 1998). O domínio do problema é discretizado, no caso tridimensional, em i_{max} células na direção x, j_{max} células na direção y e k_{max} células na direção z, todas com o mesmo tamanho. A região é discretizada usando uma malha deslocada (*staggered*, em inglês), onde as componentes da velocidade em cada dimensão e a pressão são calculados em pontos distintos. A temperatura T é calculada na mesma posição da pressão p. Esta disposição das incógnitas

evita possíveis oscilações na pressão que poderiam ocorrer caso as incógnitas envolvidas fossem avaliadas no mesmo ponto (FORTUNA, 2000; GRIEBEL, 1998). A Figura 3 apresenta o esquema de uma malha deslocada para o caso bidimensional, onde a pressão p está localizada no centro das células, a velocidade horizontal u nos pontos médios da face vertical das células e a velocidade vertical v nos pontos médios da face horizontal das células. Para o caso tridimensional, conforme apresentado na Figura 4, a pressão p é calculada no centro do cubo e as componentes das velocidades são calculadas nos centros das correspondentes faces do cubo.

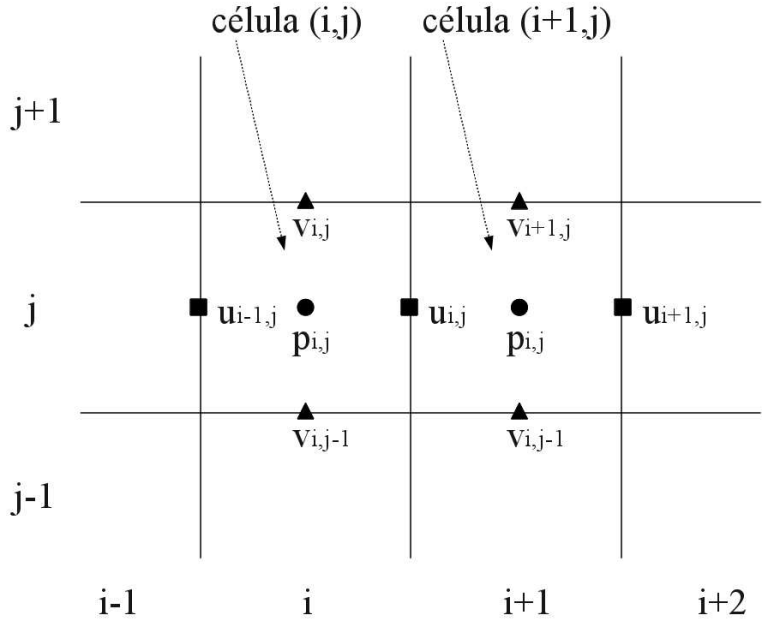


Figura 3: Malha deslocada para o caso bidimensional. A pressão p é calculada no centro da célula, a velocidade horizontal u é calculada no ponto médio da face vertical direita da célula e a velocidade vertical é calculada no ponto médio da face superior da célula.

Sendo assim, para este esquema de malha deslocada, em cada célula (i, j, k), que ocupa uma região espacial $[(i-1)\delta x, i\delta x] \times [(j-1)\delta y, j\delta y] \times [(k-1)\delta z, k\delta z]$, a pressão $p_{i,j,k}$ é localizada nas coordenadas $((i-0.5)\delta x, (j-0.5)\delta y, (k-0.5)\delta z)$, a velocidade $u_{i,j,k}$ é localizada nas coordenadas $(i\delta x, (j-0.5)\delta y, (k-0.5)\delta z)$, a velocidade $v_{i,j,k}$ é localizada nas coordenadas $((i-0.5)\delta x, j\delta y, (k-0.5)\delta z)$ e a velocidade $w_{i.j.k}$ é localizada nas coordenadas $((i-0.5)\delta x, (j-0.5)\delta y, k\delta z)$.

Consequentemente, nem todas as incógnitas das células se situam no interior do domínio ou em seu contorno. Considerando um domínio bidimensional, os limites verticais, por exemplo, possuem valores de v que não pertencem ao domínio e não possui valores no contorno do domínio, como mostrado na Figura 5. Assim, também os limites horizontais possuem valores de u que são exteriores ao domínio e não contém valores em

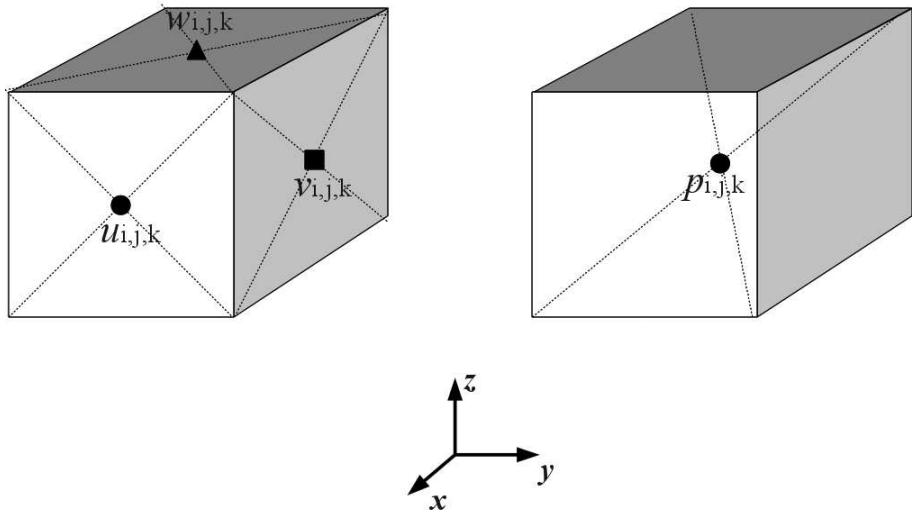


Figura 4: Disposição das velocidades u, v e w e da pressão p em uma célula tridimensional.

seu contorno. Por esta razão, é introduzida na malha, uma faixa de células de limite de maneira que as condições de contorno podem ser aplicadas pela média dos pontos adjacentes próximos a esta faixa, Figura 5. As faixas de células de limite podem ser observadas na Figura 6 para o caso bidimensional. Como isso as seguintes condições de contorno são consideradas:

Condição tipo Não-deslizamento : como o valor da velocidade é nulo, temos

$$v_c := \frac{v_e + v_i}{2} = 0 \Rightarrow v_e = -v_i. \tag{2.13}$$

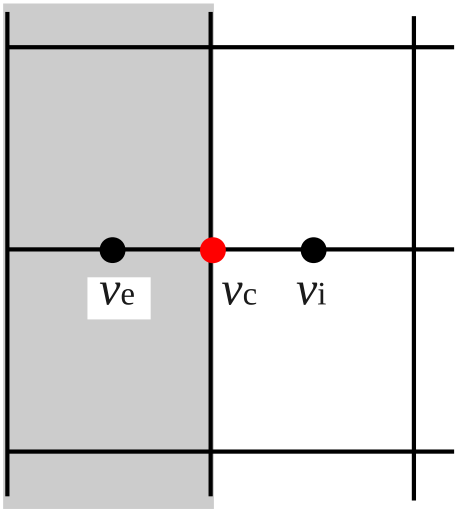


Figura 5: Cálculo do valor de contorno v_r na fronteira esquerda de um domínio bidimensional.

Condição tipo Deslizamento : como o valor da derivada normal é nulo ($\partial v/\partial n = 0$), discretizando esta derivada temos:

$$v_c := \frac{v_e - v_i}{\partial x} = 0 \Rightarrow v_e = v_i. \tag{2.14}$$

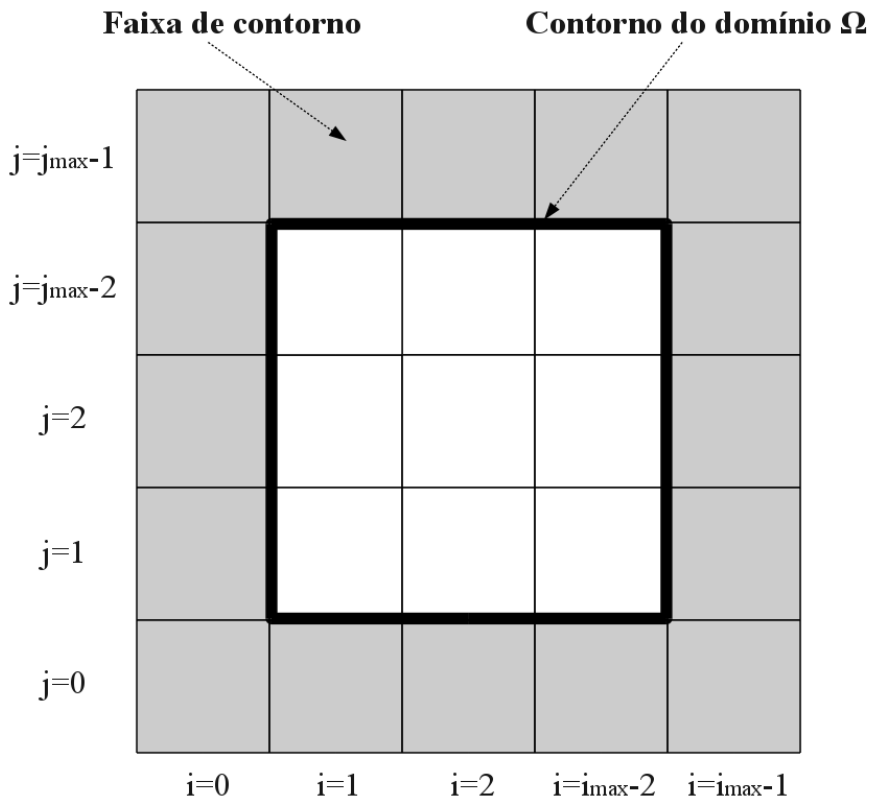


Figura 6: Domínio bidimensional com células de contorno.

2.3.1 Derivada Temporal

Dado um intervalo de tempo $[0, t_{final}]$ de análise, definimos $t_{n+1} = t_n + \delta t$. Para a discretização das derivadas temporais no tempo t_{n+1} nós utilizamos o método de Euler explícito que possui aproximação de primeira ordem, (FORTUNA, 2000):

$$\left[\frac{\partial u}{\partial t}\right]^{(n+1)} := \frac{u^{(n+1)} - u^{(n)}}{\delta t} \tag{2.15}$$

$$\left[\frac{\partial v}{\partial t}\right]^{(n+1)} := \frac{v^{(n+1)} - v^{(n)}}{\delta t} \tag{2.16}$$

$$\left[\frac{\partial w}{\partial t}\right]^{(n+1)} := \frac{w^{(n+1)} - w^{(n)}}{\delta t} \tag{2.17}$$

As derivadas são computadas no passo de tempo $(n+1)$ utilizando os valores do passo de tempo (n). Tal abordagem, quando aplicada às equações de momento, produz uma expressão explícita para o cálculo das velocidades no passo de tempo $(n+1)$, ou seja, produz uma expressão que não necessita da solução de um sistema linear.

2.3.2 A Iteração no Tempo

No tempo inicial $t = 0$ são conhecidos os valores iniciais para as velocidades u, v e w, a temperatura T e a pressão p. O tempo é incrementado de δt a cada iteração até se atingir o tempo final de execução T_{final}. Assim, com todos os valores conhecidos no passo n podemos computar os valores para o próximo passo $n+1$.

Utilizando as aproximações das derivadas temporais apresentadas nas Equações (2.15), (2.17) e (2.17) podemos calcular as velocidades no passo $n+1$ reescrevendo as equações de momento da seguinte forma:

$$\begin{aligned} u^{n+1} &= u^n + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) - \frac{\partial u^2}{\partial x} - \frac{\partial uv}{\partial y} - \frac{\partial uw}{\partial z} + g_x \right. \\ &- \left. \frac{\partial p}{\partial x} \right]^{(n)} - \beta \delta t \left(1 - (T^{(n+1)} - T_0) \right) g_x \end{aligned} \tag{2.18}$$

$$\begin{aligned} v^{n+1} &= v^n + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) - \frac{\partial vu}{\partial x} - \frac{\partial v^2}{\partial y} - \frac{\partial vw}{\partial z} + g_y \right. \\ &- \left. \frac{\partial p}{\partial y} \right]^{(n)} - \beta \delta t \left(1 - (T^{(n+1)} - T_0) \right) g_y \end{aligned} \tag{2.19}$$

$$\begin{aligned} w^{n+1} &= w^n + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) - \frac{\partial wu}{\partial x} - \frac{\partial wv}{\partial y} - \frac{\partial w^2}{\partial z} + g_z \right. \\ &- \left. \frac{\partial p}{\partial z} \right]^{(n)} - \beta \delta t \left(1 - (T^{(n+1)} - T_0) \right) g_z \end{aligned} \tag{2.20}$$

Utilizando a seguinte abreviação:

$$
\begin{aligned}
F^{(n)} &:= u^n + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) - \frac{\partial u^2}{\partial x} - \frac{\partial uv}{\partial y} - \frac{\partial uw}{\partial z} + g_x \right]^{(n)} \\
&- \beta \delta t \left(1 - (T^{(n+1)} - T_0) \right) g_x \qquad (2.21) \\
G^{(n)} &:= v^n + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2} \right) - \frac{\partial uv}{\partial x} - \frac{\partial v^2}{\partial y} - \frac{\partial vw}{\partial z} + g_y \right]^{(n)} \\
&- \beta \delta t \left(1 - (T^{(n+1)} - T_0) \right) g_y \qquad (2.22) \\
H^{(n)} &:= w^n + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2} \right) - \frac{\partial wu}{\partial x} - \frac{\partial wv}{\partial y} - \frac{\partial w^2}{\partial z} + g_z \right]^{(n)} \\
&- \beta \delta t \left(1 - (T^{(n+1)} - T_0) \right) g_z \qquad (2.23)
\end{aligned}
$$

Podemos escrever:

$$
u^{(n+1)} = F^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial x} \qquad (2.24)
$$

$$
v^{(n+1)} = G^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y} \qquad (2.25)
$$

$$
w^{(n+1)} = H^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial z} \qquad (2.26)
$$

A equação de transporte pode ser escrita de seguinte forma:

$$
T^{(n+1)} = T^{(n)} + \delta t \left[\frac{1}{Re} \frac{1}{Pr} \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) - \frac{\partial uT}{\partial x} - \frac{\partial vT}{\partial y} - \frac{\partial wT}{\partial z} + q \right]^{(n)} \qquad (2.27)
$$

Substituindo as Equações (2.24), (2.25) e (2.26), as quais calculam o campo de velocidade no passo t_{n+1}, na equação da continuidade (2.2) obtemos:

$$
\begin{aligned}
0 &= \frac{\partial u^{(n+1)}}{\partial x} + \frac{\partial v^{(n+1)}}{\partial y} + \frac{\partial w^{(n+1)}}{\partial z} \\
&= \frac{\partial F^{(n)}}{\partial x} - \delta t \frac{\partial^2 p^{(n+1)}}{(\partial x)^2} + \frac{\partial G^{(n)}}{\partial y} - \delta t \frac{\partial^2 p^{(n+1)}}{(\partial y)^2} + \frac{\partial H^{(n)}}{\partial z} - \delta t \frac{\partial^2 p^{(n+1)}}{(\partial z)^2} \qquad (2.28)
\end{aligned}
$$

Rearranjando esta equação obtemos a equação de Poisson para se calcular a pressão no passo t_{n+1}:

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} + \frac{\partial^2 p^{(n+1)}}{\partial z^2} = \frac{1}{\delta t}\left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y} + \frac{\partial H^{(n)}}{\partial z}\right) \tag{2.29}$$

no qual requer os valores de fronteira para a pressão. Neste caso assumimos:

$$\begin{aligned} p_{0,j,k} &= p_{1,j,k}, & p_{imax,j,k} &= p_{imax-1,j,k} \\ p_{i,0,k} &= p_{i,1,k}, & p_{i,jmax,k} &= p_{i,jmax-1,k} \\ p_{i,j,0} &= p_{i,j,0}, & p_{i,j,kmax} &= p_{i,j,kmax-1} \\ i = 1, ..., i_{imax-1}; & \quad j = 1, ..., j_{jmax-1}; & & k = 1, ..., k_{kmax-1} \end{aligned} \tag{2.30}$$

Além disso, também são necessários os valores de F, G e H na fronteira para calcular o lado direito da Equação (2.29). Então é definido:

$$\begin{aligned} F_{0,j,k} &= u_{0,j,k} & F_{imax,j,k} &= u_{imax,j,k} \\ G_{i,0,k} &= v_{i,0,k} & G_{i,jmax,k} &= v_{i,jmax,k} \\ H_{i,j,0} &= w_{i,j,0} & H_{i,j,kmax} &= w_{i,j,kmax} \\ i = 1, ..., i_{imax-1}; & \quad j = 1, ..., j_{jmax-1}; & & k = 1, ..., k_{kmax-1} \end{aligned} \tag{2.31}$$

Dessa maneira a discretização se caracteriza por ser implícita para a pressão e explícita para as velocidades e a temperatura. Assim, o campo de velocidade para o passo de tempo $t_{(n+1)}$ pode ser obtido uma vez que se tenha os valores correspondentes de temperatura e pressão. Resumindo, o algoritmo consiste nos passos mostrados no Algoritmo 2.1. Para os problemas que envolvem apenas as equações de Navier-Stokes, ou seja, não envolve o transporte, é feita o seguinte ajuste no algoritmo 2.1: retira-se a função `comp_T()` do algoritmo (linha 8) e utiliza $\beta = 0$.

Algoritmo 2.1: Algoritmo serial.

```
...
t = 0;
n = 0;
init_UVWP(); // Initializa as variaveis: u, v, w, T e p
while(t < t_end) {
  comp_delt(); // calcula delta t (delt) adaptativamente
  set_BondCond(); // valores de fronteira u, v e w
  comp_T(); // calcula T(n+1) de acordo com (3)
  comp_FGH(); // calcula F(n), G(n) e H(n) de acordo com (4), (5) e (6)
  poisson_syst(); // constroi o sistema linear
  RedBlack_SOR(); // resolve o sistema usando Red-Black-SOR
  comp_UVW(); // calcula u(n+1), v(n+1) e w(n+1) de acordo com (7), (8) e
      (9)
  t = t + delt;
  n = n + 1;
}
...
```

2.3.3 Condição de Estabilidade

Um fator importante que deve ser considerado é a escolha do tamanho do passo no tempo (δt), pois um valor inadequado pode gerar oscilações numéricas, já que o esquema explícito é condicionalmente estável. Utilizamos um controle adaptativo para a escolha do passo no tempo apresentado em (GRIEBEL, 1998) que, para o caso tridimensional, considera quatro condições e deve satisfazer a seguinte expressão:

$$\delta t := \tau \ \min \left(\frac{Re}{2} \left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2} + \frac{1}{\delta z^2} \right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|}, \frac{\delta z}{|w_{max}|} \right) \tag{2.32}$$

onde τ é um parâmetro escolhido no intervalo $]0, 1]$ e $|u_{max}|$, $|v_{max}|$ e $|w_{max}|$ são as velocidades máximas absolutas para as direções x, y e z, respectivamente.

2.3.4 Derivadas Espaciais

As discretizações dos termos de primeira ordem ($\partial u/\partial x$, $\partial v/\partial y$, $\partial w/\partial z$, $\partial p/\partial x$, $\partial p/\partial u$ e $\partial p/\partial z$) e dos termos de segunda ordem ($\partial^2 u/\partial x^2$, $\partial^2 u/\partial y^2$, $\partial^2 u/\partial z^2$, $\partial^2 v/\partial x^2$, $\partial^2 v/\partial y^2$, $\partial^2 v/\partial z^2$, $\partial^2 w/\partial x^2$, $\partial^2 w/\partial y^2$, $\partial^2 w/\partial z^2$, $\partial^2 T/\partial x^2$, $\partial^2 T/\partial y^2$ e $\partial^2 T/\partial z^2$), que formam o termo difusivo, são feitas por meio de diferenças finitas adiantadas e diferenças finitas

centrais respectivamente, como apresentado em (GRIEBEL, 1998). Para discretizar os termos convectivos ($\partial u^2/\partial x$, $\partial uv/\partial y$, $\partial uw/\partial z$, $\partial uv/\partial x$, $\partial v^2/\partial y$, $\partial vw/\partial z$, $\partial uw/\partial x$, $\partial vw/\partial y$, $\partial w^2/\partial z$, $\partial uT/\partial x$, $\partial vT/\partial y$ e $\partial wT/\partial z$) é necessário uma estratégia especial por dois motivos, explicados a seguir. Primeiro podem ocorrer problemas de estabilidade quando tais termos se tornam dominantes (número de Reynolds grande ou altos valores de velocidade) ou quando o espaçamento escolhido para a malha for muito grosseiro (FORTUNA, 2000). Segundo, devido a necessidade de serem considerados pontos não definidos no domínio para a discretização por causa da utilização da malha deslocada, conforme apresentado nas Figuras 5 e 6. Uma forma de discretizar o termo $\partial uv/\partial y$, como sugerido por (GRIEBEL, 1998), é calcular a média u e v nos pontos marcados com $\times$, mostrado na Figura 7. De forma semelhante pode-se discretizar o termo $\partial u^2/\partial x$ tomando a diferença central dos pontos marcado com + (Figura 7).

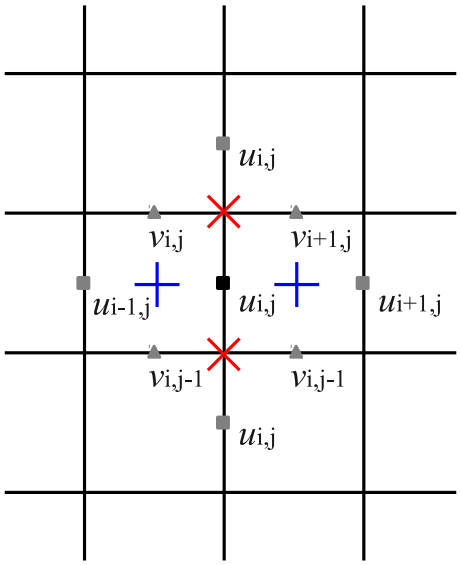


Figura 7: Interpolação de pontos não definidos na malha.

Para contornar a oscilação numérica de um escoamento com convecção dominante é utilizado o esquema *donor-cell* descrito por (GRIEBEL, 1998) como critério de estabilização. Neste esquema, considerando os termos convectivos na forma $\partial ku/\partial x$ definidos em um domínio unidimensional, assumindo que k seja dado nos pontos médios, conforme Figura 8, a discretização de tais termos é dado por:

$$\left[\frac{\partial ku}{\partial x}\right]_i^{dc} = \frac{k_r u_r - k_l u_l}{\delta x} \tag{2.33}$$

onde k_r é o ponto médio para o lado direito (*right*) da aproximação e k_l e o ponto médio para o lado esquerdo (*left*) da aproximação (Figura 8). Os valores de u_r e u_l são escolhidos

de acordo com o sinal de k_r e k_l:

$$u_r = \begin{cases} u_i, & k_r > 0 \\ u_{i+1}, & k_r < 0 \end{cases} \qquad u_l = \begin{cases} u_{i-1}, & k_l > 0 \\ u_i, & k_l < 0 \end{cases} \tag{2.34}$$

Também podemos escrever a Equação (2.33) como:

$$\left[\frac{\partial ku}{\partial x}\right]_i^{dc} = \frac{1}{2\delta x}\left(k_r(u_i + u_{i+1}) - k_l(u_{i-1} + u_i) + |k_r|(u_i - u_{i+1}) - |k_l|(u_{i-1} - u_i)\right) \tag{2.35}$$

O esquema *donor-cel* possui uma aproximação de primeira ordem. Por causa dos problemas de estabilização já mencionados é necessário usar uma mistura de discretização por diferenças centrais e uma discretização com o esquema *donor-cell*, para controlar esta mistura utilizamos o parâmetro γ.

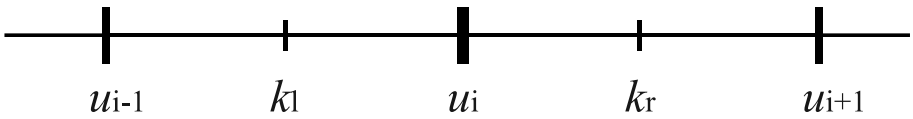


Figura 8: Discretização utilizando esquema *donor-cell*.

Sendo assim, as derivadas espaciais, considerando uma discretização tridimensional, são discretizadas da seguinte forma (GRIEBEL, 1998; THIBAULT, 2009):

Termos relativos a velocidade u :

$$
\begin{aligned}
\left[\frac{\partial u^2}{\partial x}\right]_{i,j,k} &= \frac{1}{\delta x}\left[\left(\frac{u_{i+1,j,k}+u_{i,j,k}}{2}\right)^2-\left(\frac{u_{i,j,k}+u_{i-1,j,k}}{2}\right)^2\right] \\
&+ \gamma\frac{1}{\delta x}\left[\frac{|u_{i+1,j,k}+u_{i,j,k}|}{2}\frac{(u_{i+1,j,k}-u_{i,j,k})}{2}\right. \\
&- \left.\frac{|u_{i,j,k}+u_{i-1,j,k}|}{2}\frac{(u_{i,j,k}-u_{i-1,j,k})}{2}\right] \qquad (2.36)\\
\left[\frac{\partial uv}{\partial y}\right]_{i,j,k} &= \frac{1}{\delta y}\left[\frac{(u_{i,j,k}+u_{i,j+1,k})}{2}\frac{(v_{i,j,k}+v_{i+1,j,k})}{2}\right. \\
&- \left.\frac{(u_{i,j,k}+u_{i-1,j,k})}{2}\frac{(v_{i,j-1,k}+v_{i+1,j-1,k})}{2}\right] \\
&+ \gamma\frac{1}{\delta y}\left[\frac{|u_{i,j,k}+u_{i,j+1,k}|}{2}\frac{(v_{i,j,k}-v_{i+1,j,k})}{2}\right. \\
&- \left.\frac{|u_{i,j,k}+u_{i-1,j,k}|}{2}\frac{(v_{i,j-1,k}-v_{i+1,j-1,k})}{2}\right] \qquad (2.37)\\
\left[\frac{\partial uw}{\partial z}\right]_{i,j,k} &= \frac{1}{\delta z}\left[\frac{(u_{i,j,k}+u_{i,j,k+1})}{2}\frac{(w_{i,j,k}+w_{i+1,j,k})}{2}\right. \\
&- \left.\frac{(u_{i,j,k}+u_{i,j,k-1})}{2}\frac{(w_{i,j,k-1}+w_{i+1,j,k-1})}{2}\right] \\
&+ \gamma\frac{1}{\delta z}\left[\frac{|u_{i,j,k}+u_{i,j,k+1}|}{2}\frac{(w_{i,j,k}-w_{i+1,j,k})}{2}\right. \\
&- \left.\frac{|u_{i,j,k}+u_{i,j,k-1}|}{2}\frac{(w_{i,j,k-1}-w_{i+1,j,k-1})}{2}\right] \qquad (2.38)\\
\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j,k} &= \frac{u_{i+1,j,k}-2u_{i,j,k}+u_{i-1,j,k}}{(\delta x)^2} \qquad (2.39)\\
\left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j,k} &= \frac{u_{i,j+1,k}-2u_{i,j,k}+u_{i,j+1,k}}{(\delta y)^2} \qquad (2.40)\\
\left[\frac{\partial^2 u}{\partial z^2}\right]_{i,j,k} &= \frac{u_{i,j,k+1}-2u_{i,j,k}+u_{i,j,k-1}}{(\delta z)^2} \qquad (2.41)\\
\left[\frac{\partial p}{\partial x}\right]_{i,j,k} &= \frac{p_{i+1,j,k}-p_{i,j,k}}{\delta x} \qquad (2.42)
\end{aligned}
$$

Termos relativos a velocidade $\boldsymbol{v}$:

$$
\begin{aligned}
\left[\frac{\partial uv}{\partial x}\right]_{i,j,k} &= \frac{1}{\delta x}\left[\frac{(u_{i,j,k}+u_{i,j+1,k})}{2}\frac{(v_{i+1,j,k}+v_{i,j,k})}{2}\right. \\
&- \left.\frac{(u_{i-1,j,k}+u_{i-1,j+1,k})}{2}\frac{(v_{i,j,k}+v_{i-1,j,k})}{2}\right] \\
&+ \gamma\frac{1}{\delta x}\left[\frac{|u_{i,j,k}+u_{i,j+1,k}|}{2}\frac{(v_{i+1,j,k}-v_{i,j,k})}{2}\right. \\
&- \left.\frac{|u_{i-1,j,k}+u_{i-1,j+1,k}|}{2}\frac{(v_{i,j,k}-v_{i-1,j,k})}{2}\right] \qquad (2.43)
\end{aligned}
$$

$$
\begin{aligned}
\left[\frac{\partial v^2}{\partial y}\right]_{i,j,k} &= \frac{1}{\delta x}\left[\left(\frac{v_{i,j+1,k}+v_{i,j,k}}{2}\right)^2-\left(\frac{v_{i,j,k}+v_{i,j-1,k}}{2}\right)^2\right] \\
&+ \gamma\frac{1}{\delta y}\left[\frac{|v_{i,j+1,k}+v_{i,j,k}|}{2}\frac{(v_{i,j+1,k}-v_{i,j,k})}{2}\right. \\
&- \left.\frac{|v_{i,j,k}+v_{i,j-1,k}|}{2}\frac{(v_{i,j,k}-v_{i-1,j,k})}{2}\right] \qquad (2.44)
\end{aligned}
$$

$$
\begin{aligned}
\left[\frac{\partial vw}{\partial z}\right]_{i,j,k} &= \frac{1}{\delta z}\left[\frac{(v_{i,j,k}+v_{i,j,k+1})}{2}\frac{(w_{i,j,k}+w_{i,j+1,k})}{2}\right. \\
&- \left.\frac{(v_{i,j,k}+v_{i,j,k-1})}{2}\frac{(w_{i,j,k-1}+w_{i,j+1,k-1})}{2}\right] \\
&+ \gamma\frac{1}{\delta z}\left[\frac{|v_{i,j,k}+v_{i,j,k+1}|}{2}\frac{(w_{i,j,k}-w_{i,j+1,k})}{2}\right. \\
&- \left.\frac{|v_{i,j,k}+v_{i,j,k-1}|}{2}\frac{(w_{i,j,k-1}-w_{i,j+1,k-1})}{2}\right] \qquad (2.45)
\end{aligned}
$$

$$
\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j,k} = \frac{v_{i+1,j,k}-2v_{i,j,k}+v_{i-1,j,k}}{(\delta x)^2} \qquad (2.46)
$$

$$
\left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j,k} = \frac{v_{i,j+1,k}-2v_{i,j,k}+v_{i,j+1,k}}{(\delta y)^2} \qquad (2.47)
$$

$$
\left[\frac{\partial^2 v}{\partial z^2}\right]_{i,j,k} = \frac{v_{i,j,k+1}-2v_{i,j,k}+v_{i,j,k-1}}{(\delta z)^2} \qquad (2.48)
$$

$$
\left[\frac{\partial p}{\partial y}\right]_{i,j,k} = \frac{p_{i,j+1,k}-p_{i,j,k}}{\delta x} \qquad (2.49)
$$

Termos relativos a velocidade $\boldsymbol{w}$:

$$
\begin{aligned}
\left[\frac{\partial uw}{\partial x}\right]_{i,j,k} &= \frac{1}{\delta x}\left[\frac{(u_{i,j,k}+u_{i,j,k+1})}{2}\frac{(w_{i+1,j,k}+w_{i,j,k})}{2}\right. \\
&- \left.\frac{(u_{i-1,j,k}+u_{i-1,j,k+1})}{2}\frac{(w_{i,j,k}+w_{i-1,j,k})}{2}\right] \\
&+ \gamma\frac{1}{\delta x}\left[\frac{|u_{i,j,k}+u_{i,j,k+1}|}{2}\frac{(w_{i+1,j,k}-w_{i,j,k})}{2}\right. \\
&- \left.\frac{|u_{i-1,j,k}+u_{i-1,j,k+1}|}{2}\frac{(w_{i,j,k}-w_{i-1,j,k})}{2}\right] \qquad (2.50)
\end{aligned}
$$

$$
\begin{aligned}
\left[\frac{\partial vw}{\partial y}\right]_{i,j,k} &= \frac{1}{\delta y}\left[\frac{(v_{i,j,k}+v_{i,j,k+1})}{2}\frac{(w_{i,j,k}+w_{i,j+1,k})}{2}\right. \\
&- \left.\frac{(v_{i,j-1,k}+v_{i,j-1,k+1})}{2}\frac{(w_{i,j,k}+w_{i,j-1,k})}{2}\right] \\
&+ \gamma\frac{1}{\delta y}\left[\frac{|v_{i,j,k}+v_{i,j,k+1}|}{2}\frac{(w_{i,j+1,k}-w_{i,j,k})}{2}\right. \\
&- \left.\frac{|v_{i,j-1,k}+v_{i,j-1,k+1}|}{2}\frac{(w_{i,j,k}-w_{i,j-1,k})}{2}\right] \qquad (2.51)
\end{aligned}
$$

$$
\begin{aligned}
\left[\frac{\partial w^2}{\partial z}\right]_{i,j,k} &= \frac{1}{\delta z}\left[\left(\frac{w_{i,j,k+1}+w_{i,j,k}}{2}\right)^2-\left(\frac{w_{i,j,k}+w_{i,j,k-1}}{2}\right)^2\right] \\
&+ \gamma\frac{1}{\delta y}\left[\frac{|w_{i,j,k+1}+w_{i,j,k}|}{2}\frac{(w_{i,j,k+1}-w_{i,j,k})}{2}\right. \\
&- \left.\frac{|w_{i,j,k}+w_{i,j,k-1}|}{2}\frac{(w_{i,j,k}-w_{i,j,k-1})}{2}\right] \qquad (2.52)
\end{aligned}
$$

$$
\left[\frac{\partial^2 w}{\partial x^2}\right]_{i,j,k} = \frac{w_{i+1,j,k}-2w_{i,j,k}+w_{i-1,j,k}}{(\delta x)^2} \qquad (2.53)
$$

$$
\left[\frac{\partial^2 w}{\partial y^2}\right]_{i,j,k} = \frac{w_{i,j+1,k}-2w_{i,j,k}+w_{i,j+1,k}}{(\delta y)^2} \qquad (2.54)
$$

$$
\left[\frac{\partial^2 w}{\partial z^2}\right]_{i,j,k} = \frac{w_{i,j,k+1}-2w_{i,j,k}+w_{i,j,k-1}}{(\delta z)^2} \qquad (2.55)
$$

$$
\left[\frac{\partial p}{\partial z}\right]_{i,j,k} = \frac{p_{i,j,k+1}-p_{i,j,k}}{\delta z} \qquad (2.56)
$$

Termos relativos a temperatura T :

$$\left[\frac{\partial T}{\partial t}\right]_{i,j,k}^{(n+1)} = \frac{1}{\delta t}\left(T_{i,j,k}^{(n+1)} - T_{i,j,k}^{(n)}\right) \tag{2.57}$$

$$\begin{aligned}\left[\frac{\partial uT}{\partial x}\right]_{i,j,k} &= \frac{1}{\partial x}\left(u_{i,j,k}\frac{T_{i,j,k}+T_{i+1,j,k}}{2} - u_{i-1,j,k}\frac{T_{i-1,j,k}+T_{i,j,k}}{2}\right) \\ &- \frac{\gamma}{\partial x}\left(|u_{i,j,k}|\frac{T_{i,j,k}-T_{i+1,j,k}}{2} - |u_{i-1,j,k}|\frac{T_{i-1,j,k}-T_{i,j,k}}{2}\right)\end{aligned} \tag{2.58}$$

$$\begin{aligned}\left[\frac{\partial vT}{\partial y}\right]_{i,j,k} &= \frac{1}{\partial y}\left(v_{i,j,k}\frac{T_{i,j,k}+T_{i,j+1,k}}{2} - v_{i,j-1,k}\frac{T_{i,j-1,k}+T_{i,j,k}}{2}\right) \\ &- \frac{\gamma}{\partial y}\left(|v_{i,j,k}|\frac{T_{i,j,k}-T_{i,j+1,k}}{2} - |v_{i,j-1,k}|\frac{T_{i,j-1,k}-T_{i,j,k}}{2}\right)\end{aligned} \tag{2.59}$$

$$\begin{aligned}\left[\frac{\partial wT}{\partial z}\right]_{i,j,k} &= \frac{1}{\partial z}\left(w_{i,j,k}\frac{T_{i,j,k}+T_{i,j,k+1}}{2} - w_{i,j,k-1}\frac{T_{i,j,k-1}+T_{i,j,k}}{2}\right) \\ &- \frac{\gamma}{\partial x}\left(|u_{i,j,k}|\frac{T_{i,j,k}-T_{i,j,k+1}}{2} - |w_{i,j,k-1}|\frac{T_{i,j,k-1}-T_{i,j,k}}{2}\right)\end{aligned} \tag{2.60}$$

$$\left[\frac{\partial^2 T}{\partial x}\right]_{i,j,k} = \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{(\delta x)^2} \tag{2.61}$$

$$\left[\frac{\partial^2 T}{\partial y}\right]_{i,j,k} = \frac{T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j,k-1}}{(\delta y)^2} \tag{2.62}$$

$$\left[\frac{\partial^2 T}{\partial z}\right]_{i,j,k} = \frac{T_{i,j,k+1} - 2T_{i,j,k} + T_{i,j,k-1}}{(\delta z)^2} \tag{2.63}$$

2.4 Método Iterativo Red-Black-SOR

Na solução do sistema linear originado da Equação de Poisson (2.29) é utilizado um esquema de coloração Red-Black (SHINN; VANKA, 2009) para as células internas da malha e o método iterativo *sucessive-over-relaxation* (SOR), denominado Red-Black-SOR.

A matriz septa-diagonal originária do sistema linear para o caso 3D foi construída e armazenada em sete vetores de $i_{max} \times j_{max} \times k_{max}$ posições. O ordenamento escolhido na implementação do método é apresentado na Figura 9. A numeração é feita por superfícies nos planos xy. Desta forma iniciamos a numeração no primeiro plano xy na posição $[0; 0; 0]$. A numeração segue na direção do eixo x da esquerda para a direita. As próximas linhas da numeração seguem o eixo y, formando assim o plano xy. Em seguida, avançamos no eixo z para os próximos planos. As matrizes utilizadas no algoritmo proposto foram armazenadas linearmente em vetores de tamanho $i_{max} \times j_{max} \times k_{max}$, ou seja, cada plano da malha é armazenado de forma concatenada por linha, onde nas primeiras posições

estão os valores relativos da primeira linha, depois os valores da segunda linha e assim por diante.

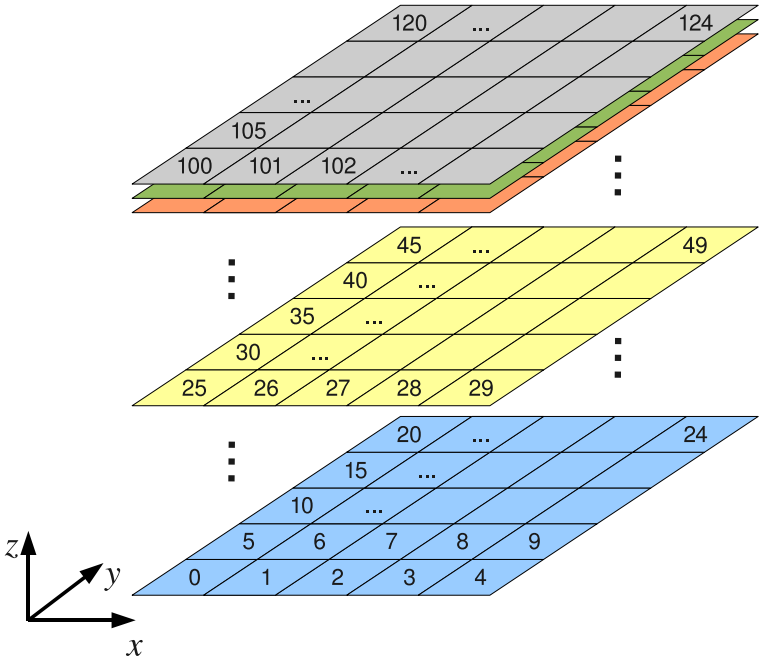


Figura 9: Ordenamento para uma malha 3D.

Todas as matrizes utilizadas são linearizadas conforme a numeração já descrita, assim o cálculo de um endereço para um elemento $p_{i,j,k}$ é feito da seguinte forma:

$$p_{i,j,k} = p[\mathtt{i_{max}} \times \mathtt{j_{max}} \times k + \mathtt{i_{max}} \times j + i] \tag{2.64}$$

No método a malha é colorida como um tabuleiro de xadrez, como apresentado na Figura 10. O objetivo desta coloração é executar primeiramente as células vermelhas (*red*) e em seguida as células pretas (*black*). Assim a atualização das células vermelhas da iteração it para $it+1$ dependem apenas das células pretas na iteração it. Depois as células pretas são atualizadas de it para $it+1$ sendo dependente somente das células vermelhas na iteração $it+1$, como mostrado a seguir:

Loop nas células vermelhas:

$$p_P^{it+1} = (1-w)p_P^{it} + \frac{w}{a_P}\left(a_D p_D^{it} + a_E p_E^{it} + a_B p_B^{it} + a_F p_F^{it} + a_W p_W^{it} + a_U p_U^{it} - rhs_P\right) \tag{2.65}$$

Loop nas células pretas:

$$p_P^{it+1} = (1-w)p_P^{it+1} + \frac{w}{a_P}\left(a_D p_D^{it+1} + a_E p_E^{it+1} + a_B p_B^{it+1} + a_F p_F^{it+1} + a_W p_W^{it+1} + a_U p_U^{it+1} - rhs_P\right) \tag{2.66}$$

A abreviação rhs_P representa o lado direito na equação da pressão (2.29) na célula P, D = sul (*down*), U = norte (*up*), E = leste (*east*), W = oeste (*west*), B = atrás (*back*) e F = frente (*front*), conforme a Figura 10. O parâmetro w deve ser escolhido no intervalo $[0, 2]$ e essa escolha afeta fortemente a taxa de convergência do método. Um valor bastante usado é $w = 1.7$. O processo iterativo é finalizado quando a norma do máximo da diferença entre duas iterações sucessivas é menor que uma dada tolerância ou quando for considerado um número fixo de iterações.

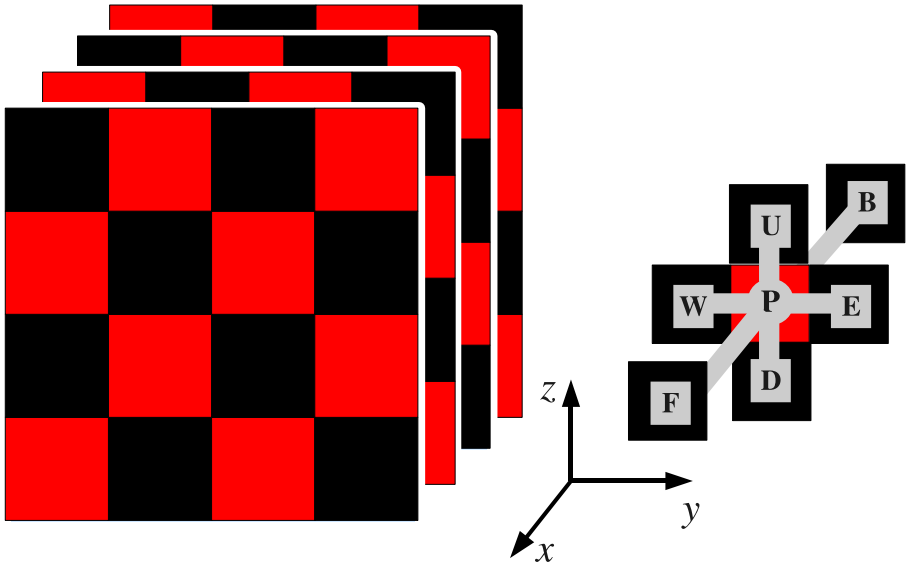


Figura 10: Esquema de coloração Red-Black para as células internas da malha.

3 Arquitetura CUDA

Neste capítulo são apresentados alguns conceitos fundamentais da arquitetura e do modelo de programação paralela desenvolvido especificamente para as placas gráficas da NVIDIA, denominada CUDA (*Compute Unified Device Architecture*). Em seguida apresentamos alguns trechos do código paralelo desenvolvido para as Equações de Navier-Stokes e de Transporte.

3.1 Breve Histórico das GPUs

Em 1999 foi lançada a primeira GPU, denominada GeForce 256, e continha instruções de ponto flutuante de 32-bits para transformação de vértices e processamento de iluminação e um *pipeline* inteiro para funções de cálculo de pixel programável por meio de APIs (*Application Programming Interfaces*) OpenGL e Microsoft DX7. Em 2001 a GPU GeForce 3 introduziu o primeiro processador de vértice programável, com um *pipeline* de ponto flutuante configurável de 32-bits, programável em DX8 e OpenGL. A GPU Radeon 9700, lançada em 2002, era caracterizada por um processador de pixel com ponto flutuante de 24-bits programável por meio de DX9 e OpenGL. A GPU GeForce FX adicionou um processador de pixel com ponto flutuante de 32-bits. O console de vídeo game XBox 360 introduziu a unificação da GPU em 2005, permitindo que vértices e pixels pudessem executar no mesmo processador, (LINDHOLM *et al.*, 2008).

A constante evolução das GPUs, motivada principalmente pela grande demanda do mercado de jogos de alta definição em tempo real, as tornaram tais placas em processadores com desempenho de ponto flutuante e programabilidade sem precedentes. Em poucos anos o poder computacional das GPUs ultrapassou em muito o das CPUs tanto em capacidade de processamento aritmético, Figura 11, quanto em largura de banda de memória, Figura 12, (NVIDIA, 2009).

Na Figura 11 observamos, a partir do ano de 2003, um salto quanto ao número de

operações de ponto flutuante das GPUs em relação as CPUs. Porém, avanços significativos quanto a precisão dupla (*double precision*) só são obtidos no ano de 2009. Como o mercado gráfico não é muito dependente de grande precisão numérica, não havia necessidade, até então, de suporte eficiente a precisão dupla. Também observamos um maior avanço na largura de banda de memória das GPUs em relação às CPUs, Figura 12, seguindo o salto apresentado no processamento de ponto flutuante.

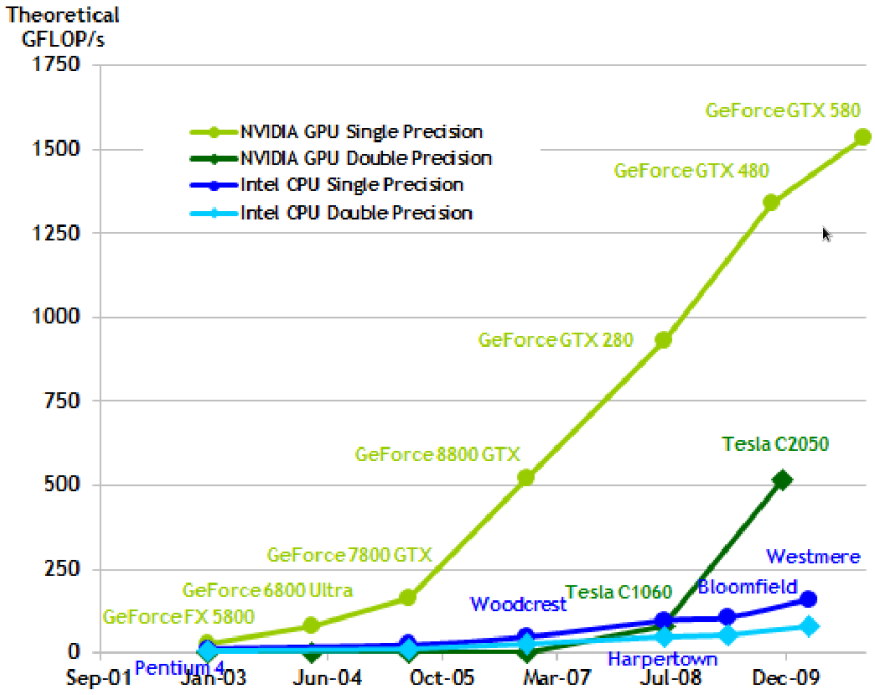


Figura 11: Operações de ponto flutuante por segundo (NVIDIA, 2011b).

Devido a esse grande poder computacional, as GPUs demonstram ser processadores ideias para acelerar o tempo de processamento de simulações computacionais que podem tirar proveito de sua arquitetura. Esforços para empregar GPUs em simulações não gráficas estão em andamento desde 2003, ano em que tanto para os processadores de *pixels* quanto para os de vértices das GPUs passaram a ser programáveis. O lançamento da linguagem Cg pela NVIDIA (RANDIMA; MARK, 2003) nesta mesma época facilitou essa programação, pois esta linguagem é bastante parecida com a linguagem C. Contudo desenvolver código de uso geral utilizando Cg é bastante difícil, uma vez que os programas devem ser escritos dentro do contexto das APIs OpenGl e DirectX, (RANDIMA; MARK, 2003). Esses primeiros esforços que utilizaram APIs gráficas para computação de propósito geral foram denominados GPGPU (*General-Purpose computing on graphics processing units*). Desde então, diversos problemas tem obtido desempenhos notáveis com o uso de GPUs, dentre os quais podemos citar simulações financeiras, consultas SQL e

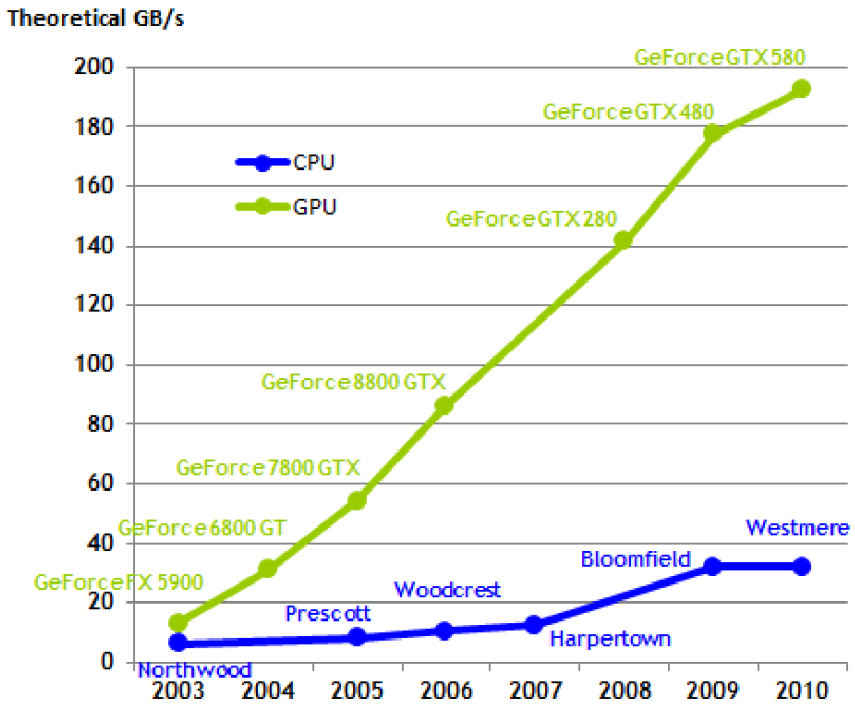


Figura 12: Largura de banda de memória (NVIDIA, 2011b).

reconstrução de ressonância magnética (NVIDIA, 2009).

Ao mesmo tempo em que o modelo GPGPU demostrava um grande avanço, apresentava vários inconvenientes: (*i*) era necessário que o programador possuísse um grande conhecimento tanto das APIs gráficas quanto da arquitetura da GPU; (*ii*) como o foco do processamento das GPUs é o processamento gráfico, os problemas deveriam ser descritos na forma de vértices e texturas, o que aumenta bastante a complexidade da descrição do problema; (*iii*) operações básicas em muitos programas, como leituras e gravações aleatórias na memória, não eram suportadas, em grande parte restringindo o modelo de programação; e (*iv*) a falta de suporte a precisão dupla (até recentemente) implicava na impossibilidade de usar processamento em GPUs para algumas simulações científicas relevantes.

Para resolver esses problemas, a NVIDIA introduziu, em novembro de 2006, duas importantes tecnologias. Uma delas foi a arquitetura G80 (introduzida a partir das placas GeForce 8800, Quadro FX 5600 e Tesla C870) que unificou o processamento de *pixels* e vértices. A outra tecnologia é CUDA (*Compute Unified Device Architecture*), um conjunto de *software* e arquitetura de *hardware* que permitiu a programação em GPU através de uma pequena extensão em uma variedade de linguagens de alto nível. Com isso o programador pode, ou invés de programar dedicadamente em um modelo gráfico, escrever, por

exemplo, programas em C com extensões em CUDA (C+CUDA) e utilizar o processador massivamente paralelo, a GPU (NVIDIA, 2009).

3.2 A arquitetura CUDA

O caminho de evolução seguido pelas GPUs tornaram esses *hardwares* naturalmente especializados em processamento de dados paralelos. A arquitetura das GPU foi concebida de maneira que mais transistores sejam dedicados ao processamento de dados, reduzindo o espaço para memória *cache* e controle de fluxo. Por outro lado, a arquitetura da CPU apresenta uma maior área do chip dedicada a memória *cache* e mecanismos de controle de fluxo sofisticados (NVIDIA, 2011b). A Figura 13 apresenta de forma esquemática a quantidade de transistores dedicada à memória e ao processamento. As porções em amarelo correspondem as unidades de controle, as porções em verde ao processamento e as porções em laranja correspondem a áreas de memória. Destacamos o uso da metade da área do *chip* da CPU para memória, enquanto que na GPU a maior área é dedicada ao processamento.

Figura 13: GPU dedica mais transistores ao processamento de dados (NVIDIA, 2011b).

3.2.1 Visão geral da arquitetura Fermi

A arquitetura CUDA é baseada em uma matriz de processadores escaláveis, ou seja, um conjunto de processadores, denominados *cores* CUDA, que executam várias *threads* simultaneamente.

Fermi é a terceira geração de GPUs CUDA desenvolvida pela NVIDIA. A primeira GPU baseada na arquitetura Fermi possui mais de 3 bilhões de transistores, com capacidade de ter até 512 *cores* CUDA. Um *core* CUDA executa uma operação inteira ou

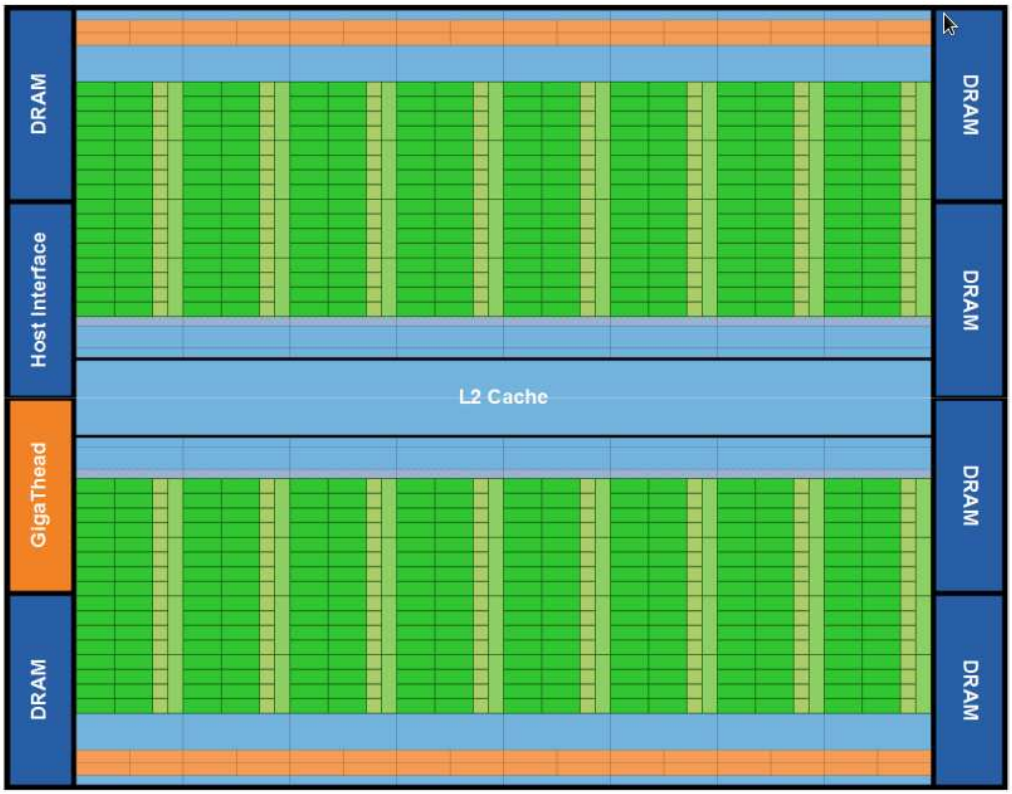


Figura 14: Visão geral da arquitetura Fermi (NVIDIA, 2009).

de ponto flutuante por *clock* para uma *thread* (NVIDIA, 2009). Os 512 *cores* CUDA, ou *streaming processors* (SP), são organizados em 16 *streaming multiprocessor* (SM) de 32 *cores* cada, Figura 14. A GPU possui seis bancos de memória de 64-bit e suporta até 6 GB de memória DRAM GDDR5. A Figura 14 apresenta uma visão geral da arquitetura Fermi, onde os 16 SM são posicionados em torno de uma memória *cache* L2. Cada SM é um retângulo vertical que possui um conjunto de unidades de execução (porção verde), registradores e memória *cache* L1 (porção azul clara) e escalonador de *threads* (*scheduler* e *dispatch* – porção laranja). A *host interface* conecta a CPU com a GPU através do barramento PCI-Express. O *GigaThread* é o responsável pelo escalonamento e distribuição dos blocos de *threads* para os SM.

A Figura 15 mostra uma visão geral da terceira geração de SM. Cada SM é formado por 32 *cores* CUDA, cada um deles possui um *pipeline* lógico e aritmético (ALU) e um de ponto flutuante (FPU). Na arquitetura Fermi é implementado o novo padrão de ponto flutuante IEEE–754 2008. A Fermi suporta operações de 32-bit para todas as instruções. A ALU também foi otimizada para suportar de forma eficiente 64-bit e operações de precisão estendida. Possui ainda suporte para vários tipos de operações, como: *boolean*, *shift*, *move*, *compare*, *convert*, *bit-field extract*, *bit-reverse insert* e *population count* (NVIDIA, 2009).

Cada SM possui 16 unidades de leitura/escrita (*load/store*), permitindo que dezesseis *threads* possam calcular os endereços de origem e destino a cada *clock*. Estes unidades suportam leitura e escrita da memória *cache* ou DRAM. As quatro unidades de funções especiais (SFU – Special Function Unit) executam funções transcendentais como seno, cosseno, função inversa (*reciprocal*) e raiz quadrada. Cada SFU executa uma *thread* por *clock*, assim uma *warp* é executado em 8 *clocks*.

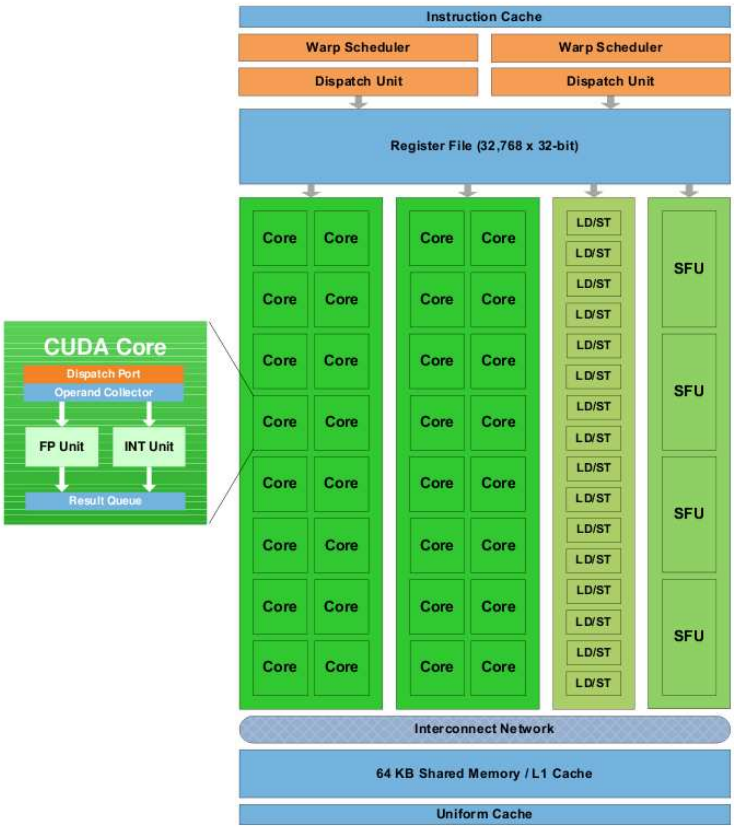


Figura 15: Visão geral do *streaming multiprocessor* Fermi (NVIDIA, 2009).

A arquitetura Fermi foi especificamente desenvolvida para oferecer um desempenho em precisão dupla sem precedentes (NVIDIA, 2009). Executa até 16 operações de precisão dupla por SM por *clock*, ou seja, é como se utilizasse dois *cores* CUDA em conjunto para a execução de uma operação de precisão dupla. Já para os demais tipos de dados (como *float*, *int* e *char*), executa 32 operações por SM por *clock*, ou seja, cada *core* executa uma operações.

As *threads* são executadas em grupos de 32 *threads*, chamados de *warp*. A execução dos *warps* promovem desempenho uma vez que se beneficiam do acesso a endereços de memória próximos e da minimização da latência de memória por meio do escalonamento entre eles. A execução e o controle dos *warps* é feita pela própria arquitetura, sendo assim

os programadores podem ignorar a execução do *warp* e programar pensando em uma *thread*. Destacamos que, é aconselhável o programador considerar o número de *threads* por bloco passado para o kernel múltiplo de 32 para que haja um melhor balanceamento de carga das *threads*, ou seja, para garantir que os *warps* estejam completos para a execução.

Cada SM possui dois escalonadores de *warp* (*warp schedulers*) e duas unidades de despacho de instrução (*Instruction Dispatch Unit*), permitindo assim que dois *warps* sejam selecionados e executados concorrentemente, como ilustrado na Figura 16. O escalonador duplo de *warp* seleciona dois *warps* e envia uma instrução de cada *warp* para um grupo de 16 *cores*, 16 unidades de leitura/escrita ou 4 SFUs. Muitas instruções podem ser despachadas aos pares (*dual dispatch*) e executadas concorrentemente, como duas instruções inteiras, duas instruções *float* ou uma combinação de instrução inteira, *float*, leitura, escrita e instruções especiais. No entanto, as instruções de precisão dupla não suportam *dual dispatch* com nenhuma outra operação (NVIDIA, 2009).

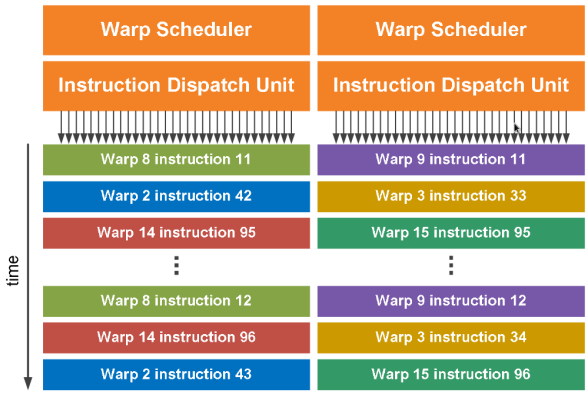


Figura 16: Execução concorrente de dois *warps* (NVIDIA, 2009).

A memória *shared* permite a cooperação entre as *threads* de um bloco, facilitando o intenso reuso de dados e a diminuição do tráfego de dados. As gerações anteriores (G80 e GT200) possuíam 16KB de memória *shared* por SM. A arquitetura Fermi possui 64KB de memória interna ao chip que pode ser configurada com 48KB de memória *shared* e 16MB de memória *cache* L1 (padrão) ou com 16KB de memória *shared* e 48MB de memória *cache* L1. A Fermi também possui 768KB de memória *cache* L2, que é mais um benefício para esta geração, dado que as anteriores não possuíam memórias *cache* L1 e L2 (NVIDIA, 2009).

A Fermi ainda permite, por meio da nova tecnologia GigaThread, a execução concorrente de até 16 *kernels*.

3.2.2 Hierarquias de *thread* e de memória

No modelo de programação CUDA os SM utilizam uma arquitetura de processamento denominada *single-instruction, multiple-thread* (SIMT). Estas SMs SIMT criam, gerenciam, escalonam e executam *threads* em grupos de 32 *threads* paralelas, denominados *warps*. Assim, a mesma instrução será executada por todas as *threads*.

Um programa em CUDA é executado a partir da chamada de um *kernel* paralelo. Um *kernel* executa, em paralelo, um conjunto de *threads*. As *threads* são organizadas em blocos de *threads* e, por sua vez, os blocos de *threads* são organizados em grades (grids), como apresentado na Figura 17. Cada bloco de *threads* é executado por um SM. A GPU instancia um *kernel* em um grid de blocos de *threads*.

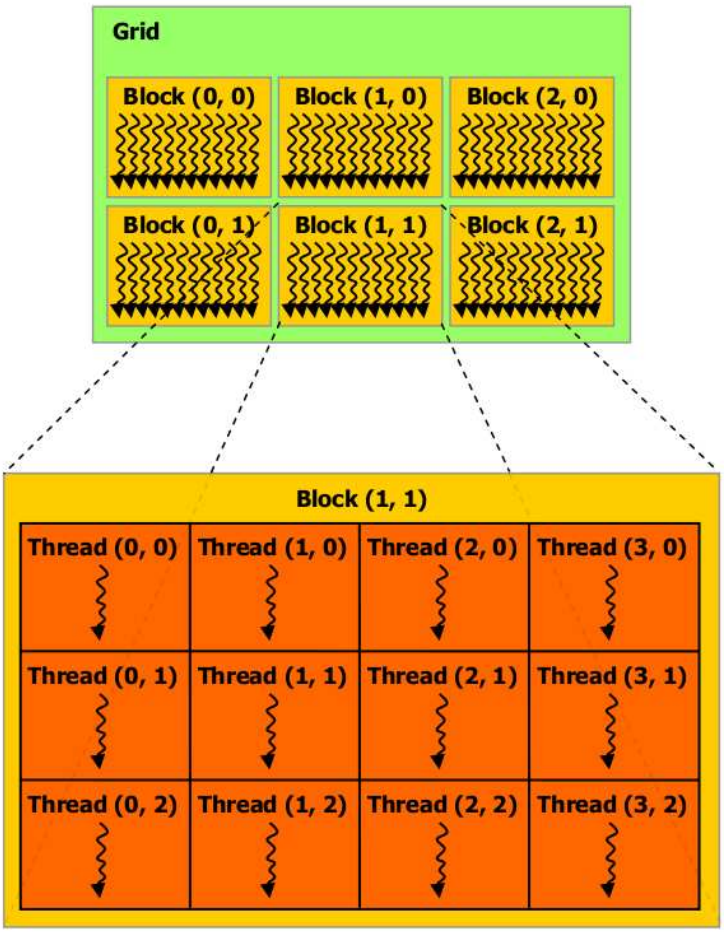


Figura 17: Grid de blocos de *threads* (NVIDIA, 2011b).

Cada *thread* de um bloco possui um valor de identificador (`threadIdx`), um contador de programa, registradores e memória privada. As *threads* concorrentes de um mesmo bloco podem cooperar entre si através de barreiras de sincronismo e por meio de uma

pequena memória compartilhada (*shared*), desta forma o sincronismo só é garantido para as *threads* de um mesmo bloco. Cada bloco de *threads* também possui um identificador (`blockIdx`) próprio. A grid, que é um conjunto de blocos de *thread* que executam em um *kernel*, lê e escreve na memória global. Por meio dos identificadores é possível identificar cada *thread* dentro de um grid. O sincronismo de todas as *threads* é garantido após a execução do kernel.

A hierarquia de memória CUDA é composta por: registradores, memória compartilhada (*shared memory*), memória de constantes (*constant memory*), memória de textura (*texture memory*) e memória global da placa (*device memory*). As *threads* CUDA tem acesso a múltiplos espaços de memória, dependendo da hierarquia de *thread*, durante sua execução, como ilustrado na Figura 18. Nesta figura vemos que uma *thread* individualmente possui seu espaço de memória privado e seus próprios registradores. Um bloco de *threads* tem acesso a uma memória compartilhada (*shared*) e o grid, ou seja, todas as *threads*, tem acesso à memória global.

A memória global, ou memória da placa, possui a maior capacidade e a maior latência. Esta memória aceita operações de leitura e escrita provenientes tanto da CPU quanto da GPU, viabilizando a troca de dados entre a CPU e a GPU. Mesmo tendo a maior capacidade, um código C+CUDA de considerar a quantidade de memória que será empregada em sua execução, ou seja, um *kernel* a ser executado não deve extrapolar a quantidade de memória da placa, uma vez que não há recursos de memória virtual, como os arquivos de *swap* utilizados na troca de dados entre a memória RAM e o disco rígido (HD). Desta forma, não é possível utilizar mais memória do que a fisicamente existente. Assim, caso haja a necessidade de executar um conjunto de dados maior que a memória disponível, a distribuição desses dados deve ser feita pelo programador.

A memória *shared* é uma memória com menor latência, porém possui um tamanho limitado. É útil para troca de dados entre *threads* de um mesmo bloco em algoritmos paralelos.

Os registradores possuem a menor latência de memória e armazenam as variáveis de uma *thread*. Cada *thread* possui um número de registradores que é determinado de acordo com o número de *threads* disparadas em um *kernel*. Sendo assim, deve-se levar em consideração o número de variáveis do código bem como o número de threads disparados pois, um menor número de registradores implica na necessidade de troca de valores das variáveis da memória global para os registradores.

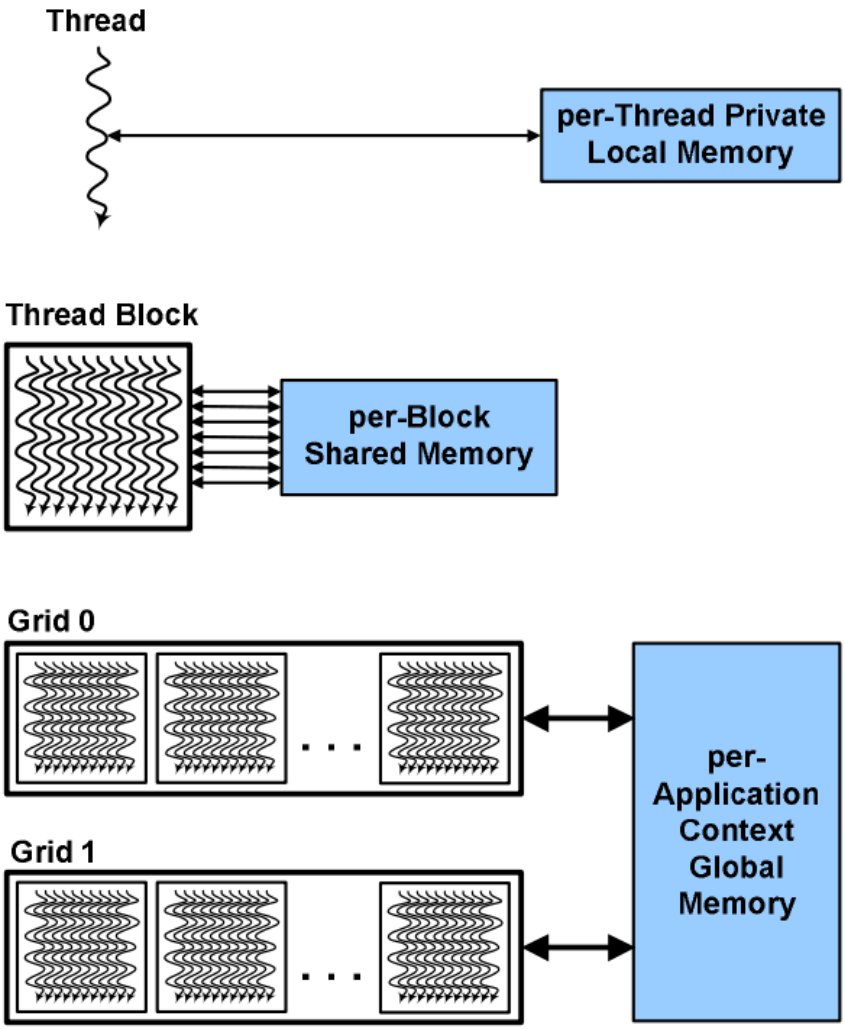


Figura 18: Modelo da Hierarquia de memória CUDA (NVIDIA, 2009).

3.3 Programação em C+CUDA

Na construção de um código C+CUDA é necessário o conhecimento da arquitetura utilizada para um melhor aproveitamento dos recursos computacionais. O sistema de programação para um programador CUDA consiste no *host*, que é a tradicional unidade central de processamento (CPU), e de um ou vários *devices* (GPU). Assim, as GPUs são utilizadas como coprocessadores capazes de executar milhares de *threads* simultaneamente. Com isso, os trechos com grandes demandas computacionais podem ser traduzidos para CUDA e executados na GPU. Este modelo de programação também assume espaços de memória distintos mantidos pelo *host* e pelo *device*, sendo chamados respectivamente de memória do *host*, a memória DRAM do computador, e memória do *device*, a memória DRAM da placa gráfica. O que implica em alocação e desalocação de memória na GPU, bem como a transferência de dados entre as memórias do *host* e do *device*.

A programação em C+CUDA é feita através de uma pequena extensão da linguagem C e por uma nova biblioteca C. Na tradução, o programador tipicamente reescreve sua versão sequencial na forma de *kernels* paralelos. A Figura 19 ilustra a execução de um código C+CUDA onde a parte serial é executada no *host* e a parte paralela é executada no *device*. O algoritmo C+CUDA, consiste em um fluxo de controle rodando em uma CPU, operações de transferência de dados entre as memórias da CPU e da GPU e chamadas de *kernel* que executarão as tarefas utilizando o paralelismo da GPU.

Como já mencionado anteriormente, um *kernel* comanda a execução, na GPU, de um conjunto de *threads* que por sua vez são organizados em grids de blocos de *threads*. Um grid é um conjunto de blocos de *threads* que executam independentemente, enquanto que um bloco de *threads* é um conjunto de *threads* que podem cooperar por meio de sincronização do tipo barreira e pelo acesso compartilhado a um espaço de memória exclusivo de cada bloco de *thread* (memória *shared*).

O legado da computação gráfica influenciou a hierarquia de *threads* de forma que seus elementos (grids, blocos e *threads*) podem ser implementados em mais de uma dimensão. As grids podem ter uma ou duas dimensões, já os blocos de *thread* podem ter até três dimensões.

Em C+CUDA existem três tipos de função:

Host : funções que são chamadas e executadas somente pela CPU. São Semelhantes as funções implementadas em C.

Kernel : funções que são chamadas pela CPU e são executadas somente pela GPU. Possuem um qualificador (`__global__`) que deve ser inserido antes da definição do tipo de retorno da função, que deve obrigatoriamente ser do tipo `void`. A sintaxe da chamada desta função é: (`função <<< Num_Blocks, Num_Threads, Num_Mem>>> (parâmetros)`) onde o número de blocos (`Num_Blocks`) e o número de *threads* (`Num_Threads`) são explicitados. `Num_Mem` é reservado para os casos em que se deseja especificar dinamicamente a quantidade de memória *shared* a ser usada em cada bloco.

Device : funções chamadas e executas somente pela GPU. Possuem um qualificador (`__device__`) que deve ser declarado antes da definição do tipo de retorno da função, neste caso, é permitido que a função retorne qualquer tipo de dado.

O programador pode declarar uma grid 1D ou ema grid 2D. O tamanho do grid em

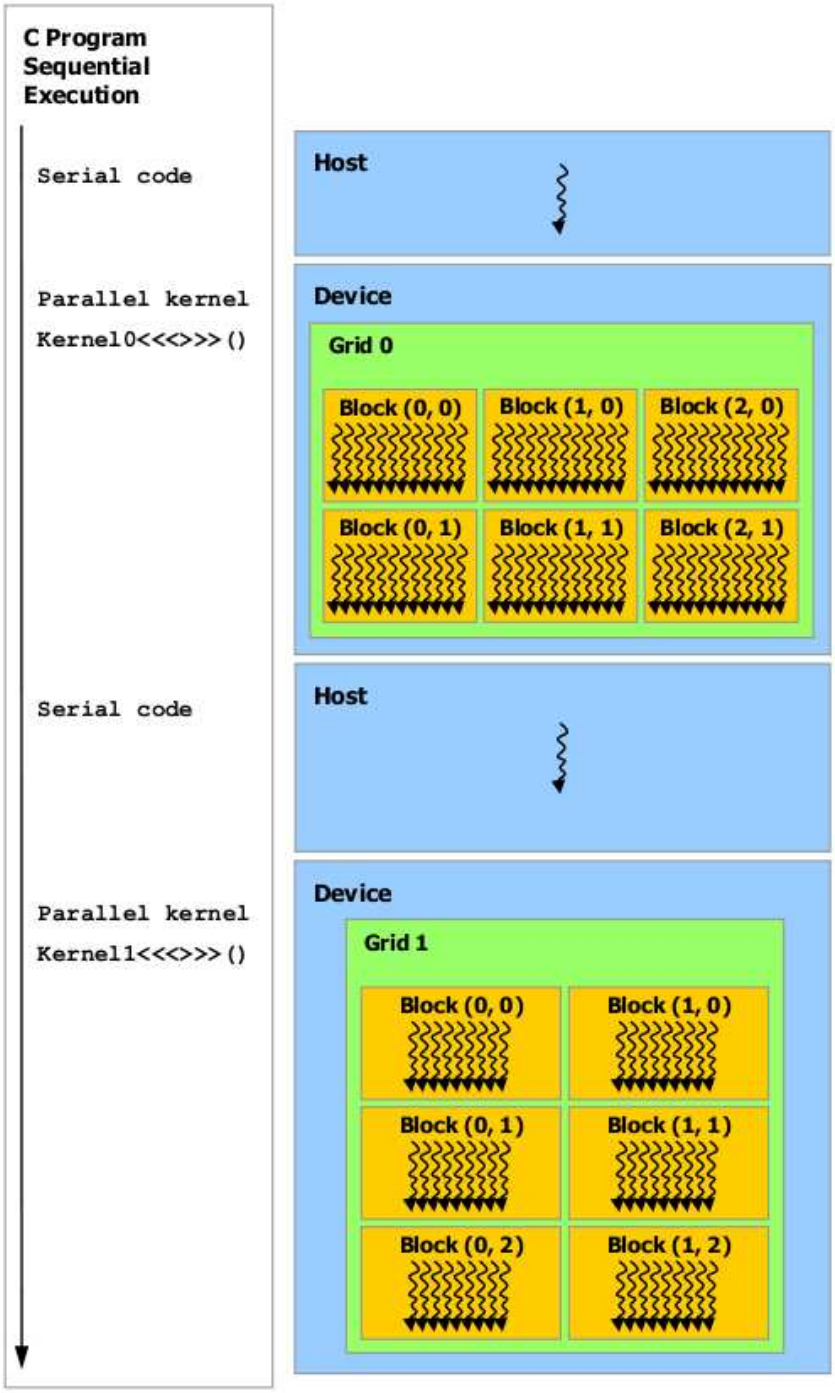


Figura 19: Execução do código serial no *host* e do código paralelo no *device* (NVIDIA, 2011b).

cada uma das dimensões pode ser especificado através de diferentes valores inteiros ou através de uma estrutura de dados (`dim3`) com os respectivos campos de *Num_Blocks.x* e *Num_Blocks.y*. A definição do tamanho dos blocos (`Num_Threads`) é feita de forma semelhante, ou seja, em até 3 dimensões. Os identificadores `blockIdx` e `threadIdx` também informam os valores relativos às referidas dimensões de bloco e *threads* (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`, `threadIdx.x`, `threadIdx.y` e `threadIdx.z`).

3.3.1 O algoritmo paralelo

Neste trabalho executamos os seguintes passos para se obter o código CUDA: (*i*) otimização do código sequencial para CPU, com o objetivo de obter um bom código sequencial que sirva de base para as comparações de *speedup*, neste trabalho utilizamos otimizações do compilador, conforme descrito na seção 4; (*ii*) uso da ferramenta de *profile* `gprof` (GRAHAM *et al.*, 2004) para determinar as partes mais custosas do código em termos de tempo de execução; (*iii*) desenvolvimento de um código paralelo CUDA procurando obter um código com o melhor desempenho possível; e (*iv*) uma versão CUDA usando memória *shared*. Ressaltamos que a otimização do código sequencial é um importante passo para para a obtenção de um código paralelo CUDA eficiente.

Os passos do algoritmo principal em C+CUDA implementado no *host* são similares aos passos apresentados para o Algoritmo 2.1. Todas as funções internas ao *loop* no tempo foram executadas na GPU. A ideia principal é evitar cópias de dados entre CPU e GPU a cada passo no tempo, já que é uma operação custosa em um programa C+CUDA. Sendo assim, para que a execução das funções seja feita na GPU é necessário copiar da CPU para a GPU os vetores inicializados, definir as funções que serão executadas na GPU (kernel) juntamente com valores apropriados para o número de *threads* por bloco (`NUM_THREADS`) e o número de blocos por grid (`NUM_BLOCKS`) e finalmente copiar os resultados obtidos na GPU para a CPU.

O *loop* no tempo para o código C+CUDA está mostrado no Algoritmo 3.1. A escolha para o número de blocos em geral deve considerar: (i) o número de registradores que serão utilizados, ou seja, o número de variáveis declaradas; (ii) o número de *threads* para cada bloco, pois cada *thread* terá seus próprios registradores; e (iii) o número de SM disponível na placa utilizada.

As funções `comp_delt ( ... )` e `Red_Black_SOR_GPU ( ... )` não são *kernels* por opção de implementação, porém internamente chamam *kernels* paralelos, tais funções serão discutidas posteriormente.

Para facilitar a implementação em CUDA, implementamos todos os vetores ou matrizes de forma linearizada e seguindo a numeração apresentada na Seção 2.4. Utilizamos também o prefixo `d_` como notação para a nomeação das variáveis do *device*, facilitando a diferenciação das variáveis do *host* e do *device* na leitura dos códigos.

Algoritmo 3.1: *Loop* no tempo do algoritmo principal no *host*.

```
1  ...
2  while( t < t_end){
3      comp_delt ( ... );
4      set_BondaryCond_GPU      <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
5      set_InputCondition_GPU   <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
6      comp_T_GPU               <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
7      comp_FGH_GPU             <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
8      poisson_system_GPU       <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
9      // Resolucao do sistema linear
10     Red_Black_SOR_GPU ( ... );
11     total_time += time;
12     comp_UV_GPU              <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
13     t += delt;
14 }
15 ...
```

3.3.1.1 Principais comandos CUDA

Apresentaremos em seguida alguns comandos básicos em CUDA que são essenciais para a tradução do código sequencial para CUDA, bem como para o entendimento dos algoritmos apresentados. Para uma lista mais completa e detalhada veja o manual de referência CUDA, (NVIDIA, 2011a).

`__global__ function ( ... )` : definição de um *kernel* CUDA, deve ser inserido antes da definição do tipo de retorno da função, que obrigatoriamente é `void`;

`__device__ function ( ... )` : definição de uma função *device* CUDA, deve ser inserido antes da definição do tipo de retorno da função;

`function <<< NUM_BLOCKS, NUM_THREADS >>> (...)` : chamada de um *kernel* CUDA, onde `NUM_BLOCKS` indica o número de blocos (tamanho do grid) e `NUM_THREADS` indica o número de *threads* por bloco (tamanho do bloco);

`cudaMalloc(void ** d_Ptr, size_t size)` : aloca um vetor na memória de tamanho igual a `size` bytes e retorna um ponteiro (`*d_Ptr`) para a memória alocada. A memória alocada é devidamente alinhada para qualquer tipo de variável. Durante a alocação, a memória não é inicializada;

`cudaFree(void * d_Ptr)` : libera o espaço de memória apontado por `d_Ptr`;

`cudaMemcpy(void * dst, const void * src, size_t size, enum cudaMemcpyKind kind)` : copia `size` bytes da área de memória apontada por `src` para a área de memória apontada por `dst`. *kind* especifica a direção da cópia de memória que pode ser: `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost` e `cudaMemcpyDeviceToDevice`. As áreas de memória não podem se sobrepor.

`__syncthreads ()` : função que implementa uma sincronização do tipo barreira entre as *threads* de um bloco;

`__shared__` : identificador inserido antes do tipo de uma variável para que esta seja mapeada na memória *shared*.

3.3.1.2 Função `comp_delt`

A função `comp_delt` é responsável por calcular de forma adaptativa em cada passo no tempo o valor do passo. Uma visão geral desta função é apresentada no Algoritmo 3.2. O

Algoritmo 3.2: Função `comp_delt`.

```
1  void comp_delt_GPU ( ... ) {
2      ...
3      // definicao de variaveis
4      ...
5      // reducao do maximo para cada uma componente da velocidade
6      reductionArvBinMax <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
7      reductionArvBinMax <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
8      reductionArvBinMax <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
9      // copia de memoria, GPU para CPU
10     cudaMemcpy (u_parc, d_u_parc, NUM_BLOCKS*sizeof(REAL),
           cudaMemcpyDeviceToHost);
11     cudaMemcpy (v_parc, d_v_parc, NUM_BLOCKS*sizeof(REAL),
           cudaMemcpyDeviceToHost);
12     cudaMemcpy (w_parc, d_w_parc, NUM_BLOCKS*sizeof(REAL),
           cudaMemcpyDeviceToHost);
13     // seleciona o maximo global para cada componente
14     for (i = 0; i < NUM_BLOCKS; i++) {
15         if (u_max < u_parc[i]) u_max = u_parc[i];
16         if (v_max < v_parc[i]) v_max = v_parc[i];
17         if (w_max < w_parc[i]) w_max = w_parc[i];
18     }
19     ...
20     // calcula o valor de delt segundo a Equacao 2.53
21 }
```

gargalo da função `comp_delt` é a busca pelo maior valor das componentes de velocidade (u, v e w). Uma forma eficiente e facilmente paralelizável para se encontrar tal valor é fazendo uma redução em arvore binária (linhas 6, 7 e 8). Esta redução será discutida a seguir. A função `reductionArvBinMax` fornece como saída um pequeno vetor com o valor máximo encontrado por cada bloco, este vetor deve ser analisado pelo *host*, ou seja, copiado para o *host*, e finalizar sua execução na CPU. Como o tamanho do vetor é pequeno, a finalização da redução na CPU, bem como a cópia, não se torna um ponto crítico para o desempenho.

3.3.1.3 Função *kernel* `RedutionArvBinMax`

Como já mencionado, a redução em árvore binária é uma forma usual e facilmente paralelizável em C+CUDA para se encontrar o maior valor em um vetor. Para maior entendimento da redução em árvore binária, suponha que queremos encontrar o maior valor em um vetor com 8 posições. Primeiramente dividimos os elementos em 2 grupos, conforme ilustrado na Figura 20. Os elementos das posições 0 e 4 são analisados e o maior é armazenado na posição 0. Em seguida os elementos 1 e 5 são analisados e o maior é armazenado na posição 1. De forma semelhante para os elementos 2 e 6, sendo o maior armazenado na posição 2 e por fim são analisados os elementos 3 e 7 e o maior é armazenado na posição 3. No segundo passo o vetor é reduzido para o tamanho 4 (metade do tamanho) e novamente dividido em dois. Assim, são analisados os elementos 0 e 2 e o maior é armazenado na posição 0. De forma semelhante são analisados os elementos 1 e 3 e o maior é armazenado na posição 1. Finalmente o vetor é reduzido para o tamanho 2 (metade do tamanho), bastando apenas analisar os últimos dois elementos e armazenar o maior valor na posição 0.

O Algoritmo 3.3, *kernel* `RedutionArvBinMax`, apresenta uma redução em árvore binária com o uso da memória *shared*. O uso da memória *shared*, neste caso, minimiza a necessidade de escrita de valores na memória global do *device*. Os valores passados por parâmetro neste *kernel* são: (i) o tamanho do vetor (`N`); (ii) o vetor com os valores a serem avaliados (`d_vet`); (iii) um vetor auxiliar (`d_parc`) do tamanho do número de blocos, que receberá resultados da redução de cada bloco; e (iv) um vetor de marcação (`d_flag`) que informa o tipo de célula que será executada, que pode ser: contorno, obstáculo ou fluido. A variável `tid` (linha 5) recebe inicialmente o valor do identificador global da *thread* dentro da grid. Caso o tamanho da malha seja maior que o número de *threads* disparadas pelo *kernel*, cada *thread* deverá operar em mais de uma posição da malha, ou seja, se n *threads*

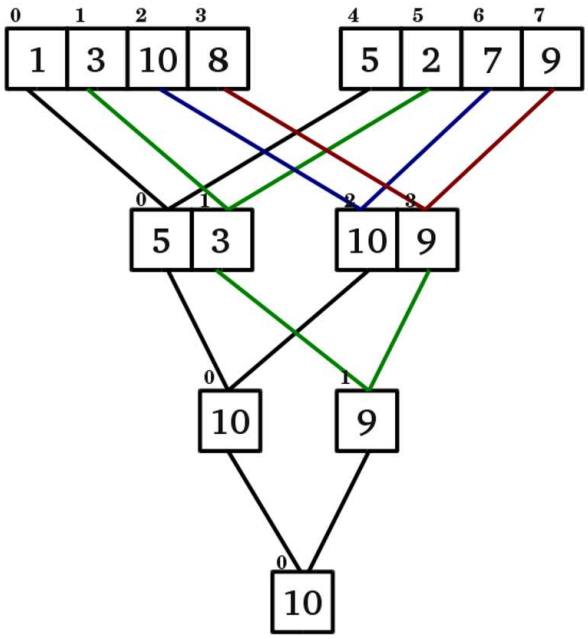


Figura 20: Exemplo de redução em árvore binária.

devem operar sobre uma malha com $3n$ posições, cada *thread* executará em 3 posições, onde a primeira *thread* computará nas posições 0, n e $2n$; a segunda *thread* computará sobre as posições 1, $n+1$ e $2n+1$ e assim sucessivamente. O incremento das posições, ou seja, o salto da *thread*, é calculado na variável `inc` (linha 6), que recebe o número de *threads* total disparadas pelo *kernel*, e um *loop* (linha 10) garante a execução de todas as posições. Cada *thread* armazenará em um vetor *cache* compartilhado o maior valor encontrado por ela (linha 19). Em seguida é realizado uma redução em árvore binária no vetor `cache` (linhas 24 a 31); observe que antes de cada iteração no *loop* de redução é necessário a sincronização das *threads* (linhas 21 e 30). Em seguida, uma única *thread* de cada bloco (linha 32) escreve o resultado de sua redução em um vetor denominado `d_parc` na memória global do dispositivo (linha 33). Para finalizar, é fornecido como saída o vetor `d_parc`, que possui dimensões reduzidas, quando comparada com o vetor original, e valores máximos distribuídos por blocos. Este vetor é copiado no *host* e a escolha do máximo é finalizada na CPU. Como o número de blocos é pequeno, a finalização da redução na CPU não se torna um ponto crítico para o desempenho.

Algoritmo 3.3: Algoritmo de redução do máximo.

```
__global__
void reductionArvBinMax (int N, REAL *d_vet, REAL *d_parc, int* d_flag)
{
    __shared__ float cache[NUM_THREADS];
    int tid   = threadIdx.x + blockIdx.x*blockDim.x;
    int inc   = blockDim.x * gridDim.x;
    REAL temp = 0.0;
    REAL aux;
    int i, k;
    for (i = tid; i < N; i += inc) {
        if ( d_flag[i] == C_F ) {
            aux = d_vet[i];
            if (temp < aux) {
                temp = aux;
            }
        }
    }
    // setando os valores na variavel cache
    cache[threadIdx.x] = temp;
    // sincronizando as threads neste bloco
    __syncthreads();
    // para reduzir, o numero de threasd por bloco deve ser
    // uma potencia de 2 por causa do seguinte codigo
    for ( k = (blockDim.x >> 1); k > 0; k >>= 1) {
        if (threadIdx.x < k) {
            if (cache[threadIdx.x] < cache[threadIdx.x + k]) {
                cache[threadIdx.x] = cache[threadIdx.x + k];
            }
        }
        __syncthreads();
    }
    if (threadIdx.x == 0)
        d_parc[blockIdx.x] = cache[0];
}
```

3.3.1.4 Função `Red_Black_SOR_GPU`

A implementação do método iterativo Red Black SOR é apresentada no Algoritmo 3.4. Este método é implementado no *host* e chama *kernels* CUDA nos pontos onde há necessidade de maior poder de processamento. O algoritmo foi escrito de forma que sejam

realizadas 5 iterações do método Red_Black_SOR (linhas 6 a 9) antes que a norma do máximo (linhas 10 a 18) seja calculada. Esta escolha se deu pelo fato do cálculo da norma a cada iteração ser oneroso. O algoritmo prevê dois critérios de parada: (*i*) norma do máximo (linhas 10 a 18) e (*ii*) número máximo de iterações (linha 21).

Os *kernels* `red GPU` (linha 7), referente a Equação (2.65), e `black GPU` (linha 8), referente a Equação (2.66), realizam a execução das células vermelhas e pretas, respectivamente. A escolha da escrita de um *kernel* para cada tipo de célula garante que todas as células vermelhas terminem sua execução antes de começar a executar as células pretas (outro *kernel*), isto porque o sincronismo dentro de um *kernel* só é garantido para as *threads* de um mesmo bloco. A implementação destes *kernels* será discutida a seguir.

O *kernel* `reductionMax` (linha 10) realiza uma redução do máximo de forma similar ao apresentado no Algoritmo 3.3. O que diferencia uma função da outra são a quantidade de parâmetros passados e os resultados que serão retornados por meio dos vetores parciais. Os parâmetros da função `reductionMax` são: (*i*) o tamanho dos vetores; (*ii*) o vetor com os valores atualizados (`d_p`); (*iii*) o vetor com os valores anteriores (`d_ant`); (*iv*) um vetor parcial (`d_parc_diff`) que receberá o maior valor, por bloco, da diferença entre os vetores `d_p` e `d_ant`; e (*v*) um vetor parcial (`d_parc_max`) com o maior valor, de cada bloco, encontrado para o vetor `d_p`. Os vetores parciais devem se finalizados no *host*, sendo assim, são copiados para a memória do *host* (linhas 12 e 13) e finalizam sua execução na CPU (linhas 14 a 19). Ao final da execução da função é retornado o número de iterações realizadas.

Algoritmo 3.4: Método Red_Black_SOR no *host*.

```
1  int Red_Black_SOR_GPU ( ... ) {
2      int i;
3      int Iter = 0;
4      REAL Diff, Max;
5      do {
6          for (i = 0; i < 5; i++) {
7              red_GPU   <<< NUM_BLOCKS, NUM_THREADS >>>   ( ... );
8              black_GPU <<< NUM_BLOCKS, NUM_THREADS >>>   ( ... );
9          }
10         reductionMax <<< NUM_BLOCKS, NUM_THREADS >>> (imax*jmax*kmax, d_p,
               d_ant, d_parc_diff, d_parc_max);
11         cudaMemcpy (parc_diff, d_parc_diff, NUM_BLOCKS*sizeof(REAL),
               cudaMemcpyDeviceToHost);
12         cudaMemcpy (parc_max, d_parc_max, NUM_BLOCKS*sizeof(REAL),
               cudaMemcpyDeviceToHost);
13         DiffMax = 0.0;
14         Max     = 0.0;
15         for (i = 0; i < NUM_BLOCKS; i++) {
16             if (Dif < parc_diff[i]) Diff = parc_diff[i];
17             if (Max < parc_max[i] ) Max  = parc_max[i];
18         }
19         Iter = Iter+5;
20     }
21     while ( Iter <= IterMax && DiffMax > Toler*Max);
22     return Iter;
23 }
```

3.3.1.5 Função *kernel* `red_GPU`

O Algoritmo 3.5 mostra a implementação do *kernel* das células vermelhas (`red_GPU`). O *kernel* relativo às células pretas (`black_GPU`) é similar ao apresentado no Algoritmo 3.5, se diferenciando apenas no resultado do cálculo do tipo de célula, sendo assim, será omitido.

Os parâmetros do *kernel* são, respectivamente, as dimensões nos eixos x (`imax`), y (`jmax`), z (`kmax`); o parâmetro de relaxação do método SOR (`omg`); as diagonais com os coeficientes do sistema linear (`d_diag_d`, `d_diag_w`, `d_diag_b`, `d_diag_p`, `d_diag_f`, `d_diag_e` e `d_diag_u`); o vetor com o lado direito do sistema linear (`d_rhs`); um vetor para guardar o resultado anterior (`d_VetAux`) e o vetor resultado (`d_p`).

Nesta função, e também no *kernel* `black_GPU`, utilizamos uma estratégia diferente da

apresentada anteriormente para o controle das *threads*. Esta mudança foi necessária para uma melhor utilização e controle da memória *shared* que será apresentado a seguir.

A ideia utilizada é que cada bloco execute em apenas uma linha (direção x) da matriz. O *loop* externo (linha 10) controla a iteração dos blocos nas linhas, já o *loop* interno (linha 17) controla as iterações das *threads* de um bloco em uma linha. A variável `first` guarda o índice para o início das linhas da matriz `d_p`, enquanto a variável `last` guarda o índice do final das linhas de `d_p`. Caso o tamanho da linha (dimensão x) seja maior que o número

Algoritmo 3.5: *Kernel* para as células vermelhas na GPU.

```
1  __global__
2  void red_GPU (int imax, int jmax, int kmax, REAL omg, REAL* d_diag_d, REAL*
       d_diag_w, REAL* d_diag_b, REAL* d_diag_p, REAL* d_diag_f, REAL*
      d_diag_e, REAL* d_diag_u, REAL* d_rhs, REAL* d_VetAux, REAL* d_p) {
3      int first;
4      int last;
5      int linha;
6      int num_Linhas = jmax*kmax;
7      int i, j, k;
8      int pos;
9      REAL aux;
10     for (linha = blockIdx.x; linha <  num_Linhas; linha += gridDim.x) {
11         first = linha*imax;
12         last  = first+imax;
13         i = 2*threadIdx.x;
14         j = linha / jmax;
15         k = linha % jmax;
16         pos = ( (k % 2) == 0 ) ? ( ((j % 2) == 0) ? (i+1) : (i) ) : ( ((j %
               2) == 0 ) ? (i) : (i+1) );
17         for (pos += first; pos < last; pos += blockDim.x) {
18             aux = ( d_diag_d[pos] *  d_p[pos - imax*jmax]
19                 +   d_diag_w[pos] *  d_p[pos - imax      ]
20                 +   d_diag_b[pos] *  d_p[pos - 1         ]
21                 +   d_diag_f[pos] *  d_p[pos + 1         ]
22                 +   d_diag_e[pos] *  d_p[pos + imax      ]
23                 +   d_diag_u[pos] *  d_p[pos + imax*jmax] );
24             d_VetAux[pos] = d_p[pos];
25             d_p[pos]      = omg*( d_rhs[pos] - aux ) / d_diag_p[pos] + (1-
                   omg)*d_p[pos];
26         }
27     }
28 }
```

de *threads* por bloco, as *threads* deverão executar em mais de uma posição na linha até que todas as posições da linha sejam executadas. O incremento das *threads* em uma linha é calculado pelo tamanho do bloco (`blockDim.x` – linha 17). De forma similar, caso o número de linhas seja maior que o número de blocos, os blocos deverão executar em mais de uma linha até que todas as linhas sejam executadas. O incremento do bloco é feito de acordo com o tamanho do grid (`gridDim.x` – linha 10).

As variáveis i, j e k guardam os índices de uma dada posição na malha referentes as dimensões x, y e z, respectivamente (linhas 13, 14 e 15). O índice k também representa o plano xy a que a célula pertence. Como a malha está separada entre células vermelhas e células pretas, cabe a cada *thread* realizar o cálculo da posição na malha que deverá ser computada (linhas 13 a 16). O cálculo das posições das células vermelhas (`pos` – linha 16) utiliza dois critérios: um para verificar a paridade dos planos xy, e outro para verificar a paridade das linhas desses planos. Assim, $\texttt{pos} = i + 1$ se $\texttt{j} = \texttt{par}$, senão, $\texttt{pos} = i$, isto se o plano xy também for par, ou seja, se $k = par$. Se $k = impar$, a lógica para o valor da posição é invertida, garantindo assim o acesso apenas as células vermelhas, segundo a malha apresentada na Figura 10. O corpo da iteração do método apresentado é o mesmo para o método SOR sequencial (linhas 18 a 25).

3.3.1.6 Função *kernel* `red_GPU_shared` – Uso da memória *shared*

Normalmente o uso de uma memória com menor latência, como a memória *shared*, é o melhor caminho para se obter maior desempenho. Porém a memória *shared* é um recurso limitado, visto que seu tamanho máximo nas placas Fermi é de 64KB. Desta forma é preciso modelar o problema de forma que os dados reutilizáveis sejam armazenados nesta memória, reduzindo assim o acesso a memória global.

Para a utilização da memória *shared* no *kernel* `red_GPU_shared` foi feita a opção pelo uso de vetores *shared* de tamanho $\texttt{NUM\_THREAD} + 1$. Considerando a utilização de tipo de dados *double*, que utiliza 8 bytes para ser armazenado na memória, um bloco com 512 *threads* e 16KB de memória *shared*, podemos contabilizar o número máximo de vetores que a memória comporta da seguinte forma:

$$\begin{aligned} \text{Número de Vetores} &= \frac{\text{tamanho da memória } shared}{\text{tamanho do vetor}} = \frac{16\text{KB}}{\texttt{NUM\_THREADS} * 8\text{B}} \\ &= \frac{16\text{KB}}{512 * 8\text{B}} = \frac{2^{14}\text{B}}{2^{12}\text{B}} = 4 \text{ vetores} \end{aligned} \tag{3.1}$$

Nesta função utilizamos apenas um vetor na memória *shared* aproveitando o reuso de memória da seguinte forma: em uma mesma linha da malha, a posição W (oeste) de uma célula vermelha será a posição E (leste) da célula vermelha anterior, como ilustrado na Figura 21.

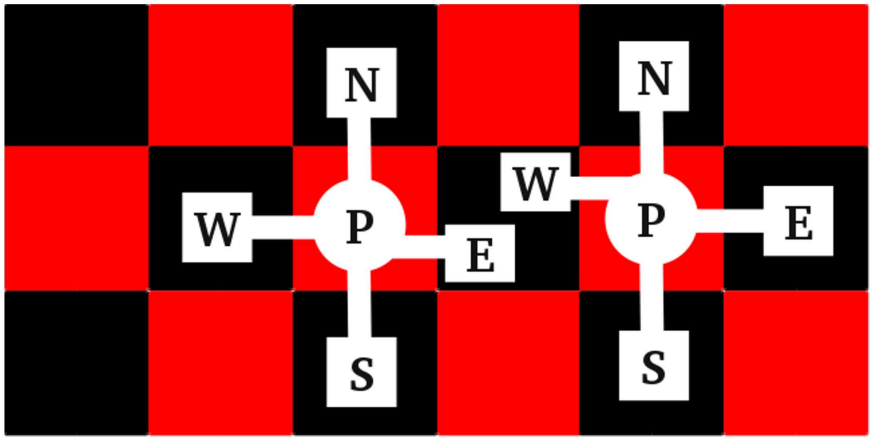


Figura 21: Exemplo de reuso de memória *shared*.

O Algoritmo 3.6 apresenta um *kernel* de células vermelhas utilizando memória *shared* (`red_GPU_shared`). As diferenças entre o Algoritmo 3.5 e o Algoritmo 3.6 são: (*i*) declaração do vetor na memória compartilhada (linha 10); (*ii*) inicialização do vetor compartilhado (linhas 19 e 20); sincronização das *threads* (linha 21); e (*iv*) utilização do vetor compartilhado no lugar de um vetor na memória global do dispositivo (linhas 24 e 25). Para a inicialização do vetor é necessário o acerto da última posição de cada linha, que é controlado pelo `if` da linha 20. Assim o vetor é preenchido em todas as posições com os valores correspondentes às células oeste (W), exceto a última posição, que corresponde a célula leste (E). Destacamos a necessidade de uma barreira de sincronismo logo após a inicialização da memória *shared* para garantir que uma *thread* só executará o método SOR quando o vetor *shared* estiver todo inicializado, evitando a utilização de dados incorretos.

Algoritmo 3.6: *Kernel* para as células vermelhas na GPU com memória *shared*.

```
1  __global__
2  void red_GPU (int imax, int jmax, int kmax, REAL omg, REAL* d_diag_d, REAL*
       d_diag_w, REAL* d_diag_b, REAL* d_diag_p, REAL* d_diag_f, REAL*
      d_diag_e, REAL* d_diag_u, REAL* d_rhs, REAL* d_VetAux, REAL* d_p) {
3      int first;
4      int last;
5      int linha;
6      int num_Linhas = jmax*kmax;
7      int i, j, k;
8      int pos;
9      REAL aux;
10     __shared__ REAL s_p[NUM_THREADS + 1];
11     for (linha = blockIdx.x; linha <  num_Linhas; linha += gridDim.x) {
12         first = linha*imax;
13         last  = first+imax;
14         i = 2*threadIdx.x;
15         j = linha / jmax;
16         k = linha % jmax;
17         pos = ( (k % 2) == 0 ) ? ( ((j % 2) == 0) ? (i+1) : (i) ) : ( ((j %
               2) == 0 ) ? (i) : (i+1) );
18         for (pos += first; pos < last; pos += blockDim.x) {
19             s_p[threadIdx.x] = d_p[pos-1];
20             if(threadIdx.x == (NUM_THREADS-1)) s_p[threadIdx.x+1] = d_p[pos
                   +1];
21             __syncthreads();
22             aux = ( d_diag_d[pos] *  d_p[pos - imax*jmax]
23                 +   d_diag_w[pos] *  d_p[pos - imax      ]
24                 +   d_diag_b[pos] *  s_p[threadIdx.x     ]
25                 +   d_diag_f[pos] *  s_p[threadIdx.x + 1]
26                 +   d_diag_e[pos] *  d_p[pos + imax      ]
27                 +   d_diag_u[pos] *  d_p[pos + imax*jmax] );
28             d_VetAux[pos] = d_p[pos];
29             d_p[pos]      = omg*( d_rhs[pos] - aux ) / d_diag_p[pos] + (1-
                   omg)*d_p[pos];
30         }
31     }
32 }
```

4 Experimentos

Neste capítulo são apresentados os resultados de cinco problemas com solução conhecida para a avaliação do algoritmo proposto: (*i*) Cavidade com superfície deslizante, (*ii*) Escoamento sobre um degrau, (*iii*) Escoamento laminar com um obstáculo circular, (*iv*) Convecção natural e (*v*) Convecção Rayleigh-Bénard. Para cada problema são apresentados os parâmetros físicos e os domínios utilizados. São considerados diferentes tamanhos de malha e analisados os tempos de processamento, número de iterações e *speedup* para todos os problemas. São utilizadas seis versões do algoritmo, que levam em consideração a arquitetura utilizada (CPU, GPU/CUDA), a influência do uso da memória *shared* nas versões em CUDA e o tipo de representação de dados utilizado em cada versão (*float*, *double*). Uma análise do número de *threads* disparado por *kernel* nas versões CUDA é apresentada para o primeiro problema (Seção 4.1).

Com o objetivo de simplificar a notação adotamos a seguinte nomenclatura para estas versões:

$\boldsymbol{seq_f}$ – corresponde a versão sequencial com tipo *float*;

$\boldsymbol{seq_d}$ – corresponde a versão sequencial com tipo *double*;

$\boldsymbol{C_f}$ – corresponde a versão CUDA com tipo *float*;

$\boldsymbol{C_d}$ – corresponde a versão CUDA com tipo *double*;

$\boldsymbol{Cs_f}$ – corresponde a versão CUDA *shared* com tipo *float*.

$\boldsymbol{Cs_d}$ – corresponde a versão CUDA *shared* com tipo *double*.

É utilizado o método iterativo Red-Black-SOR para a resolução dos sistemas lineares resultantes na etapa implícita do algoritmo em todas as versões implementadas. O critério de parada deste método consiste em atingir uma tolerância igual a 10^{-3} ou um número máximo de iterações, que nunca foi atingido em nossos experimentos. Como o cálculo

do resíduo a cada iteração onera o código significativamente, optamos por avaliar se a tolerância foi atingida a cada cinco iterações do método Red-Black-SOR. Em todos os experimentos adotamos um passo de tempo adaptativo como definido na Equação (2.32). Nos problemas bidimensionais utilizamos o valor do parâmetro $\gamma = 0.9$ – parâmetro da discretização por diferenças finitas utilizando diferenças centrais e o esquema *donor-cell* (ver seção 2.3.4). Para os problemas tridimensionais o valor do parâmetro foi $\gamma = 0$, ou seja, foi utilizada apenas as diferenças finitas centrais na discretização dos experimentos tridimensionais.

O tempo de execução dos algoritmos é medido através da função `gettimeofday()` da linguagem `C`. O tempo de execução considera todo o algoritmo, incluindo inicializações de variáveis e escrita em arquivos. Ou seja, a primeira e uma das últimas funções do algoritmo é a função `gettimeodday()`.

Para a aferição do tempo de execução final executamos cada experimento cinco vezes, descartamos o maior e o menor tempo obtido e calculamos a média das três medições restantes. O número total de iterações é obtido acumulando o número de iterações necessárias em cada passo do tempo. O *speedup*, representado na Equação (4.1), é obtido pela razão entre o tempo gasto para o processamento sequencial ($t_{sequencial}$) e o tempo gasto para o algoritmo paralelo ($t_{paralelo}$), versões C_f, C_d, Cs_f e Cs_d.

$$Sp = \frac{t_{sequencial}}{t_{paralelo}} \tag{4.1}$$

Os testes foram executados em uma CPU Intel Core i7 930 (Quad-Core) $2,80GHz$ com $8192KB$ de cache L2 e $12GB$ de DRAM, com sistema operacional Fedora 13 (Goddard) 64 bits e o compilador `gcc` 4.4.5 com a flag de otimização `-O3`. A placa de vídeo utilizada foi a NVIDIA GeForce GTX 480, que é uma placa com arquitetura Fermi e possui $1536MB$ de DRAM GDDR5 e 64KB de memória interna ao chip com tamanho configurável entre *cache* L1 e *shared*, nos experimentos utilizamos os valores padrão, ou seja, 48KB para a memória *shared* e 16KB para a memória it cache L1. A versão do compilador utilizado foi o `nvcc` 3.2. Esta placa possui 15 *Stream Multiprocessor* (SM), cada um com 32 *cores* CUDA (*Stream Processor* – SP), totalizando 480 SPs. Os SPs operam com um clock máximo de $1401MHz$. Esta placa possui ainda 32768 registradores por bloco e suporta um máximo de 1024 *threads* por bloco, o tamanho máximo em cada dimensão de bloco é de $1024 \times 1024 \times 64$ e o tamanho máximo em cada dimensão de um grid é de $655635 \times$

655635×655635[1]

4.1 Problema da Cavidade com Cobertura Deslizante

4.1.1 Problema Bidimensional

O problema da cavidade com cobertura deslizante (GHIA *et al.*, 1982; PINELLI; VACCA, 1994) consiste em um domínio quadrado, com lados de tamanho iguais a 1 *metro*, preenchido com fluido, conforme a Figura 22. O topo do domínio, ou seja, a parte superior, se move com velocidade constante, causando assim movimentação do fluido. Impõe-se condição de Não-deslizamento nos demais seguimentos de contorno, ou seja, nos contornos: inferior, direito e esquerdo. A velocidade do contorno superior é igual a ($u = 1$ e $v = 0$) e as velocidades nos demais contornos são nulas ($u = 0$ e $v = 0$). Este problema representa um escoamento que, a partir de uma condição inicial, o movimento é gradualmente acelerado até que uma condição de regime permanente seja atingida, isto é, até que a solução se torne estacionária a partir de um determinado tempo. Inicialmente as velocidades u e v internas ao domínio são nulas. O número de Reynolds utilizado nos teste foi igual a 1000 e o tempo final de computação necessário para se atingir o regime permanente foi $t_{final} = 26$ *segundos*.

A Figura 23 apresenta os gráficos em barras do tempo de processamento para malhas de tamanho igual a 32×32, 64×64, 128×128, 256×256, 512×512 e 1024×1024 considerando tipo de dados *float*, uma variação no número de *threads* por bloco e um número fixo de 30 blocos a serem disparados pelos *kernels* CUDA. Para o número de *threads* disparadas consideramos os seguintes valores: 32 (um *warp* completo), 64, 128, 256 e 512. Percebemos que, na menor malha, apresentada na Figura 23(a), a diferença entre os tempos de execução é muito pequena e o melhor resultado foi obtido disparando 128 *threads*, ou seja, o número de *threads* disparado é aproximadamente quatro vezes maior que o número de células. Isso ocorre pois, para uma malha pequena, não é possível obter um bom balanceamento da arquitetura. Para as demais malhas, percebemos que o valor ótimo do número de *threads* segue o tamanho da malha, ou seja, 64 *threads* para a malha 64×64, 128 *threads* para a malha 128×128, e assim por diante até a malha 512×512.

A Figura 24 apresenta os resultados considerando tipo de dados *float*, variação do

[1] É possível verificar algumas características da placa por meio de funções da API CUDA. Para maiores informações consultar (NVIDIA, 2011a)

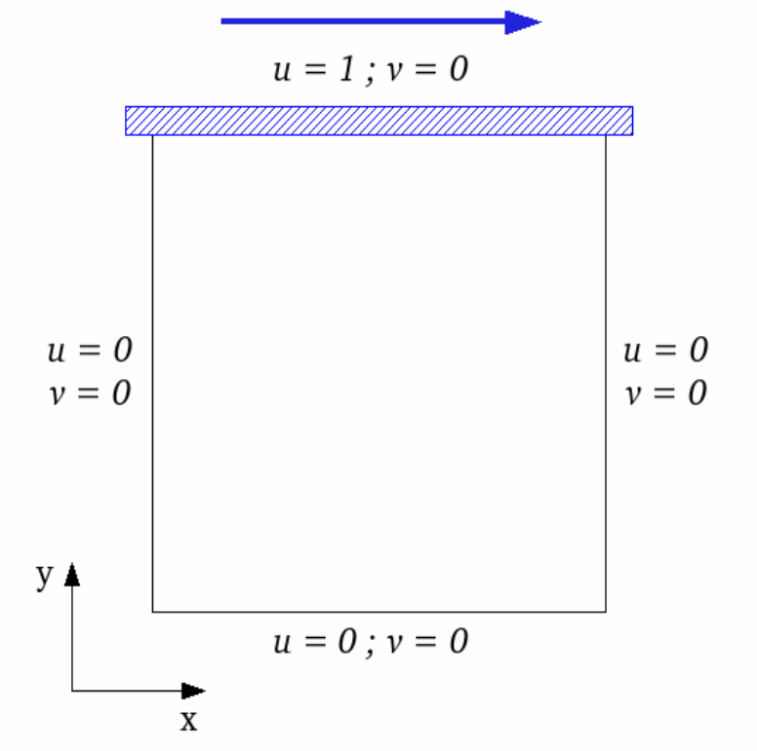


Figura 22: Descrição do Problema – Escoamento em uma cavidade com superfície deslizante.

número de blocos e o número fixo de *threads* por bloco para dois tamanhos de malha 256×256 e 512×512. Percebemos que o melhor número de blocos para a malha 512×512 é 30. Já para a malha 256×256 é 120, porém a diferença do tempo de execução considerando $120, 60$ ou 30 blocos é pequena. Desta forma escolhemos fixar o número de blocos em `NUM_BLOCKS` = 30, o que consiste em 2 blocos para cada SM da placa gráfica utilizada. Já o número de *threads* por bloco (`NUM_THREADS`) é escolhido de acordo com o tamanho da malha, sendo: 128 *threads* por bloco para a malha 32×32 e para as demais malhas o número de *threads* por bloco segue a dimensão x do domínio, até ser atingido o limite de 512 *threads* por bloco.

A Figura 25 apresenta a análise do número de *threads* disparados para o tipo de dados *double* e malhas de tamanho 256×256 e 512×512. Observamos que, para este tipo de dado, os resultados do tempo de execução para números de *threads* distintos apresentam comportamento semelhante aos resultados obtidos com dados do tipo *float*. Assim, decidimos utilizar os mesmos valores de número de blocos e de *threads* por bloco tanto para as versões com tipo de dados *float* quanto para o tipo *double*. Os valores de `NUM_BLOCKS` e `NUM_THREADS` para os demais problemas são escolhidos de forma semelhante.

A Figura 26 apresenta os perfis das velocidades para uma malha 512×512 considerando uma linha horizontal, para a direção x, e uma linha vertical, para a direção

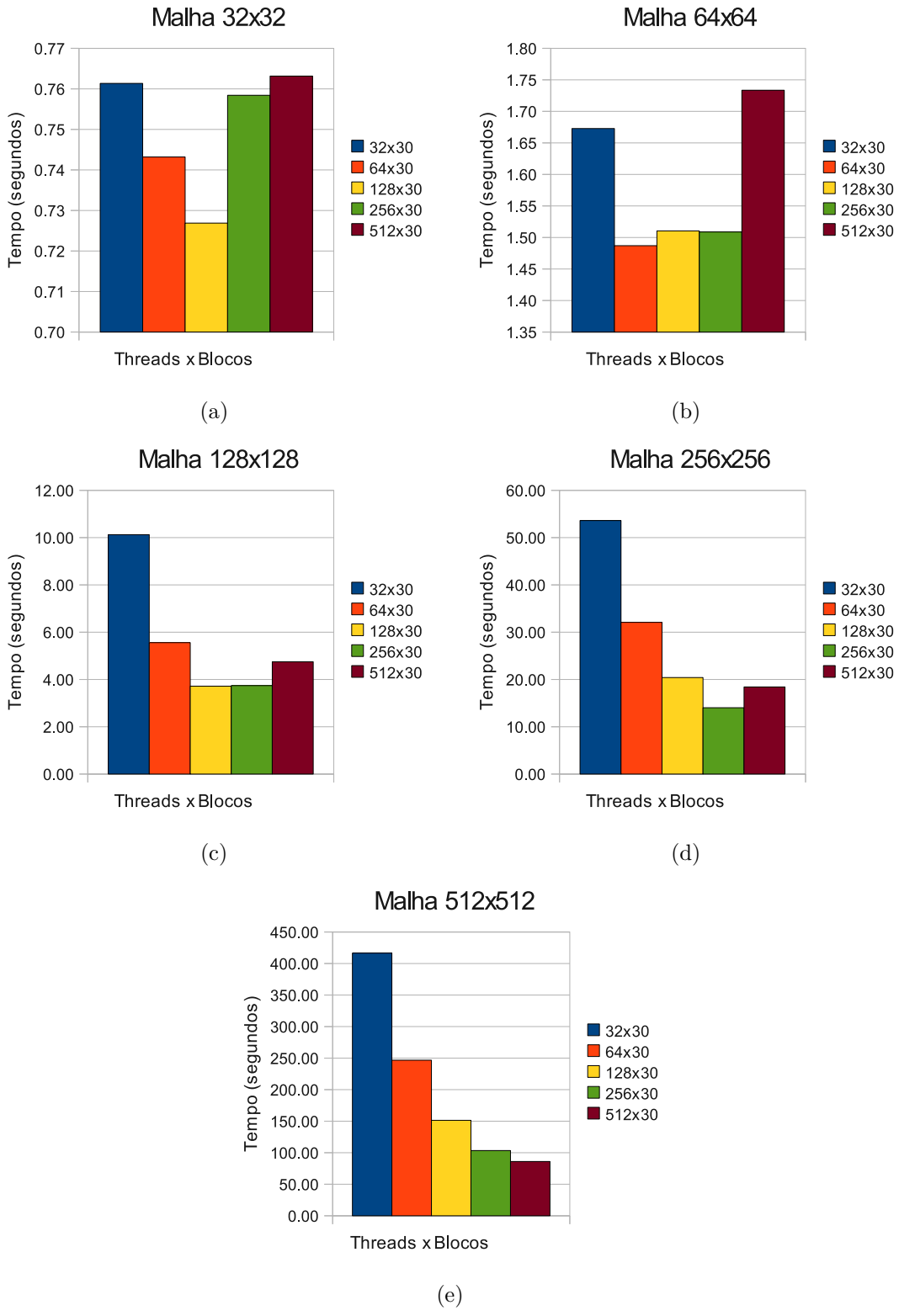
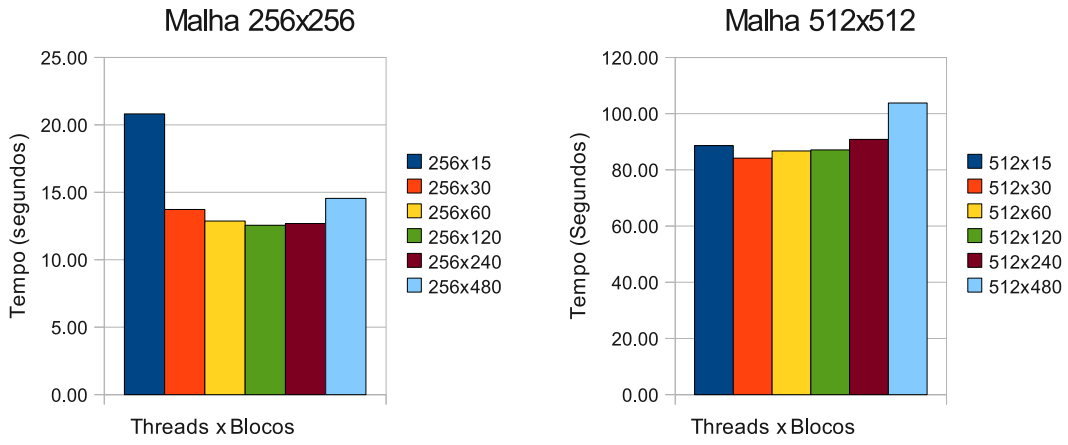


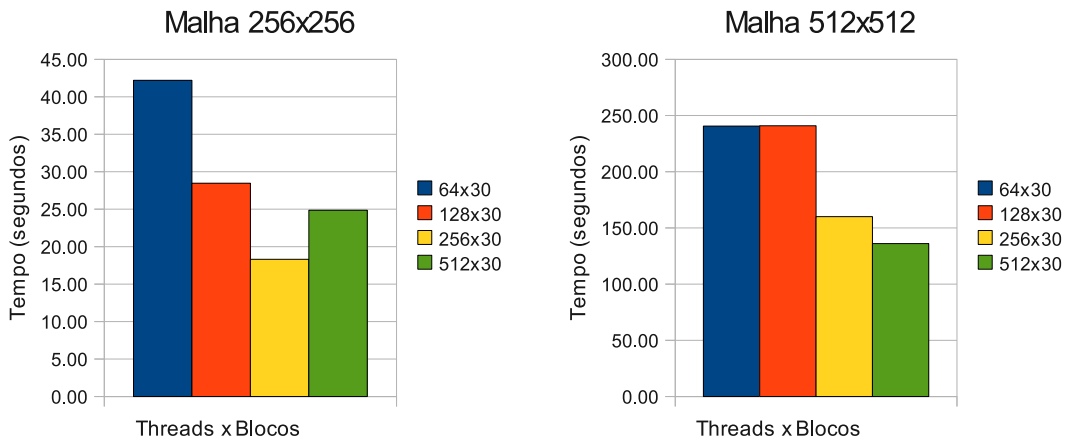
Figura 23: Tempo de processamento em relação ao número de *threads* disparadas considerando dados do tipo *float*.

y, que passa pelo centro geométrico da cavidade e considera os algoritmos: Seq_d; C_f; C_d e o resultado semi-analítico apresentado em (ERTURK *et al.*, 2005). Observa-se que os algoritmos desenvolvidos neste trabalho (Seq_d, C_f e C_d) produzem resultados equiva-



(a) Malha 256 × 256. (b) Malha 512 × 512.

Figura 24: Tempo de processamento considerando a variação do número de Blocos e número de *threads* por Bloco fixo considerando dados do tipos *float*.

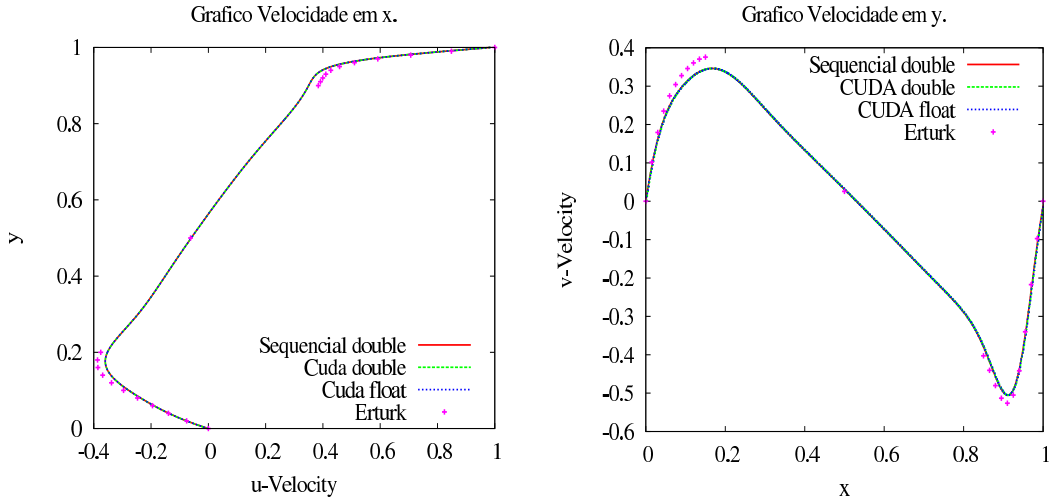


(a) Malha 256 × 256. (b) Malha 512 × 512.

Figura 25: Tempo de processamento considerando a variação do número de Blocos e número de *threads* por Bloco fixo com tipo de dados *double*.

lentes, mesmo quando uma solução com precisão do tipo *float* no algoritmo CUDA for considerada. A solução semi-analítica em (ERTURK *et al.*, 2005) utiliza uma malha mais refinada, 640 × 640, que a definida nos demais experimentos da Figura 26 (512 × 512), fato que explica a pequena diferença entre as soluções. Como os resultados não diferem significativamente entre si, a acuidade da aproximação, tanto para o algoritmo sequencial quanto para os algoritmos CUDA, está garantida. A Figura 27, mostra as *streamlines* e os vetores do campo de velocidade calculados pelo algoritmo proposto. Observa-se que os resultados obtidos são similares aos apresentados em (GRIEBEL, 1998).

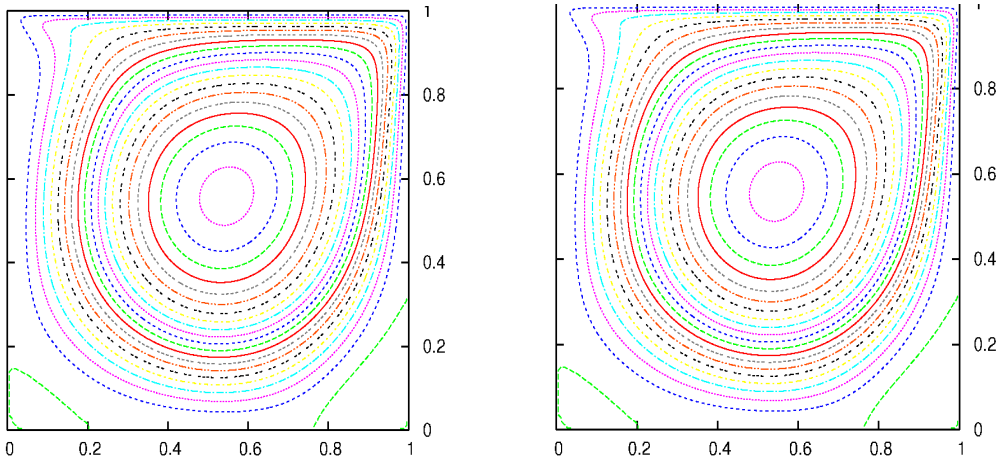
A Tabela 1 apresenta os tempos de execução em segundos, o número total de iterações e os *speedups* para diferentes tamanhos de malhas considerando as seguintes versões do



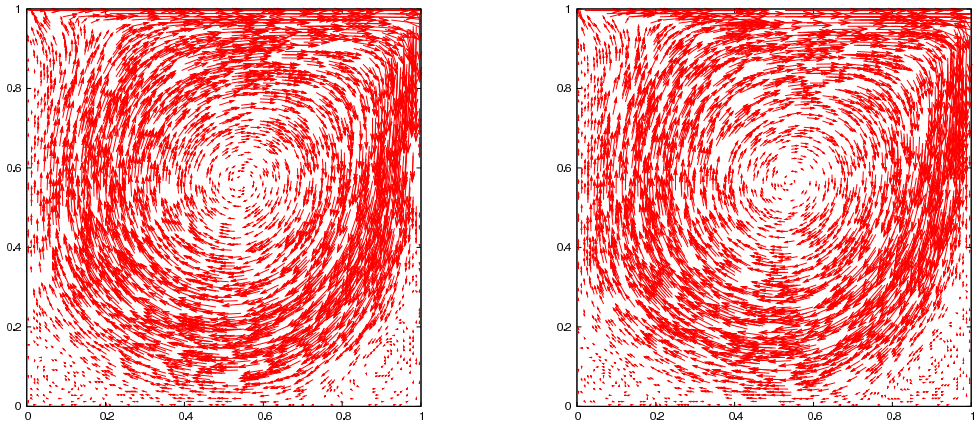
(a) Perfil da velocidade u.

(b) Perfil da velocidade v.

Figura 26: Perfil das velocidades – problema da cavidade com superfície deslizante em regime permanente para $Re = 1000$.



(a) *Streamlines* para os casos: sequencial (esquerda) e CUDA com tipo *double* (direita).



(b) Campo de velocidade para os casos: sequencial (esquerda) e CUDA com tipo *double* (direita).

Figura 27: *Streamlines* e campo de velocidade do algoritmo proposto para uma malha 128×128 – problema da cavidade com superfície deslizante em regime permanente para $Re = 1000$.

algoritmo: Seq_f, C_f e Cs_f. Nesta tabela apresentamos os resultados utilizando dados *float*. Observamos que a variação do número de iterações entre as versões do algoritmo é pequena e justificável pelo fato de serem executadas em arquiteturas distintas (CPU e GPU/CUDA). Observamos ainda que, a medida que aumentamos o número de células, ou seja, refinamos a malha, o *speedup* também aumenta. Esse aumento se dá devido a um melhor aproveitamento da arquitetura da GPU, pois o maior número de células possibilita um melhor balanceamento de carga. O uso da memória *shared* pouco influenciou os resultados para este experimento, ocasionando para a maioria das malhas um *speedup* discretamente inferior àquele alcançado utilizando apenas a memória global. Fatores como o baixo número de pontos de reuso de memória encontrados no código e a disponibilidade de uma memória *cache* por bloco na arquitetura CUDA fermi podem ter influenciado o baixo impacto do uso da memória *shared*. Mesmo assim, o uso da memória *shared* se torna importante, não diretamente para o uso como memória rápida de reuso de dados (ou seja, um *cache* programável), mas para a fácil utilização na cooperação de troca de dados entre as *threads* de um mesmo bloco. Por exemplo, na redução do valor máximo do um vetor (Algoritmo3.3), o uso da memória *shared* minimiza a necessidade de escrita na memória global da GPU.

A Tabela 2 apresenta os tempos de execução em segundos, o número de iterações e os *speedups* para diferentes tamanhos de malhas considerando as seguintes versões do algoritmo: Seq_d, C_d e Cs_d, isto é, resultados similares àqueles observados na Tabela 1, mas considerando tipo de dados *double*. Como no caso anterior, observamos o aumento do *speedup* a medida que aumentamos o número de células no problema. Cabendo aqui a mesma explicação, um maior número de células possibilita um melhor balanceamento de carga entre as *threads* disparadas. Novamente o uso da memória *shared* não apresenta uma melhora do *speedup*.

Analisando os *speedups* obtidos paras as versões CUDA apresentados nas Tabelas 1 e 2, verificamos que o uso do tipo de dados *float* obteve melhores resultados. Este fato já era esperado uma vez que a arquitetura CUDA utiliza de forma distinta os *cores* CUDA de acordo com o tipo de dado (*float* ou *double*), como mencionado anteriormente (ver seção 3.2.1). Sendo assim, o desempenho da arquitetura, a priori, cairia à metade utilizando tipo de dados *double*, pois o tipo *double* utiliza dois cores em sua execução. Porém, verificamos que, para as malhas testadas, o desempenho com tipos de dados *double* foi melhor que o esperado. Mesmo para a maior malha (512×512) o *speedup* para as versões com tipo de dados *double* é superior a metade do valor dos *speedups* obtidos nas versões com tipo de

dados *float*. Porém, devemos considerar que o código não é inteiramente *double*, uma vez que há a execução de outros tipos de dados, como o tipo *int*, utilizados nas estruturas de repetição (*loops*), que também consideram um *core* CUDA para a sua execução.

Tabela 1: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema da Cavidade em regime permanente para $Re = 1000$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
32×32	$0,40$	$0,73$	$0,73$	16525	16560	16560	$0,55$	$0,55$
64×64	$2,88$	$1,45$	$1,45$	34135	34850	34850	$1,99$	$1,99$
128×128	$23,69$	$3,56$	$3,60$	66280	69735	69735	$6,65$	$6,58$
256×256	$203,36$	$13,02$	$13,55$	132520	132520	132620	$15,62$	$15,01$
512×512	$1834,86$	$80,47$	$83,01$	270805	270835	270835	$22,80$	$22,10$

Tabela 2: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema da Cavidade em regime permanente e $Re = 1000$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_d	C_d	Cs_d	Seq_d	C_d	Cs_d	C_d	Cs_d
32×32	$0,51$	$0,76$	$0,76$	16525	16560	16560	$0,67$	$0,67$
64×64	$3,88$	$1,27$	$1,76$	34130	34850	34850	$3,06$	$2,20$
128×128	$34,40$	$4,35$	$5,05$	66310	69735	69735	$7,81$	$6,81$
256×256	$280,83$	$18,17$	$19,36$	132515	132620	132620	$15,46$	$14,51$
512×512	$2760,08$	$135,38$	$137,02$	270840	270835	270835	$20,39$	$20,14$

4.1.2 Problema Tridimensional

O problema da cavidade com superfície deslizante tridimensional é uma extensão para a dimensão z do problema da cavidade com domínio quadrado, apresentado anteriormente. Desta forma, o problema consiste em um domínio cúbico com arestas de tamanho 1 *metro*, preenchido com fluido, conforme apresentado na Figura 28. A face superior do domínio (U) se move com velocidade constante, provocando assim movimentação no fluido. Impõe-se condição de Não-deslizamento nas faces frente (F – *front*), atrás (B – *back*) e sul (D – *down*) e condição de Deslizamento nas faces oeste (W – *west*) e leste (E – *east*) [2]. A velocidade da face superior do contorno é igual a ($u = 1$, $v = 0$ e $w = 0$) e a velocidade nas demais faces do contorno são nulas ($u = 0$, $v = 0$ e $w = 0$). De forma semelhante ao problema bidimensional, o problema tridimensional parte de uma condição inicial e o movimento é gradativamente acelerado até uma condição de regime permanente ser atingida. Inicialmente as velocidade internas à cavidade são nulas ($u = 0$, $v = 0$ e $w = 0$). O número de Reynolds e o tempo final de computação escolhidos são os mesmos do problema bidimensional, sendo $Re = 1000$ e $t_{final} = 26$ *segundos*.

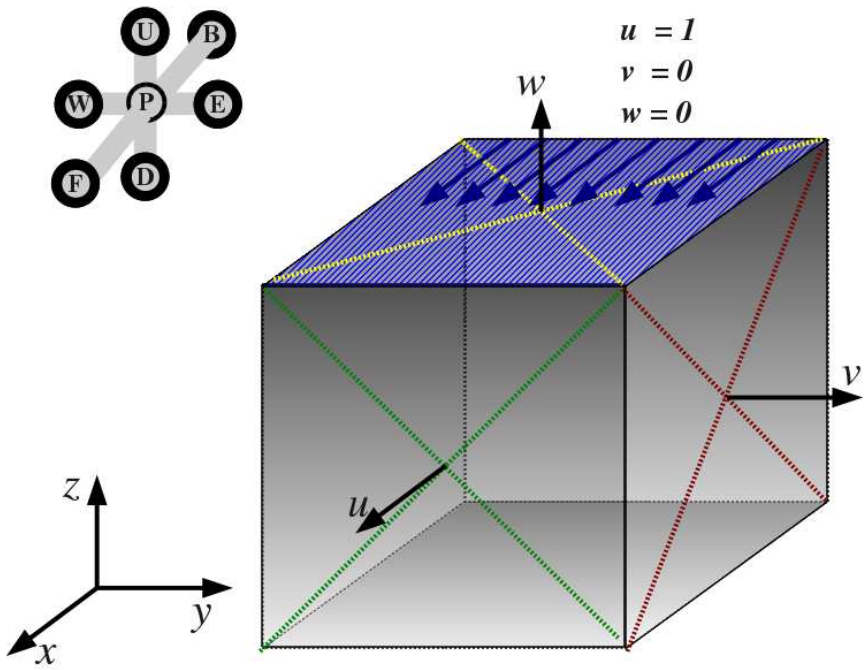


Figura 28: Descrição do Problema – Escoamento em uma cavidade cúbica com superfície deslizante.

A extensão para o problema tridimensional implica em dois fatores a serem analisados:

[2] Condições de contorno definidas na Seção 2.2

(i) uma diminuição considerável no valor do δt, uma vez que a condição de estabilidade do passo no tempo, Equação (2.32), se torna ainda mais restritiva; (ii) o aumento da quantidade de memória necessária, uma vez que a memória da GPU não realiza *swap*. A Tabela 3 apresenta uma estimativa do gasto de memória para os principais vetores alocados na GPU para as malhas tridimensionais utilizadas. Optamos por fixar o número de células em uma das dimensões uma vez que, o tamanho da memória RAM da placa NVIDIA GeForce GTX 480 é de 1536MB. Conforme mostrado na Tabela 3, para o tamanho da malha $512 \times 32 \times 512$, só os principais vetores ocupam em torno de 2/3 da memória da placa utilizada.

Tabela 3: Estimativa de memória utilizada na GPU para diferentes malha.

Malha	Memória
$32 \times 32 \times 32$	4MB
$64 \times 32 \times 64$	16MB
$128 \times 32 \times 128$	64MB
$256 \times 32 \times 256$	256MB
$512 \times 32 \times 512$	1024MB

Um estudo para a escolha do número de *threads* por bloco utilizado nas versões dos experimentos em CUDA foi realizado. A Figura 29 apresenta os resultados considerando o mesmo número de blocos fixo utilizado para os experimentos bidimensionais (`NUM_BLOCKS` = 30), e uma variação para o número de *threads* por bloco, considerando quatro malhas distintas: $32 \times 32 \times 32$, $64 \times 32 \times 64$, $128 \times 32 \times 128$, $256 \times 32 \times 256$ e $512 \times 32 \times 512$. Percebemos que na maioria das malhas testadas o valor `NUM_THREADS` = 256 obteve melhores resultados quanto ao tempo de execução ou obteve resultados muito semelhantes, com exceção da malha $512 \times 32 \times 512$ que apresentou melhor resultado para `NUM_THREADS` = 512. Sendo assim, decidimos escolher os valores do número de *threads* por bloco da seguinte forma: `NUM_THREADS` = 512 e `NUM_BLOCKS` = 30 para a malha $512 \times 32 \times 512$ e `NUM_THREADS` = 256 e `NUM_BLOCKS` = 30 para as demais malhas. Estes valores serão considerados para todos os experimentos tridimensionais.

A Figura 30 apresenta o perfil das velocidades para o plano xz considerando $y = 0.5$. Este perfil apresenta os valores das velocidades x e z considerando, respectivamente, uma linha horizontal, para a direção x, e uma linha vertical, para a direção z, que passa no centro geométrico da cavidade. Os resultados foram obtidos para uma malha de tamanho $512 \times 32 \times 512$. São considerados ainda os resultados das seguintes versões do algoritmo: Seq_d, C_d e C_f. Novamente percebemos que as versões do algoritmo desenvolvido neste trabalho produzem resultados equivalentes, mesmo utilizando dados com precisão *float*.

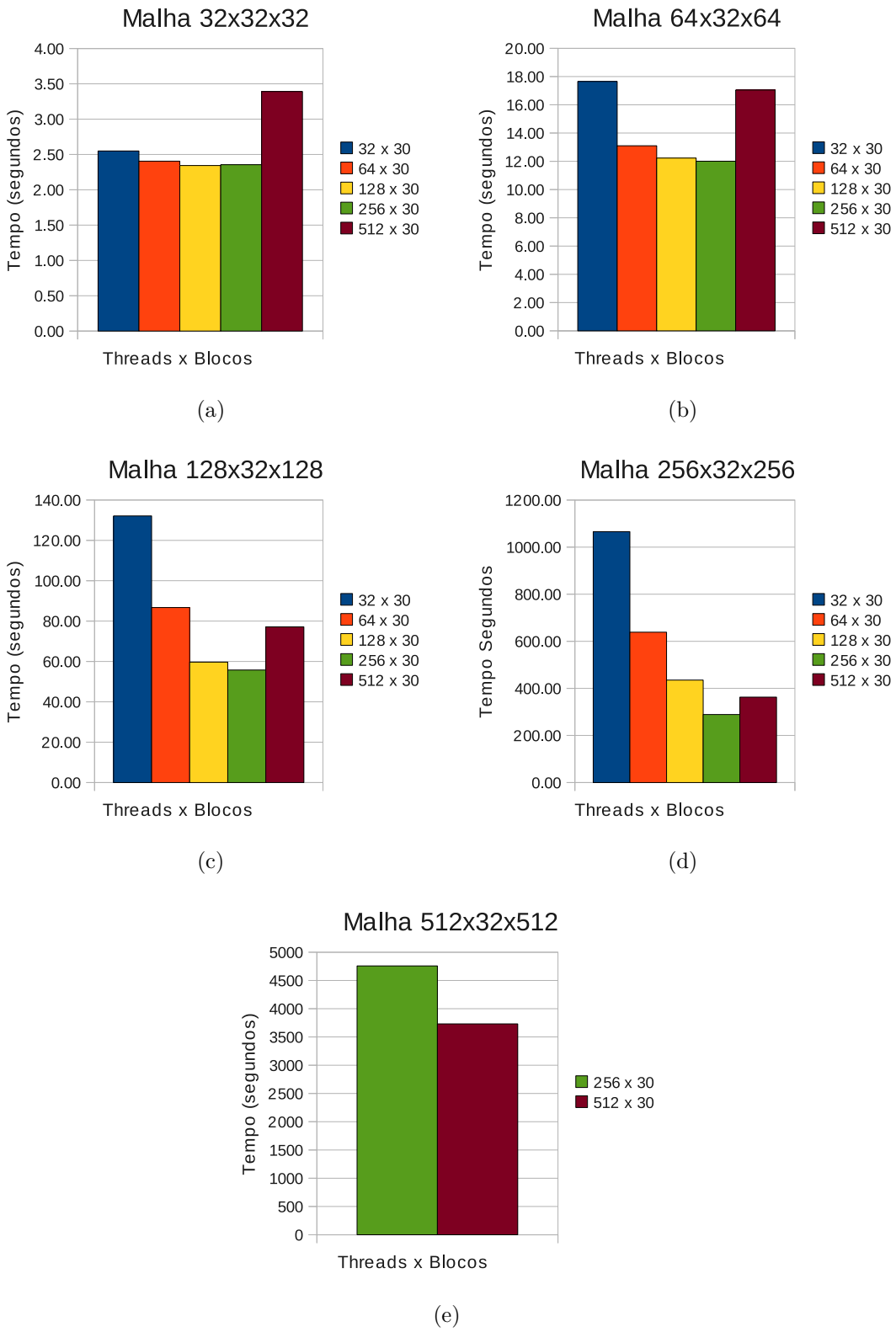
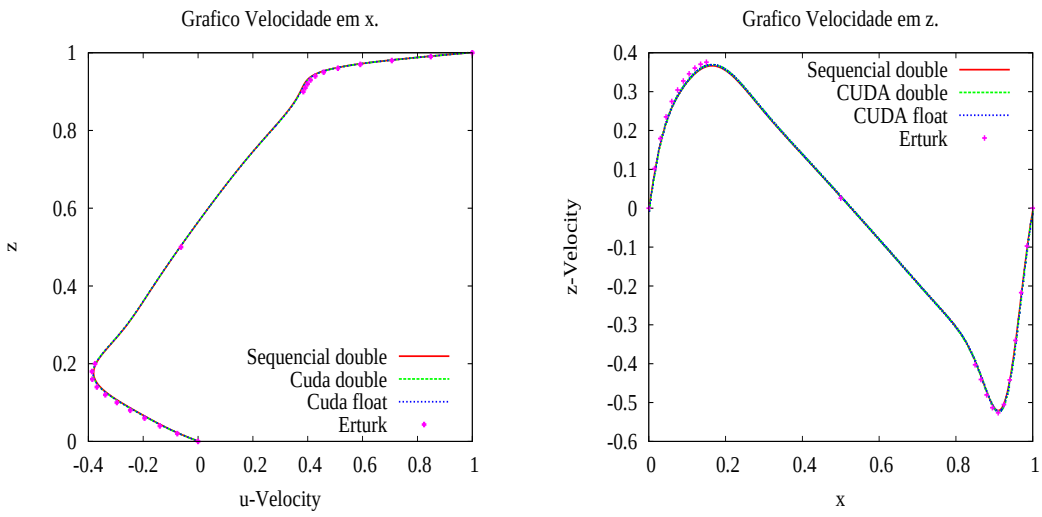


Figura 29: Tempo de processamento em relação ao número de *threads* disparadas considerando dados do tipo *float* – problema tridimensional.

Como os resultados obtidos não diferem significativamente entre si, a acuidade da aproximação, tanto do algoritmo sequencial quanto dos algoritmos CUDA, estão garantidas.



(a) Perfil da velocidade u.

(b) Perfil da velocidade w

Figura 30: Perfil das velocidades no plano xz para $y = 0.5$ – Problema da cavidade com superfície deslizante tridimensional em regime permanente para $Re = 1000$.

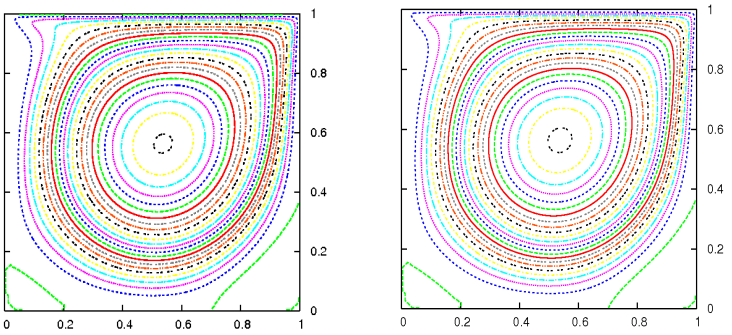
A Figura 32(a) e 32(b) apresentam, respectivamente, as *streamlines* e o campo de velocidade para o plano xz com $y = 0.5$ nas versões Seq_d (à direita) e C_d (à esquerda) para uma malha de tamanho $128 \times 32 \times 128$. Já as Figuras 32(c) e 32(d) apresentam o campo de velocidade dos mesmos experimentos em diferentes posições. Observamos que os resultados obtidos condizem com àqueles apresentados em (GRIEBEL, 1998).

A Tabela 4 apresenta os resultados de tempos de execução, número total de iterações e *speedup* considerando as versões do algoritmo que utilizam dados do tipo *float* (Seq_f, C_f e Cs_f). De forma semelhante ao problema bidimensional, observamos o aumento do *speedup* a medida que aumentamos o número de células na malha. Novamente o uso da memória *shared* não apresentou melhora no tempo de execução dos algoritmos.

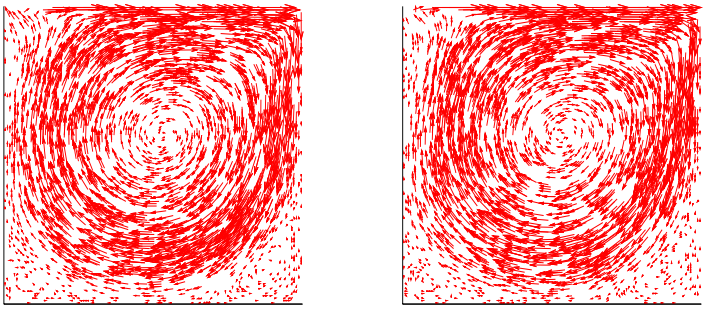
A Tabela 5 apresenta os mesmos resultados da tabela anterior considerando as versões do algoritmo que utilizam dados do tipo *double*. Os resultados obtidos, na maioria dos casos, seguem as mesmas características dos resultados obtidos com tipo *float*, ou seja, o *speedup* aumenta a medida que o número de células na malha cresce. Destacamos que para esse experimento na malha $256 \times 32 \times 256$ o uso da memória *shared* apresentou um pequeno ganho de desempenho.

Comparando os *speedups* obtidos nas tabelas 4 e 5 observamos que o tipo de dados *float* obteve melhor desempenho para as malhas maiores ($512 \times 32 \times 512$ e $256 \times 32 \times 256$), porém para as demais malhas o tipo *double* obteve melhores resultados.

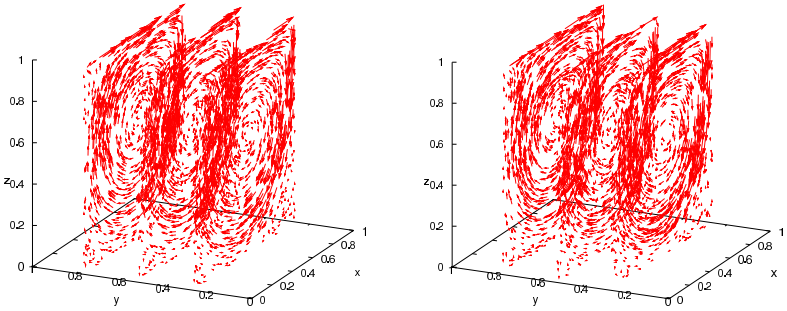
Em ambas as tabelas percebemos que, geralmente, o uso da memória *shared* propor-



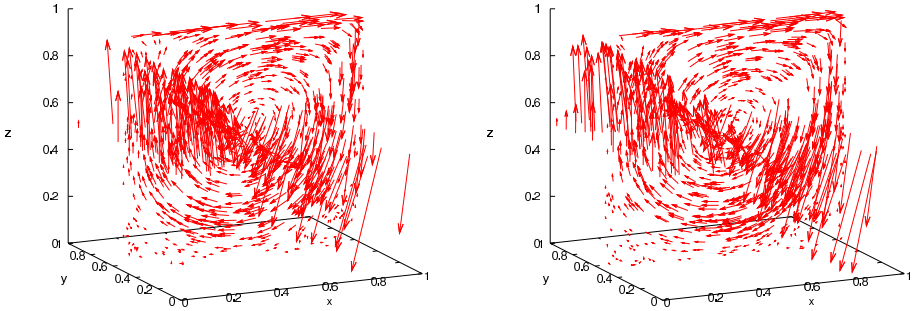
(a) *Streamlines* para os casos: sequencial (direita) e CUDA (esquerda) com tipo *double* – plano xz com $y = 0.5$.



(b) Campo de velocidades para os casos: sequencial (direita) e CUDA (esquerda) com tipo *double* – plano xz com $y = 0.5$.



(c) Campo de velocidades para os casos: sequencial (direita) e CUDA (esquerda) com tipo *double* – planos xz com $y_1 = 0.25$, $y_2 = 0.5$ e $y_3 = 0.75$.



(d) Campo de velocidades para os casos: sequencial (direita) e CUDA (esquerda) com tipo *double* – planos xy com $y = 0.5$ e xy com $z = 0.5$.

Figura 31: *Streamlines* para o plano xz com $y = 0.5$ e campo de velocidade em diferentes posições para uma malha de tamanho $128 \times 32 \times 128$ – Problema da cavidade com superfície deslizante tridimensional.

cionou um desempenho menor, com a exceção da malha $128 \times 32 \times 128$ com tipo *float*, porém percebemos que a diferença nos valores de *speedups* das versões com tipo *double* é menor que a diferença dos *speedups* com tipo *float*. Por exemplo, considerando a malha $512 \times 32 \times 512$, a diferença entre os *speedups* com tipo *double* é $Speedup_{C_d} - Speedup_{Cs_d} = 0,33$, enquanto que a diferença para o tipo *float* é $Speedup_{C_f} - Speedup_{Cs_f} = 5,11$. Acreditamos que isso ocorre porque, como haveria um maior uso do barramento para as versões com tipo *double* (pois o tipo *double* possui maior tamanho em relação ao tipo *float* – 4 bytes) o uso da memória *shared* ajudou a reduzir o uso deste barramento.

Tabela 4: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema da Cavidade tridimensional em regime permanente para $Re = 1000$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
$32 \times 32 \times 32$	$6,12$	$2,36$	$2,84$	5755	5755	5755	$2,60$	$2,15$
$64 \times 32 \times 64$	$63,14$	$12,01$	$13,87$	14030	14030	14030	$5,26$	$4,55$
$128 \times 32 \times 128$	$629,61$	$55,76$	$61,21$	30705	30715	30715	$11,29$	$20,29$
$256 \times 32 \times 256$	$5881,12$	$289,03$	$311,03$	67575	67580	67580	$20,35$	$18,91$
$512 \times 32 \times 512$	$94206,83$	$3731,58$	$4678,36$	270820	270825	270855	$25,25$	$20,14$

Tabela 5: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema da Cavidade tridimensional em regime permanente para $Re = 1000$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
$32 \times 32 \times 32$	$7,65$	$2,73$	$2,95$	5755	5755	5755	$2,80$	$2,59$
$64 \times 32 \times 64$	$92,92$	$13,50$	$14,62$	14030	14030	14030	$6,88$	$6,36$
$128 \times 32 \times 128$	$854,12$	$66,21$	$70,12$	30705	30715	30715	$12,90$	$12,18$
$256 \times 32 \times 256$	$7683,37$	$409,11$	$406,01$	67585	67590	67580	$18,78$	$18,92$
$512 \times 32 \times 512$	$128183,54$	$6015,79$	$6111,12$	270995	271000	270855	$21,31$	$20,98$

4.2 Problema do Escoamento sobre um Degrau

Outro problema encontrado na literatura para o processo de validação de códigos de simulação das equações de Navier-Stokes para fluidos incompressíveis é a análise do comportamento do fluido sobre um degrau (GARTLING, 1990; KAIKTSIS *et al.*, 1991), como mostrado na Figura 32. O fluido entra no domínio pela fronteira da esquerda, com condição de contorno de Entrada, com uma velocidade horizontal u_0 e sai pela fronteira da direita, que possui a condição de contorno de Saída – condições de contorno definidas na Seção 2.2 – e seu comportamento se altera devido ao alargamento do canal. Nas fronteiras superior e inferior são consideradas condições de contorno de Não-deslizamento. O domínio possui dimensões $(0; 29) \times (0; 1,5)$ e o degrau está localizado em $(7,5; 0,75)$.

Os valores iniciais para pressão e velocidades são $p = 0$, $u = 0$ e $v = 0$ e na metade superior do domínio consideramos $u = 1$ para que, no instante inicial, o fluido dentro do domínio satisfaça a equação da continuidade (GRIEBEL, 1998). O número de Reynolds utilizado nos experimentos foi igual a 100 e o tempo final de computação utilizado para se atingir o estado estacionário foi igual a $t_{end} = 100$ *segundos*. Os valores de `NUM_BLOCKS` e `NUM_THREADS` disparados nas versões CUDA foram escolhidos de forma semelhante ao problema anterior, ou seja, utilizamos um número fixo de blocos igual a 30 e um número variável para as *threads* por bloco igual ao tamanho da dimensão x da malha utilizada.

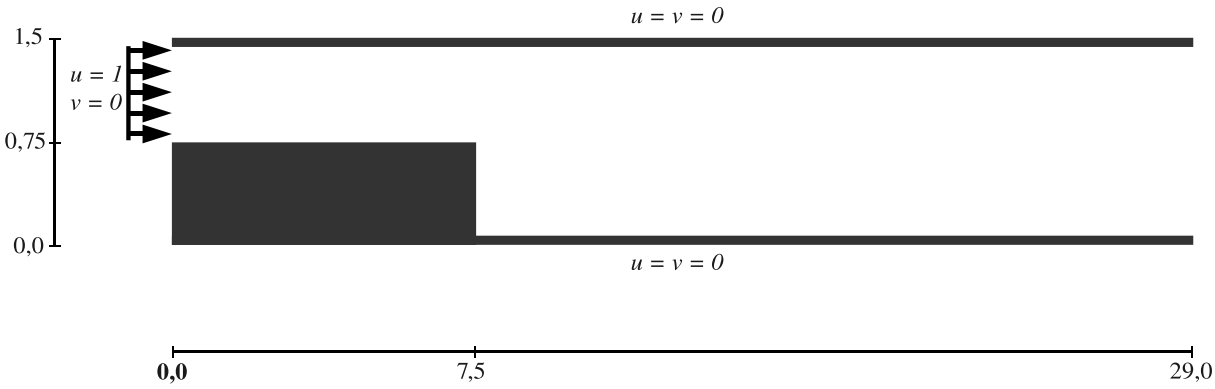
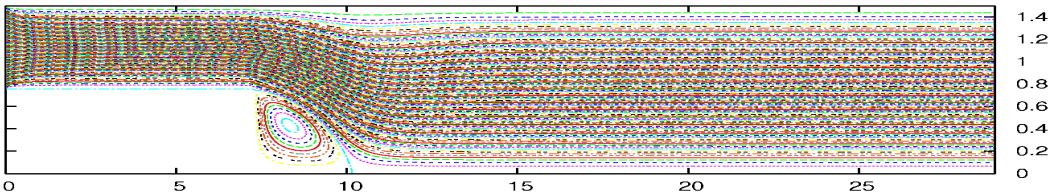


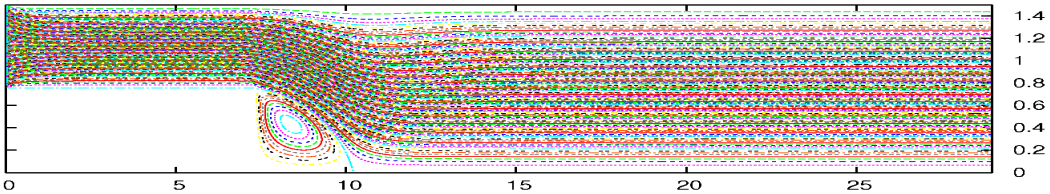
Figura 32: Descrição do problema – Escoamento sobre um degrau.

A Figura 33 mostra as linhas de corrente (*streamlines*) para as versões Seq_d, C_f e C_d considerando o domínio discretizado em uma malha com 512×128 células, sendo os resultados virtualmente iguais e similares aos obtidos por (GRIEBEL, 1998). Sendo assim, percebemos que o uso de diferentes precisões de dados (*float* ou *double*) nas versões CUDA não afetam significativamente a solução, como já verificado nos experimentos anteriores.

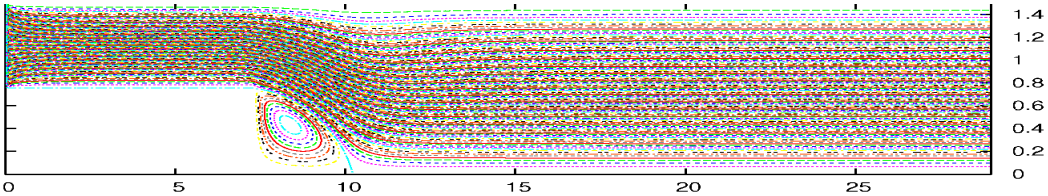
A Tabela 6 apresenta, para diferentes tamanhos de malha, os valores de tempo de



(a) Algoritmo sequencial com tipo *double*.



(b) Algoritmo CUDA com tipo *float*



(c) Algoritmo CUDA com tipo *double*

Figura 33: *Streamlines* para o problema do escoamento sobre um degrau em uma malha 512×128 no regime permanente para $Re = 100$.

execução em segundos, número de iterações e os *speedups* obtidos para as seguintes versões do algoritmo: Seq_f, C_f e Cs_f. Neste experimento consideramos tipo de dados *float*. Como no experimento anterior, notamos o aumento do *speedup* a medida que o número de células é aumentado. Novamente o uso da memória *shared* pouco impactou o *speedup*, e ainda apresentando um menor desempenho em relação à versão que não utiliza a memória *shared*. Para este problema podemos considerar, além dos fatores citados no experimento anterior, que apesar das células que formam o obstáculo (degrau) não serem executadas, os valores referentes a elas são copiados para a memória *shared*.

A Tabela 7 mostra os tempos de execução em segundos, o número total de iterações e os *speedups* obtidos considerando diferentes tamanhos de malhas, dados do tipo *double* e as seguintes versões do algoritmo: Seq_d, C_d e Cs_d. Como anteriormente, a medida que aumentamos o número de células, o *speedup* aumenta mas o ganho neste caso é menor, considerando as malhas maiores. Observamos ainda que a utilização da memória *shared* pouco influenciou nos *speedups*.

Tabela 6: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema do degrau em regime permanente para $Re = 100$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
128×32	$0,96$	$0,42$	$0,42$	13475	9540	9540	$2,29$	$2,29$
256×64	$14,47$	$1,53$	$1,56$	43980	37005	37005	$9,48$	$9,28$
512×128	$216,53$	$13,36$	$13,57$	159835	148515	148515	$16,21$	$15,96$
1024×256	$3755,09$	$192,19$	$214,21$	616645	598655	598655	$19,54$	$17,53$

Tabela 7: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema do degrau em regime permanente para $Re = 100$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
128×32	$1,27$	$0,42$	$0,42$	13475	9540	9540	$3,02$	$3,02$
256×64	$18,25$	$2,00$	$2,10$	43980	37005	37005	$9,13$	$8,69$
512×128	$298,73$	$20,97$	$21,02$	159865	148545	148545	$14,25$	$14,09$
1024×256	$6055,61$	$355,66$	$357,06$	617135	599465	599465	$17,03$	$16,96$

4.3 Problema do Escoamento Laminar com um Obstáculo Circular

O escoamento com obstáculos imersos também é um problema muito simulado para a validação das equações de Navier-Stokes para fluidos incompressíveis. A complexidade do obstáculo circular para tratamento das condições de contorno gera um código CUDA com muitas estruturas de decisão (*if*). Portanto, neste experimento, decidimos executar a função responsável pelas condições de contorno na CPU, e não como um *kernel* CUDA como nos demais experimentos; necessitando assim de cópia de dados entre GPU-CPU e CPU-GPU a cada passo no tempo. Com isso podemos observar o impacto desta troca de dados nos resultados.

Neste experimento consideraremos um escoamento ao redor de um obstáculo circular (disco) imerso em um fluido dentro de um canal (TRITTON, 1959; EATON, 1987), conforme descrito na Figura 34. O fluido entra pela fronteira esquerda com condição de contorno de Entrada e sai pela fronteira da direita com condição de contorno de Saída. Nas demais fronteiras, superior e inferior, é garantida a condição de contorno tipo Não-deslizamento. Os valores iniciais de pressão e velocidades no interior do domínio são, respectivamente, $p = 0$, $u = 0$ e $v = 0$, contudo é considerado uma velocidade horizontal de entrada na fronteira esquerda $u = 1$. O domínio possui dimensões $(0; 29) \times (0; 5)$ e o disco, cujo diâmetro é igual a 1, está centrado em $(2,5; 2,5)$.

Este problema, dependendo do valor de Re, não atinge um regime permanente, pois o escoamento ao redor de corpos angulosos, a partir de um determinado valor de Re, é caracterizado por um constante movimento oscilatório (GRIEBEL, 1998). Os testes foram realizados até o tempo computacional $t_{final} = 20$ *segundos* e com número de Reynolds igual a $Re = 100$. Os valores de `NUM_BLOCKS` e `NUM_THREADS` utilizados nas versões CUDA são similares aos utilizados nos problemas anteriores.

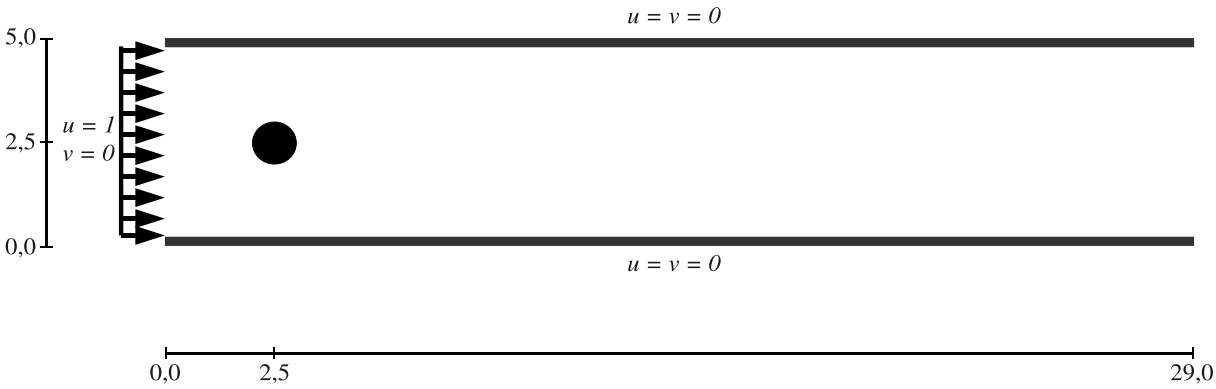
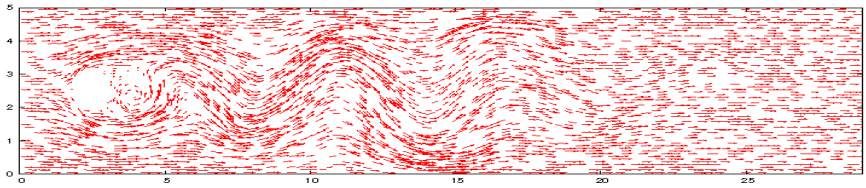


Figura 34: Descrição do problema - Escoamento com um obstáculo circular.

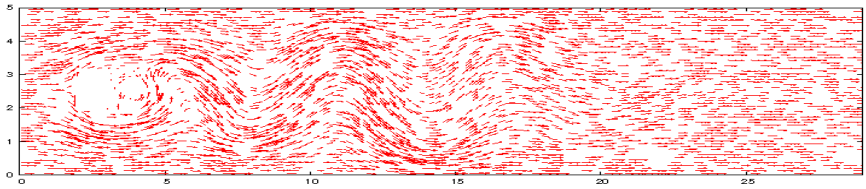
As Figuras 35 e 36 mostram, respectivamente, os campos de velocidades e as *streamlines* para uma malha 256×64 considerando as três versões do código, Seq_d (topo), C_f (meio) e C_d (base). Podemos perceber a similaridade entre as três versões apresentadas, mesmo considerando diferentes tipos de dados (*float*, *double*) com os resultados obtidos por (GRIEBEL, 1998).

As Tabelas 8 e 9 apresentam os tempos de execução em segundos, os números de iterações e os *speedups* obtidos considerando dados do tipo *float* e *double*, respectivamente, e diferentes tamanhos de malhas e versões do algoritmo. Diferentemente dos problemas anteriores, observamos neste experimento uma diferença significativa do número de iterações para as diferentes versões dos algoritmos. Por exemplo, na malha 1024×256 com tipo *double*, o algoritmo com a versão com memória *shared* realizou quase o dobro de iterações quando comparado a outra versão CUDA. Esta disparidade pode ocorrer, principalmente, devido a dois motivos: à imensa troca de dados entre a GPU e a CPU e/ou ao fato deste problema específico não possuir um estado estacionário. Com o objetivo de verificar como a simulação avança no tempo, vamos observar os valores de Δy, do número de iterações e do resíduo a cada passo de tempo.

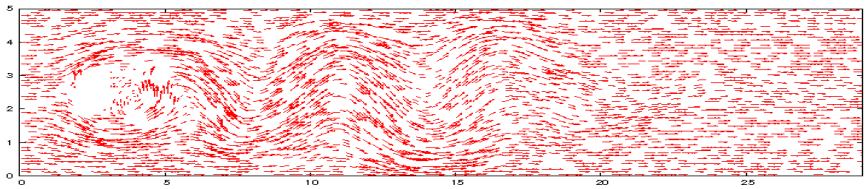
As Figuras 37, 38 e 39 apresentam os resultados destas variações, respectivamente, para as malhas 256×64, 512×128 e 1024×256, considerando o tipo de dados *float*. Percebemos que existe uma variação distinta para os três parâmetros observados ao longo



(a) Algoritmo sequencial com tipo *double*.

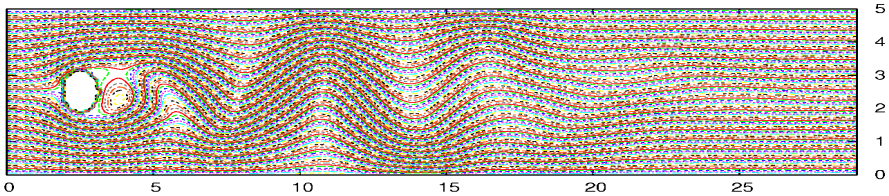


(b) Algoritmo CUDA com tipo *float*.

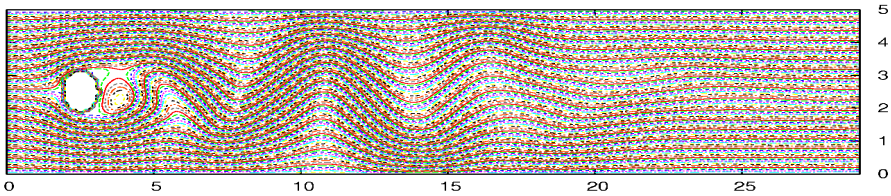


(c) Algoritmo CUDA com tipo *double*.

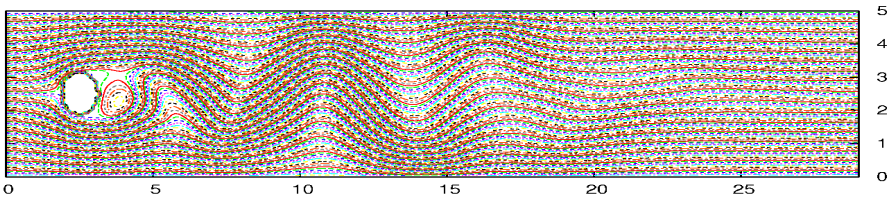
Figura 35: Campo de velocidades para o problema do escoamento com um obstáculo circular para uma malha 256×64 no tempo de computação $t_{final} = 20$ e para $Re = 100$.



(a) Algoritmo sequencial com tipo *double*



(b) Algoritmo CUDA com tipo *float*

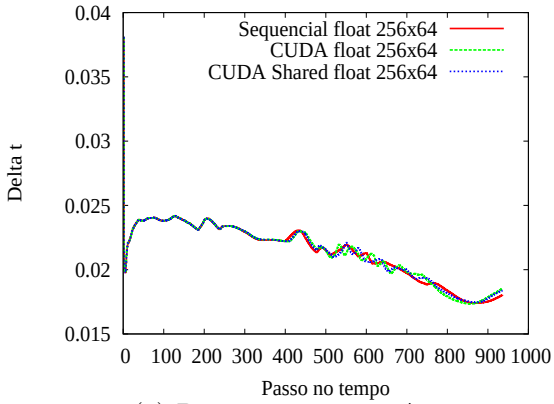


(c) Algoritmo CUDA com tipo *double*

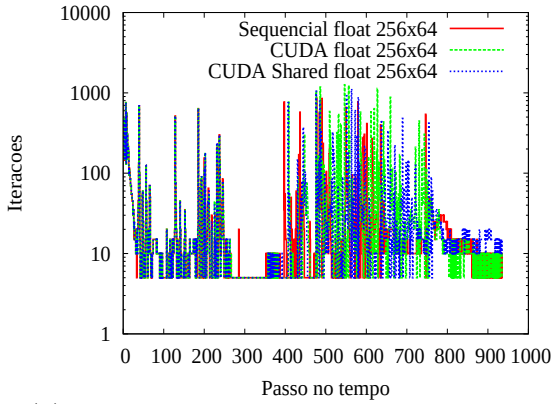
Figura 36: *Streamlines* para o problema do escoamento com um obstáculo circular para uma malha 256×64 no tempo de computação $t_{final} = 20$ e para $Re = 100$.

dos passos de tempo. Além disso, podemos observar que a variação é mais pronunciada a

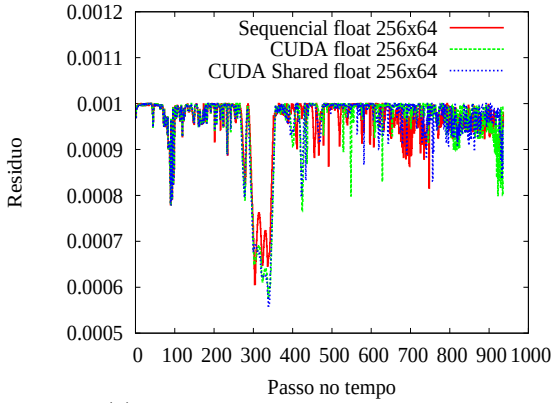
medida que a malha fica mais refinada.



(a) Passo no tempo × Δt



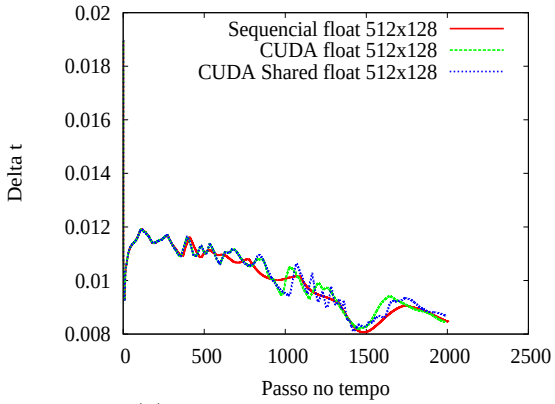
(b) Passo no tempo × Número de Iterações



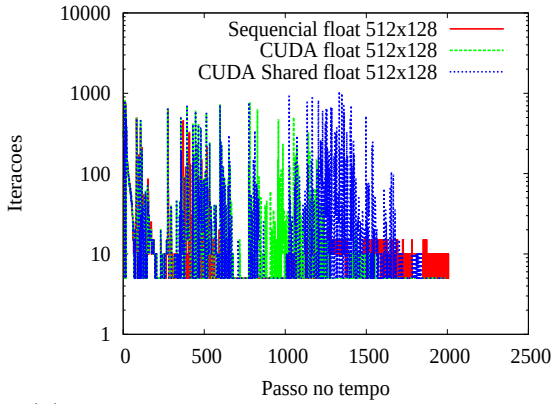
(c) Passo no tempo × Resíduo

Figura 37: Análise do Δt, número de iterações e resíduo para a malha 512×256 e tipo de dados *float* – Problema do obstáculo circular com tempo computacional $t_{final} = 20$ e $Re = 100$.

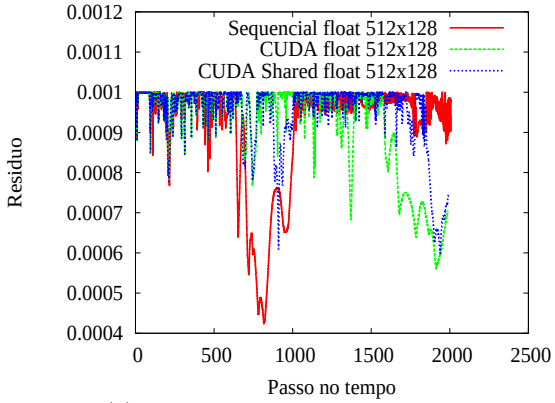
As Figuras 40, 41 e 42 apresentam os resultados destas variações, respectivamente, para as malhas 256×64, 512×128 e 1024×256, considerando o tipo de dados *double*. Percebemos as mesmas variações observadas para o caso *float*, porém neste caso, as variações são ainda mais pronunciadas a medida que as malhas crescem. Destacamos que para a malha maior, o número de iterações para os algoritmos CUDA são muito menores.



(a) Passo no tempo × Δt

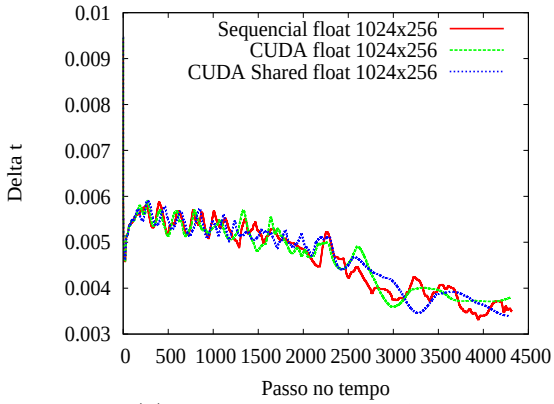


(b) Passo no tempo × Número de Iterações

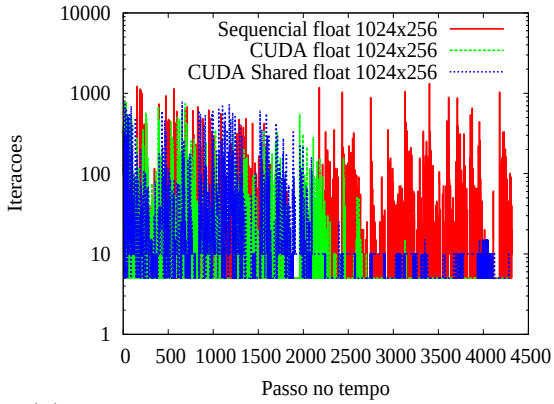


(c) Passo no tempo × Resíduo

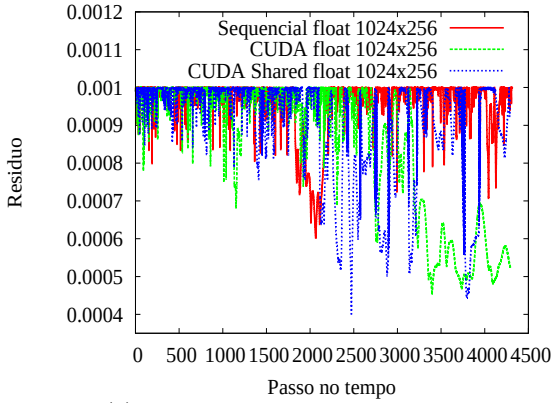
Figura 38: Análise do Δt, número de iterações e resíduo para a malha 512×128 e tipo de dados *float* – Problema do obstáculo circular com tempo computacional $t_{final} = 20$ e $Re = 100$.



(a) Passo no tempo × Δt

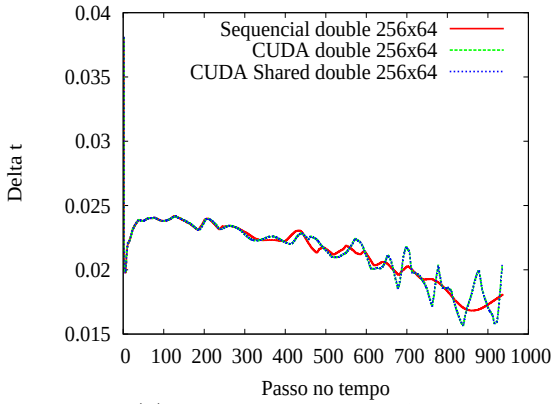


(b) Passo no tempo × Número de Iterações

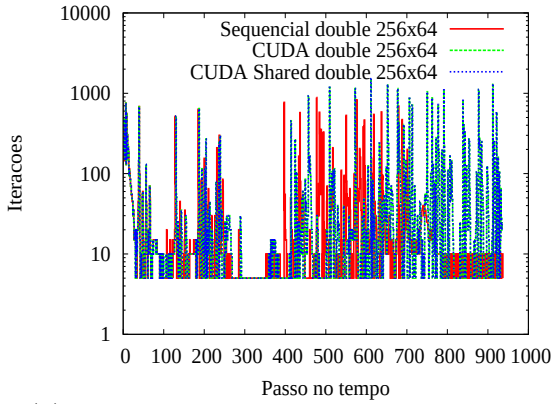


(c) Passo no tempo × Resíduo

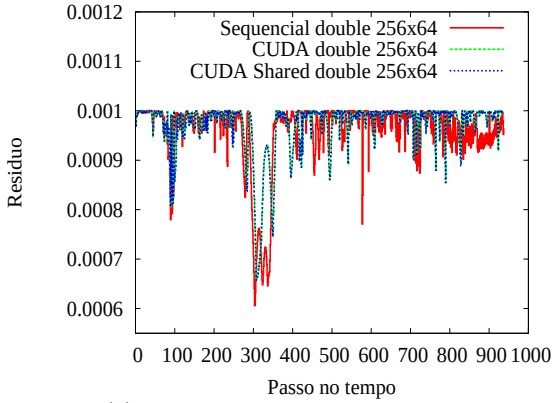
Figura 39: Análise do Δt, número de iterações e resíduo para a malha 1024×256 e tipo de dados *float* – Problema do obstáculo circular com tempo computacional $t_{final} = 20$ e $Re = 100$.



(a) Passo no tempo × Δt

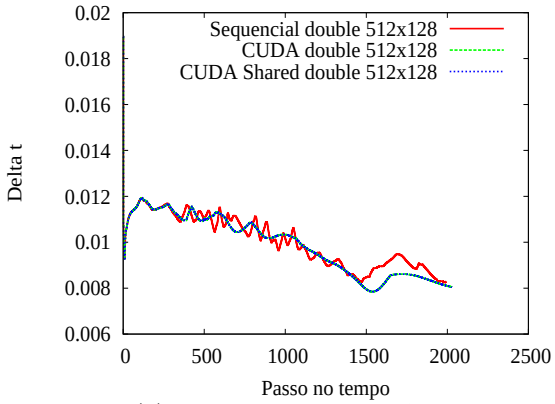


(b) Passo no tempo × Número de Iterações

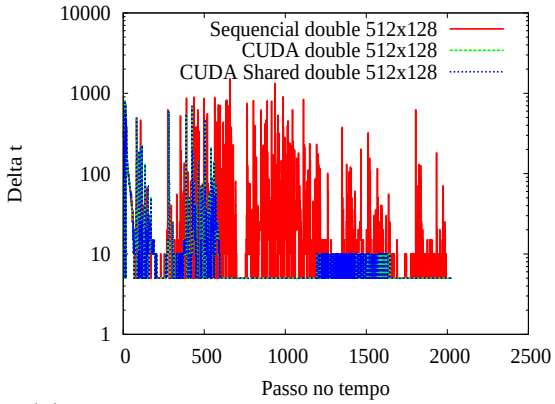


(c) Passo no tempo × Resíduo

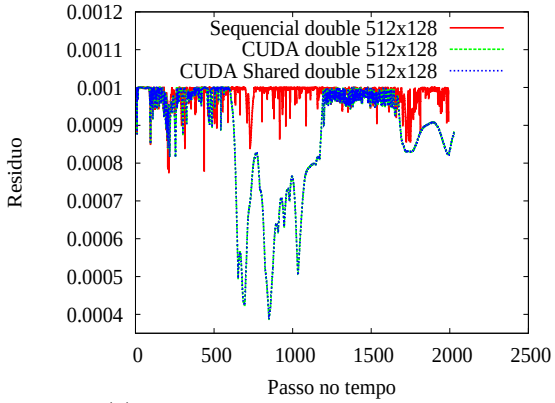
Figura 40: Análise do Δt, número de iterações e resíduo para a malha 512×256 e tipo de dados *double* – Problema do obstáculo circular com tempo computacional $t_{final} = 20$ e $Re = 100$.



(a) Passo no tempo × Δt

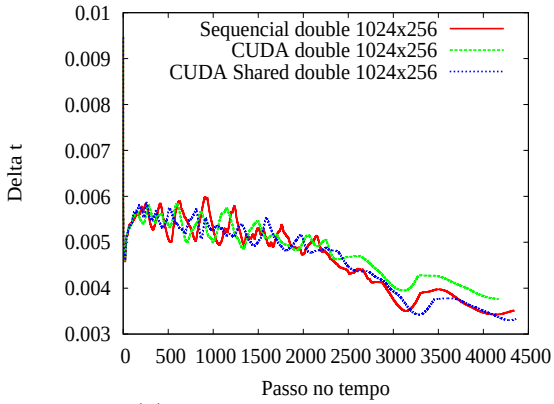


(b) Passo no tempo × Número de Iterações

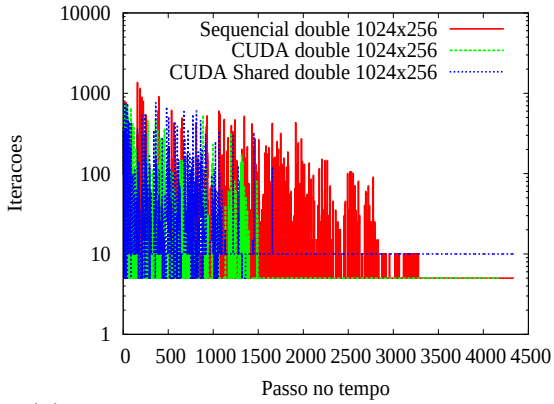


(c) Passo no tempo × Resíduo

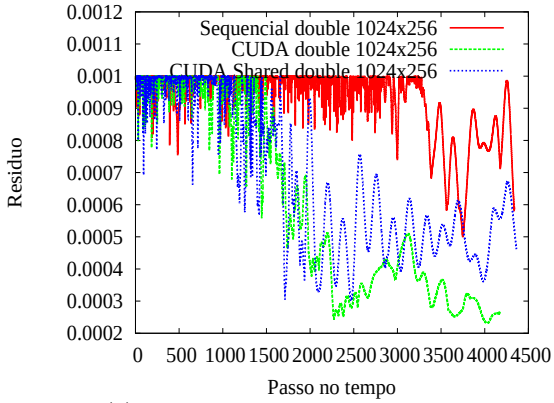
Figura 41: Análise do Δt, número de iterações e resíduo para a malha 512×128 e tipo de dados *double* – Problema do obstáculo circular com tempo computacional $t_{final} = 20$ e $Re = 100$.



(a) Passo no tempo × Δt



(b) Passo no tempo × Número de Iterações



(c) Passo no tempo × Resíduo

Figura 42: Análise do Δt, número de iterações e resíduo para a malha 1024×256 e tipo de dados *double* – Problema do obstáculo circular com tempo computacional $t_{final} = 20$ e $Re = 100$.

Destacamos que cada versão executa passos de tempo diferentes, uma vez que o passo de tempo é calculado de forma adaptativa. Sendo assim, concluímos que o comportamento do avanço no tempo é diferente para cada versão do algoritmo. Este comportamento era esperado, uma vez que, o problema não possui um estado estacionário, portanto os algoritmos avançam em passos de tempos distintos, o que acaba influenciando no número total de iterações ao final da execução do algoritmo.

Novamente a memória *shared* não contribuiu para o desempenho do algoritmo. Destacamos também que as versões *double* apresentaram um maior *speedup*, influenciados pelo varição no número de iterações já discutido anteriormente.

Tabela 8: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema do obstáculo cilíndrico para o tempo de computação $t_{final} = 20$ e $Re = 100$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
256×64	$5,76$	$1,02$	$0,96$	30885	39055	34510	$5,65$	$6,00$
512×128	$34,49$	$5,96$	$7,30$	31575	44725	67880	$5,79$	$4,72$
1024×256	$455,60$	$34,14$	$39,13$	142080	73660	89325	$13,35$	$11,64$

Tabela 9: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema do obstáculo cilíndrico para o tempo de computação $t_{final} = 20$ e $Re = 100$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_d	C_d	Cs_d	Seq_d	C_d	Cs_d	C_d	Cs_d
256×64	$6,21$	$1,23$	$1,26$	31540	48880	48880	$5,05$	$4,99$
512×128	$62,39$	$6,88$	$6,97$	82400	29720	29720	$9,08$	$8,95$
1024×256	$964,24$	$43,28$	$58,54$	93960	49645	80230	$22,28$	$16,47$

4.4 Problema da Convecção Natural

4.4.1 Problema bidimensional

A formação de um fluxo por meio de convecção natural gerado pela diferença de temperatura entre duas fronteiras foi utilizada para analisar o transporte implementado no algoritmo (ver Seção 2.1). O problema de convecção natural (KIMURA; BEJAN, 1983; PINELLI; VACCA, 1994), apresentado na Figura 43, consiste em um domínio quadrado com os lados de tamanho iguais a 1 *metro* preenchido com fluido. Impõem-se a condição de Não-deslizamento nas quatro fronteiras do domínio. Os valores iniciais de pressão, temperatura e velocidades são, respectivamente, dados por: $p = 0$, $T = 0$, $u = 0$ e $v = 0$. A temperatura na fronteira da esquerda é mantida com a valor constante, $T_{esq} = 1,0$, assim

como a fronteira da direita, $T_{dir} = 0,0$. Os contornos superior e inferior são considerados adiabáticos, ou seja, não há troca de calor com estas fronteiras. Os valores de `NUM_BLOCKS` e `NUM_THREADS` foram escolhidos da mesma forma apresentada no problema da cavidade (ver seção 4.1).

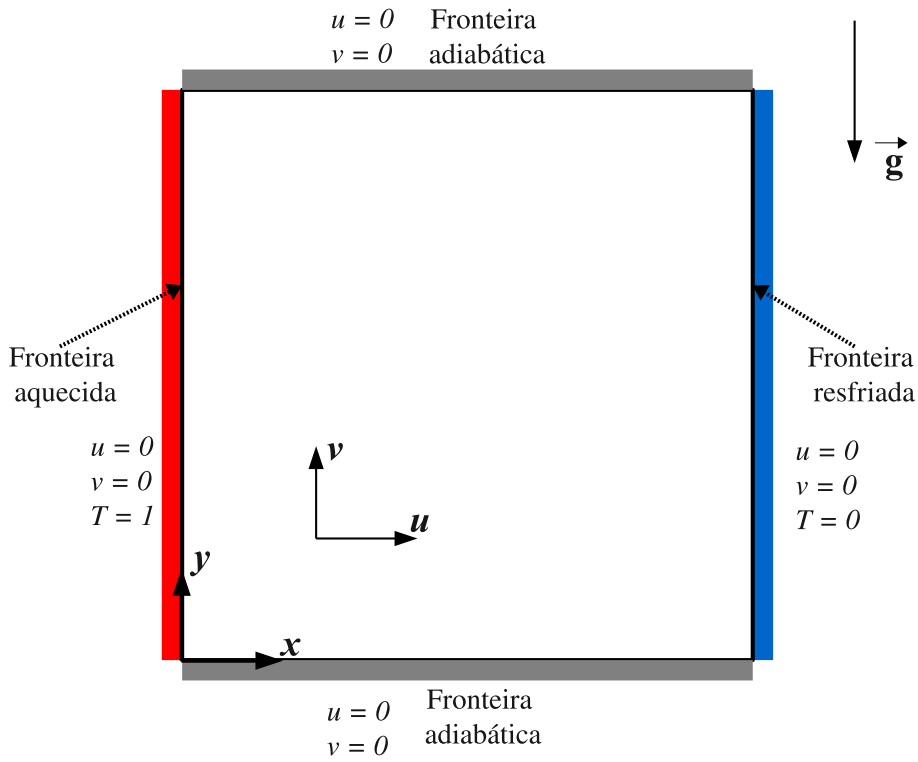


Figura 43: Descrição do domínio – Convecção Natural.

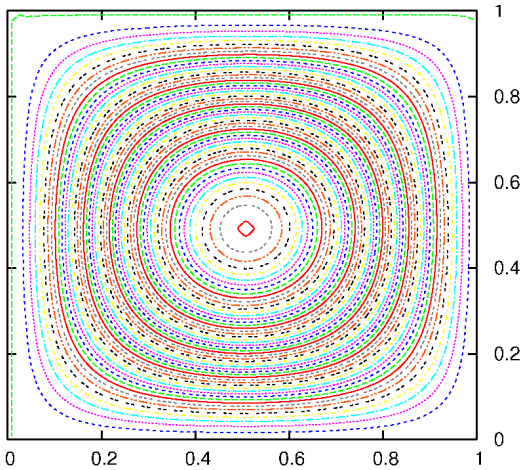
O problema da convecção natural representa um escoamento partindo de uma condição inicial, sendo gradualmente acelerado até que a solução se torne estacionária no tempo. O processo cíclico de aquecimento do fluido, gerando a sua expansão, seguida de um resfriamento do fluido, gerando assim compressão do mesmo, produz a movimentação do fluido.

Os valores dos números de Reynolds e Prandtl[3], do coeficiente de expansão termal e das componentes da força de corpo (gravidade) utilizados nos experimento foram, respectivamente: $Re = 985,7$, $Pr = 7$, $\beta = 2,1e^{-4}$, $\overrightarrow{g_x} = 0,0$ e $\overrightarrow{g_y} = -9.706e^{-2}$. O tempo final de computação necessário para se atingir o regime permanente é igual a $t_{final} = 5000$ *segundos*.

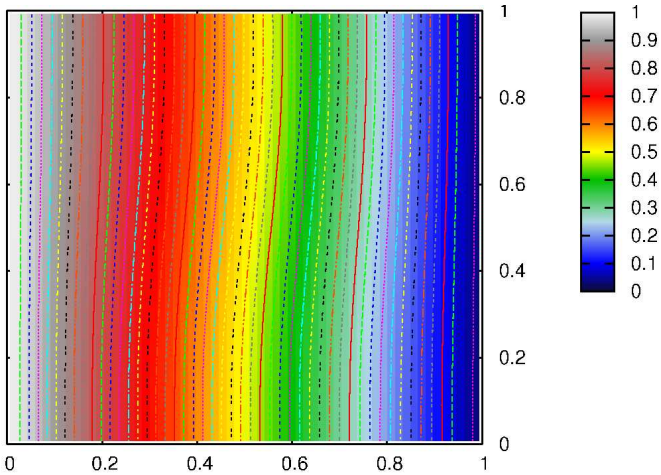
A Figura 44 mostra as linhas de corrente (*streamlines*), as linhas de contorno da temperatura e as linhas de fluxo de transporte de calor (*heatlines*) (GRIEBEL, 1998) obtidos pela versão CUDA, dados do tipo *double* e uma malha de tamanho 64×64. Observamos

[3]Definição dos números de Reynolds e Prandtl na Seção 2.1.

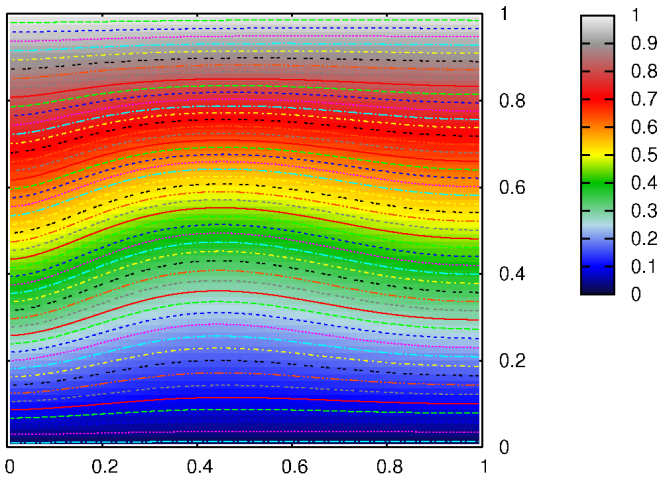
que os resultados calculados são similares àqueles apresentados em (GRIEBEL, 1998).



(a) *Streamlines.*



(b) Linhas de contorno da temperatura.



(c) *Heatlines.*

Figura 44: Solução no regime permanente ($t_{final} = 5000$) para o problema da convecção natural para uma malha 64×64 com $Re = 985,7$ e $Pr = 7$.

As Tabelas 10 e 11 apresentam, para diferentes tamanhos de malha, os tempos de

execução em segundos, o número total de iterações e o *speedup* obtidos para as diversas versões do algoritmo e tipos de dados *float* e *double* respectivamente. Como na maioria dos experimentos anteriores, observamos uma pequena diferença no número de iterações da versão sequencial em relação às versões CUDA, o aumento do *speedup* a medida que o número de células no domínio do problema aumenta e o impacto negativo do uso da memória *shared* no *speedup*. No entanto, vale a pena destacar que, mesmo para uma malha relativamente pequena, (256×256), conseguiu-se *speedups* da ordem de 20 e 16 vezes para os dados do tipos *float* e *double* respectivamente.

Tabela 10: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema da convecção natural para o tempo de computação $t_{final} = 5000$, $Re = 985,7$ e $Pr = 7$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
32×32	$5,48$	$7,56$	$8,69$	183025	183050	183050	$0,72$	$0,63$
64×64	$96,11$	$33,04$	$41,44$	780600	780885	780885	$2,91$	$2,32$
128×128	$1719,79$	$160,59$	$224,23$	3206450	3206550	3206550	$10,71$	$7,67$
256×256	$29561,77$	$1478,40$	$1762,79$	12956950	12872585	12872585	$20,00$	$16,77$

Tabela 11: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema da convecção natural para o tempo de computação $t_{final} = 5000$, $Re = 985,7$ e $Pr = 7$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_d	C_d	Cs_d	Seq_d	C_d	Cs_d	C_d	Cs_d
32×32	$6,05$	$7,57$	$7,62$	183030	183055	183055	$0,80$	$0,79$
64×64	$104,46$	$33,36$	$34,09$	780875	781160	781160	$3,13$	$3,06$
128×128	$1972,16$	$233,44$	$251,51$	3222800	3222900	3222900	$8,45$	$7,84$
256×256	$33104,63$	$1974,82$	$1980,65$	13136755	13094830	13094830	$16,76$	$16,71$

4.4.2 Problema tridimensional

O problema da convecção natural tridimensional, apresentado na Figura 45, consiste de um domínio cúbico com arestas de tamanho iguais a 1 *metro* preenchido com fluido. Impõem-se a condição de Não-deslizamento em todas as seis faces do cubo que formam o contorno do domínio. Os valores iniciais de pressão, temperatura e velocidades são os mesmos do caso bidimensional, sendo respectivamente: $p = 0$, $T = 0$, $u = 0$, $v = 0$ e $w = 0$. A temperatura da face esquerda é mantida com valor constante igual a $T_{esq} = 1,0$. De forma semelhante, a temperatura da face direita também é mantida com valor constante igual a $T_{dir} = 0,0$. As demais faces do contorno do domínio são consideradas adiabáticas, ou seja, não são consideradas trocas de calor nestes planos.

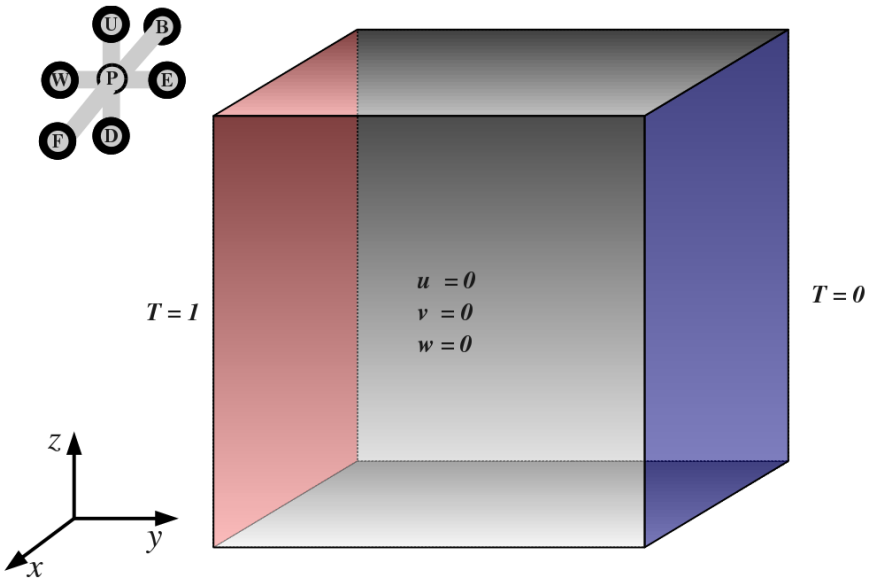
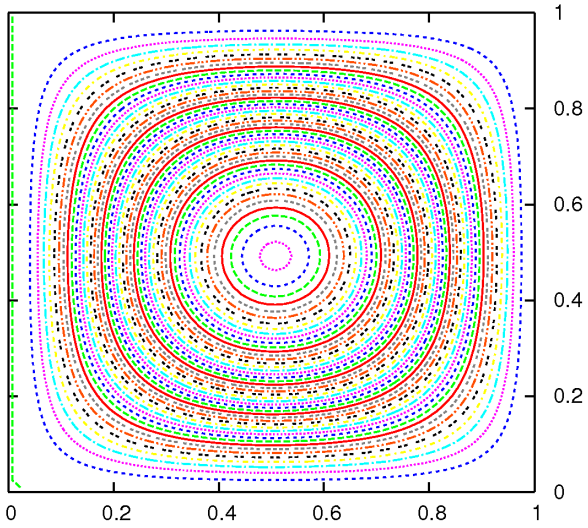


Figura 45: Descrição do domínio – convecção natural tridimensional.

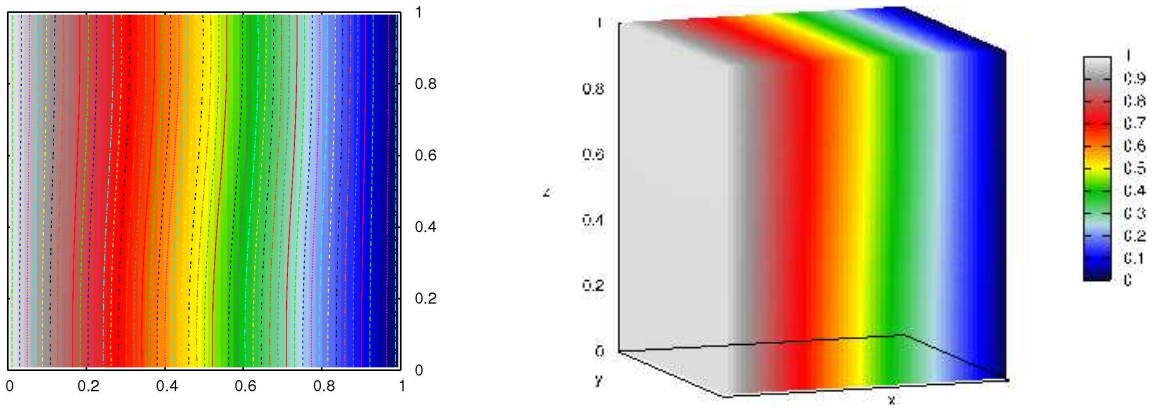
Os parâmetros físicos utilizados são os mesmos do caso bidimensional, sendo: $Re = 985,7$, $Pr = 7$, $\beta = 2,1e^{-4}$, $g_x = 0$, $g_y = 0$, $g_z = -9,706e^{-2}$ e $T_{final} = 5000$ *segundos*, valor este que garante o regime permanente. Para os experimentos com as versões CUDA, foram considerados os mesmos valores de números de blocos e números de *threads* apresentados no experimento da cavidade com superfície deslizante tridimensional, ou seja, fixamos os valores em `NUM_BLOCKS` = 30 e `NUM_THREADS` = 256, uma vez que as malhas utilizadas foram $32 \times 32 \times 32$, $64 \times 32 \times 64$ e $128 \times 32 \times 128$.

A Figura 46 apresenta, respectivamente, as *streamlines* para o plano xz com $y = 0.5$, as linhas de contorno da temperatura para o mesmo plano, a distribuição da temperatura no domínio tridimensional, as *heatlines* considerando o plano já citado e a distribuição do calor no domínio. Observamos que os resultados obtidos são similares aos apresentados em (GRIEBEL, 1998). Os resultados obtidos pela versão sequencial e pelas versões CUDA, independentemente do tipo de dado utilizado, *float* ou *double*, também apresentam resultados semelhantes.

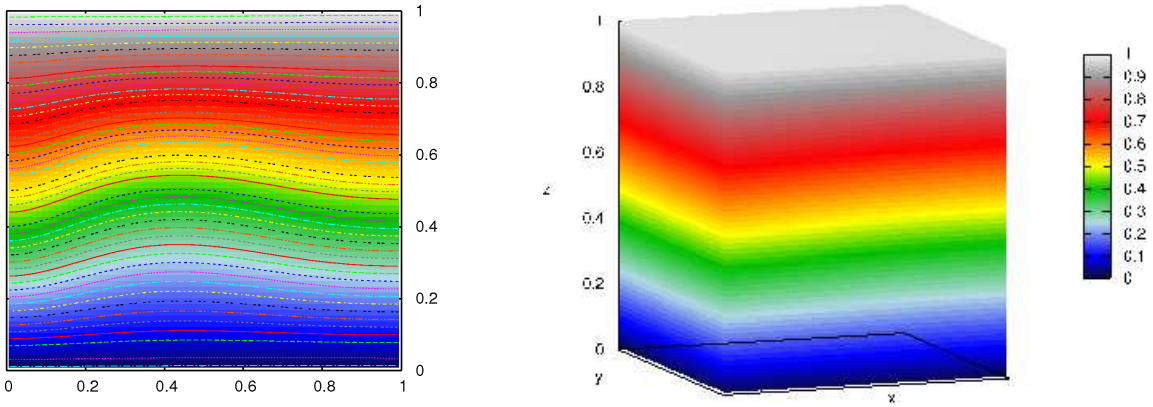
De forma similar ao caso bidimensional, apresentamos nas Tabelas 12 e 13 os tempos de execução em segundos, o número total de iterações e o *speedup* obtidos para os tipos de dados *float* e *double*, respectivamente, e diferentes malhas e versões do código. O comportamento das soluções com respeito ao número total de iterações, *speedup* e uso da memória *shared* não mudou em relação ao caso bidimensional. No entanto, podemos observar que o caso tridimensional alcançou *speedups* maiores quando comparadas com



(a) *Streamlines* para o plano xz com $y = 0.5$,



(b) Linhas de contorno da temperatura para o plano xz com $y = 0.5$ (esquerda) e distribuição da temperatura (direita).



(c) *Heatlines* para o plano xz com $y = 0.5$ (esquerda) e distribuição de calor (direita).

Figura 46: Solução no regime permanente ($t_{final} = 5000$) para o problema da convecção natural uma malha de tamanho $64 \times 32 \times 64$ com a versão CUDA *double* para $Re = 985,7$ e $Pr = 7$.

malhas bidimensionais de tamanhos correspondentes, como esperado. Por exemplo, para a malha 128×128 na versão CUDA *float*, o *speedup* atingido foi de $10,71$, enquanto que a malha $128 \times 32 \times 128$, com a mesma versão, obteve $14,23$ de *speedup*.

Tabela 12: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema da convecção natural tridimensional para $t_{final} = 5000$, $Re = 987,5$ e $Pr = 7$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
$32 \times 32 \times 32$	$403,03$	$124,34$	$139,46$	273940	273925	273925	$3,24$	$2,89$
$64 \times 32 \times 64$	$5839,53$	$839,30$	$934,63$	869425	869385	869385	$6,96$	$6,25$
$128 \times 32 \times 128$	$96845,64$	$6803,81$	$7391,54$	3324365	3324320	3324320	$14,23$	$13,10$

Tabela 13: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema da convecção natural tridimensional para $t_{final} = 5000$, $Re = 987,5$ e $Pr = 7$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_d	C_d	Cs_d	Seq_d	C_d	Cs_d	C_d	Cs_d
$32 \times 32 \times 32$	$405,26$	$139,06$	$150,64$	274030	274015	274015	$2,91$	$2,69$
$64 \times 32 \times 64$	$6331,62$	$913,64$	$987,64$	871380	871340	871340	$6,93$	$6,41$
$128 \times 32 \times 128$	$103597,90$	$7880,37$	$8352,83$	3312690	3312645	3312645	$13,15$	$12,40$

4.5 Problema da Convecção de Rayleigh-Bénard

Este experimento é similar ao problema apresentado na Seção 4.4, porém fixamos diferentes temperaturas para as fronteiras superior e inferior (JAGER, 1982; OZAL; HARA, 1995), como o apresentado na Figura 47. O domínio é definido com dimensões $(0;4)\times(0;1)$ e seu interior é preenchido com fluido. São consideradas condição de Não-deslizamento nas quatro fronteiras do fluido. A diferença de temperatura, por meio de ciclos de `resfriamento`, `compressão`, `aquecimento` e `expansão`, promove circulações no interior do domínio, denominadas células de Rayleigh-Bénard. Os valores iniciais para pressão, temperatura e velocidades são, respectivamente, $p = 0$, $T = 0$, $u = 0$ e $v = 0$. No entanto, mantem-se as fronteiras superior e inferior com temperaturas distintas e contantes iguais a $T_{sup} = -0,5$ e $T_{inf} = 0.5$, respectivamente. As demais fronteiras, esquerda e direita, são adiabáticas, ou seja, não há troca de calor com o meio. Os valores dos números de Reynolds e Prandtl, coeficiente de expansão termal e componentes da força de corpo (gravidade) empregados nos testes foram, respectivamente, $Re = 4365$, $Pr = 0.72$, $\beta = 3,4e^{-3}$, $g_x = 0$ e $g_y = -0.6432$. O tempo final de computação necessário para se garantir o regime permanente é igual a $t_{final} = 60000$ *segundos*. Para cada malha, nas versões CUDA, são disparados valores distintos de *threads* utilizando os mesmos critérios discutidos na Seção 4.1.

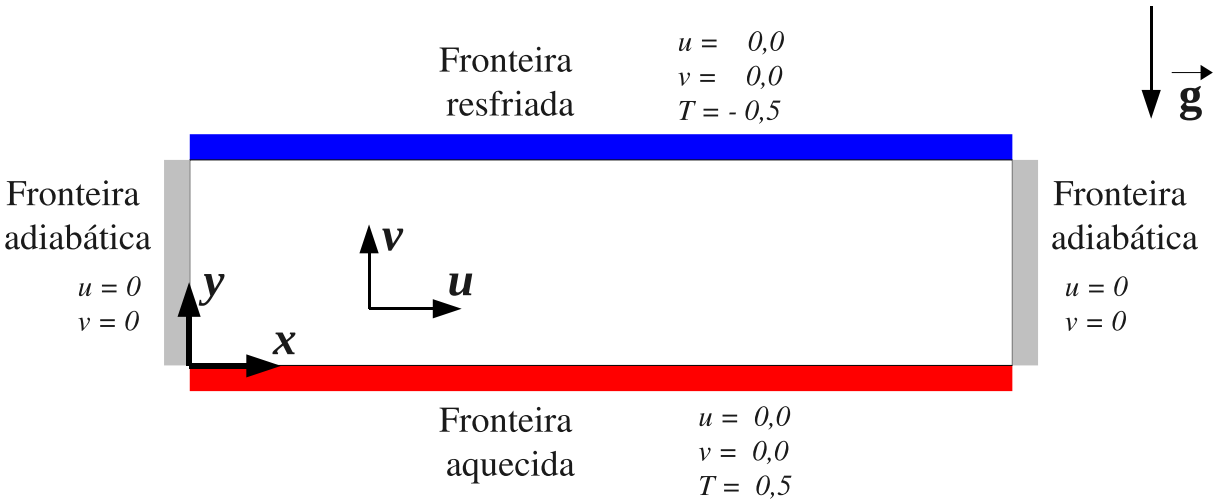
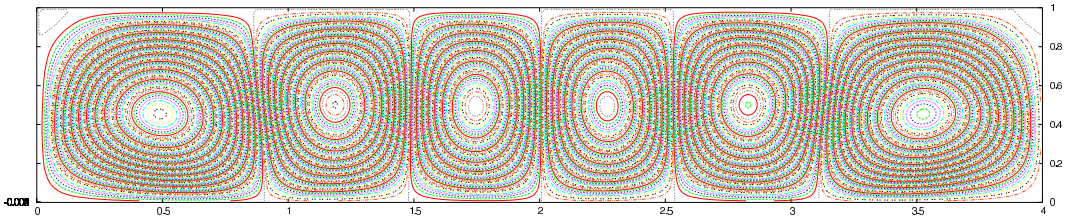


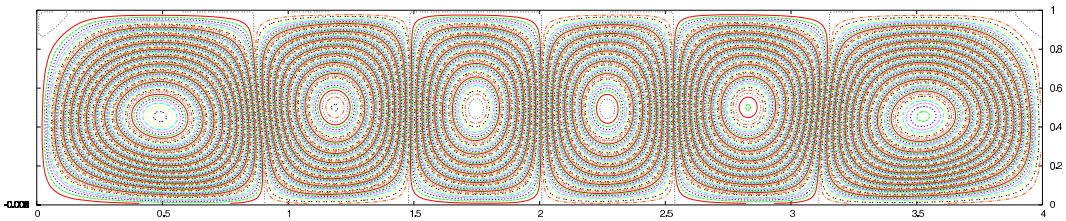
Figura 47: Descrição do problema – convecção de Rayleigh Bénard.

A Figura 48 apresenta as *streamlines* para as versões sequencial com tipo de dados *double*, CUDA com tipo de dados *float* e CUDA com tipo de dados *double* para uma malha de tamanho 256×64. Observamos que os resultados são virtualmente iguais, mesmo considerando os diferentes tipos de dados utilizados. As Figuras 49 e 50 apresentam,

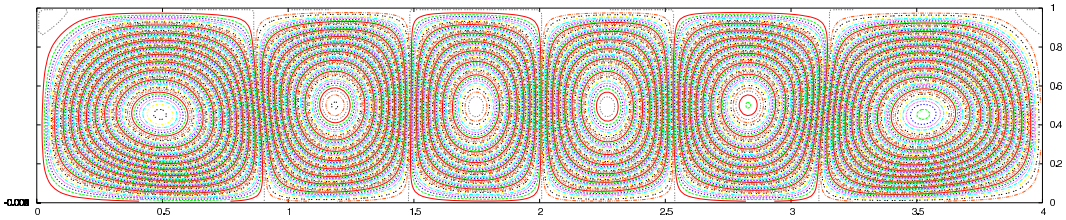
respectivamente, os campos de velocidade e os valores da temperatura para as mesmas versões e tamanho de malha da Figura 48. Observamos que a estrutura com seis células de Rayleigh-Bénard apresentadas condizem com aquelas apresentadas em (GRIEBEL, 1998).



(a) Versão sequencial.



(b) Versão CUDA com tipo de dados *float*.



(c) Versão CUDA com tipo de dados *double*.

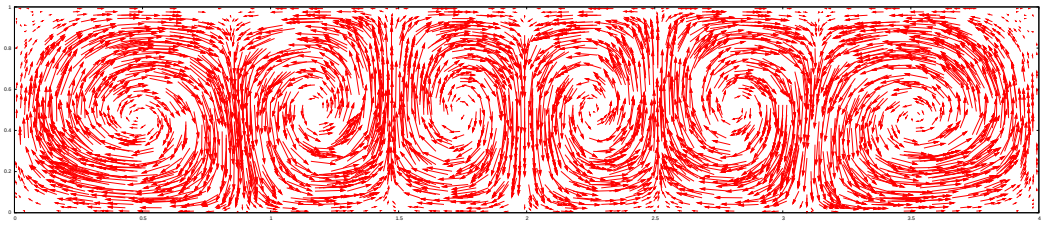
Figura 48: *Streamlines* para o problema da convecção Rayleigh-Bénard em uma malha 256×64 considerando $t_{final} = 60000$, $Re = 4365$ e $Pr = 0,72$.

A Tabela 14 apresenta os valores máximos para as velocidades obtidos no último passo de tempo considerando a versão sequencial com dado do tipo *float* (Seq_f), a versão CUDA com tipo de dado *float* (C_f) e uma malha 64×16 e os valores presentados em (GRIEBEL, 1998) onde foi considerado uma malha similar. Percebemos que os valores obtidos são condizentes com os apresentados em (GRIEBEL, 1998) uma vez que a tolerância utilizada no método iterativo é igual a 10^{-3}.

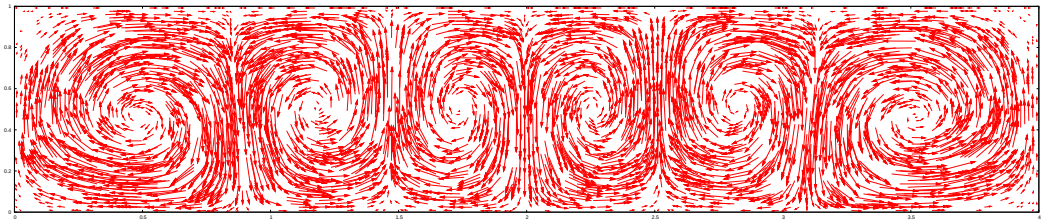
Tabela 14: Velocidades máximas obtidas considerando o último passo no tempo para o algoritmo sequencial proposto e o apresentado em (GRIEBEL, 1998).

Resultados	u_{max}	v_{max}
Seq_f	$0,013027$	$0,017620$
C_f	$0,013043$	$0,017580$
Griebel	$0,013040$	$0,017800$

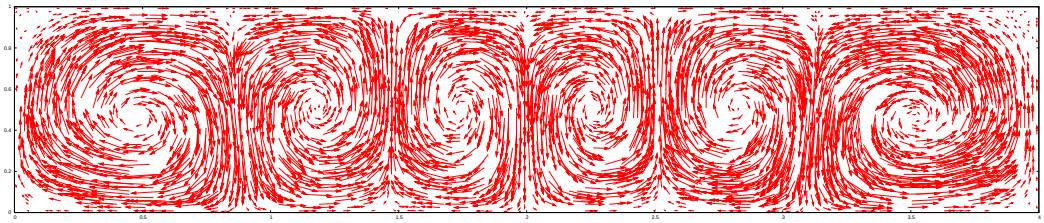
A Tabela 15 mostra, para três diferentes tamanhos de malhas, os valores de tempo



(a) Versão sequencial com tipo *double*.



(b) Versão CUDA com tipo *float*.



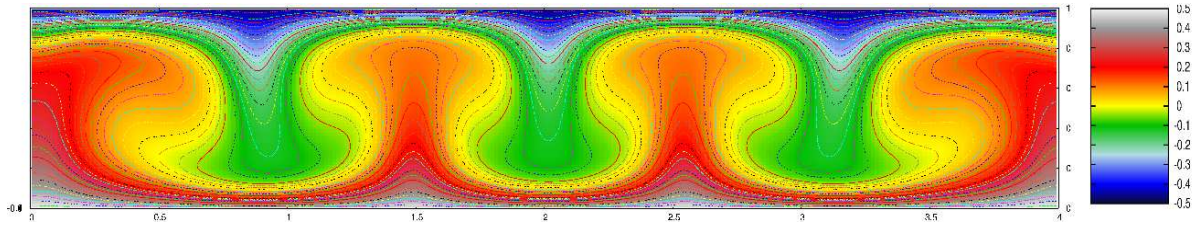
(c) Versão CUDA com tipo *double*.

Figura 49: Campo de velocidades para o problema da convecção Rayleigh-Bénard em uma malha 256×64 considerando $t_{final} = 60000$, $Re = 4365$ e $Pr = 0,72$.

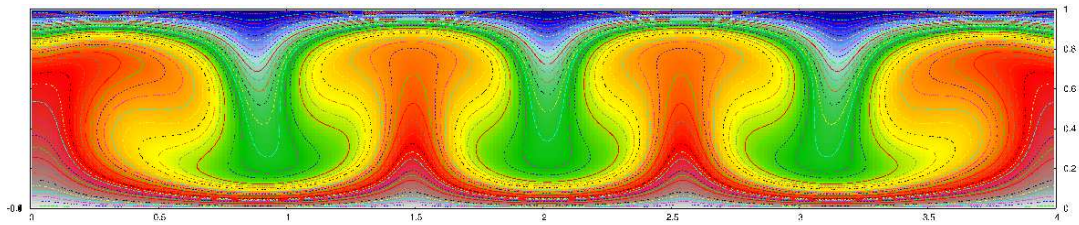
de execução, número máximo de iterações e *speedup* obtido para cada uma das seguintes versões do algoritmo: Seq_f, C_f e Cs_f. Todas essas versões consideram dados do tipo *float*. As mesmas considerações cabem nesse experimento com relação ao número total de iterações e *speedup*. Observamos ainda que o uso da memória *shared* não influenciou significativamente nos resultados pelas mesmas razões já relatadas anteriormente.

A Tabela 16 apresenta resultados similares aos anteriores, porém considerando as versões com dados do tipo *double*, ou seja, as versões Seq_d, C_d e Cs_d, e os mesmos tamanhos de malha. Destacamos mais uma vez o aumento do *speedup* com o aumento das malhas, devido ao melhor balanceamento de carga da arquitetura ocasionado pelo melhor aproveitamento da mesma pelas *threads*. Novamente o uso da memória *shared* não surtiu efeito no tempo de execução, devido aos fatores já relatados anteriormente.

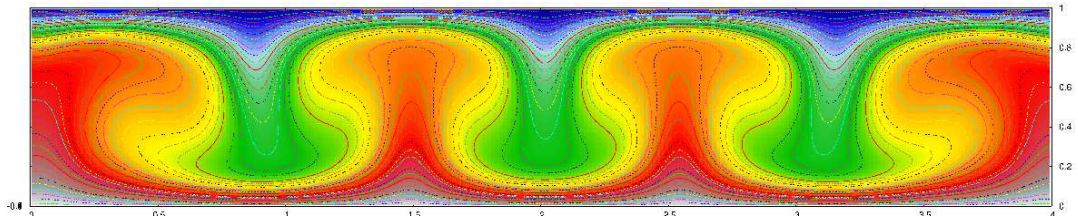
Considerando os impactos do uso de tipos de dados *float* e *double* pelas versões CUDA do algoritmo, observamos o melhor desempenho das versões com o uso de tipos *float*, o que já era esperado, pois, como já mencionado, o tipo *double* necessita de dois *cores* CUDA para ser executado, enquanto os demais tipos necessita de um *core*. No entanto os



(a) Versão sequencial.



(b) Versão CUDA com tipo de dados *float*.



(c) Versão CUDA com tipo de dados *double*.

Figura 50: Valores de temperatura para o problema da convecção Rayleigh-Bénard em uma malha 256×64 considerando $t_{final} = 60000$, $Re = 4365$ e $Pr = 0,72$.

resultados obtidos com tipos *double* são melhores do que o esperado. Isso ocorre porque, independentemente da versão executada, existe a execução de outros tipos de dados, como o tipo *int* utilizado nas estruturas de repetição (*loop*).

Tabela 15: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *float* – Problema da convecção de Rayleigh Bénard considerando $t_{final} = 60000$, $Re = 4365$ e $Pr = 0,72$.

Float	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_f	C_f	Cs_f	Seq_f	C_f	Cs_f	C_f	Cs_f
64×16	$4,81$	$7,26$	$6,77$	166850	166800	166800	$0,66$	$0,71$
128×32	$87,83$	$30,17$	$29,16$	723985	723765	723685	$2,91$	$3,01$
256×64	$1589,14$	$127,44$	$132,06$	2987625	2986680	2986680	$12,47$	$12,03$

Tabela 16: Tempo de execução, número de iterações e *speedup* para o algoritmo CUDA com tipo *double* – Problema Problema da convecção de Rayleigh Bénard considerando $t_{final} = 60000$, $Re = 4365$ e $Pr = 0,72$.

Double	Tempo			Número de Iterações			*Speedup*	
Malha	Seq_d	C_d	Cs_d	Seq_d	C_d	Cs_d	C_d	Cs_d
64×16	$5,36$	$7,10$	$7,02$	166930	166780	166780	$0,75$	$0,76$
128×32	$98,49$	$30,37$	$30,41$	723060	723085	723085	$3,24$	$3,24$
256×64	$1902,33$	$181,49$	$188,70$	3008370	3008920	3008920	$10,48$	$10,08$

5 Conclusões

Neste trabalho foi realizado um estudo do uso de GPUs em aplicações de mecânica do fluidos computacional envolvendo escoamentos viscosos e incompressíveis descritos pelas equações de Navier-Stokes, acopladas à equação de transporte de calor por convecção e condução. Realizamos experimentos para cinco problemas clássicos da literatura, considerando casos bidimensionais e tridimensionais e testes de desempenho para: (i) cavidade com superfície deslizante, (ii) escoamento sobre um degrau, (iii) escoamento laminar com obstáculo circular, (iv) convecção natural e (v) convecção Rayleigh-Bénard. Uma formulação implícita para a pressão e explícita para as velocidades e temperatura do método das diferenças finitas foi implementada. Na resolução dos sistemas lineares resultantes para a pressão foi considerado o método de coloração Red-Black para as células da malha juntamente com o método iterativo SOR, denominado Red-Black-SOR. Com a escolha desse método foi possível garantir uma sequência de execução das células de forma semelhante, tanto no algoritmo sequencial quanto no algoritmo paralelo.

A plataforma *Compute Unified Device Architecture* (CUDA) e a linguagem de programação C para a programação em placas gráficas NVIDIA foram os componentes computacionais utilizados. Todos os experimentos foram realizados na placa NVIDIA GeForce GTX 480, utilizando seis versões do algoritmo que consideram a arquitetura utilizada (CPU, GPU/CUDA), o tipo de dados utilizado em cada versão (*float*,*double*) e a influência do uso da memória *shared*. Nos experimentos tridimensionais, observamos que a execução de problemas em malhas superiores a $512 \times 32 \times 512$ ficaram inviáveis por dois motivos: (i) o critério de adaptatividade do tamanho do passo no tempo ser ainda mais restritivo no caso tridimensional, ocasionando um enorme tempo computacional necessário para a execução do algoritmo serial; e (ii) a limitação do tamanho da memória RAM da GPU, uma vez que o tamanho do espaço de memória para tal malha se aproxima de 1GB, mesmo utilizando uma estratégia otimizada para guardar apenas as diagonais não nulas do sistema linear.

Realizamos experimentos envolvendo dois tipos distintos de dados, um com precisão

simples (*float*) e outro com precisão dupla (*double*). Não encontramos diferenças significativas nos resultados quando comparamos os experimentos para os dois tipos de dados, *float* e *double*, e tolerância no método iterativo de 10^{-3}. Observamos que os resultados com tipo *float* forneceram melhores *speedups* do que aqueles com tipo *double*. Isto já era esperado, uma vez que o tipo *double* necessita de dois *cores* CUDA em sua execução. Porém, em todos os experimentos os resultados com tipo *double* foram melhores que o esperado, ou seja, superiores a metade do valor do *speedup* obtido com o tipo *float*. Acreditamos que a utilização conjunta de outro tipos de dados nas versões *double*, como o tipo *int* utilizado nos contadores das estruturas de *loop*, é a responsável por esse sobre-ganho. Observamos ainda, que para malhas pequenas, a diferença entre os *speedups* das versões *float* e *double* foi pequena, porém essa diferença aumenta a medida que a o número de células aumenta. Se uma aplicação específica necessitar de uma maior precisão nos cálculos, é possível perceber nos testes executados que o uso de dados do tipo *double* trazem mesmo assim um ganho computacional muito significativo; por exemplo, algo da ordem de até 21 vezes no problema da cavidade com superfície deslizante.

Em todos os cinco problemas implementados observamos o aumento do *speedup* a medida que a malha é refinada, ou seja, a medida que o número de células do domínio cresce. Tal aumento é justificado pelo melhor aproveitamento da arquitetura com um maior número de células. Os experimentos demonstram que a utilização de GPUs foram capazes de proporcionar *speedups* da ordem de até 25 vezes para tipos de dados *float* e da ordem de até 21 vezes para tipos de dados *double* (cavidade tridimensional). De uma forma geral, os resultados tridimensionais apresentaram resultados superiores quando comparados aos mesmos problemas bidimensionais para malhas correspondentes. Por exemplo, considerando o problema da convecção natural, para a malha 128×128 no caso *float*, o speedup obtido foi de $10,71$, enquanto que na malha $128 \times 32 \times 128$ foi de $14,23$. Destacamos que, mesmo nos experimentos que necessitaram de uma intensa troca de dados entre a CPU e a GPU – como por exemplo, o problema do escoamento laminar com obstáculo circular – obtivemos *speedups* da ordem de 13 vezes com dados *float* e 22 para dados com tipo *double*. Assim, é evidente que a utilização de GPUs é uma alternativa para a redução do tempo computacional envolvido na resolução de problemas de mecânica dos fluidos computacional discretizados pelo método das diferenças finitas.

Apenas em alguns tamanhos de malha do experimentos da cavidade tridimensional ($128 \times 32 \times 128$ com tipo *float* e $256 \times 32 \times 256$ com tipo *double*) a utilização da memória *shared* demostrou vantagem quando utilizada como uma memória rápida de reuso de

dados. Fatores como a alta velocidade do barramento da placa utilizada nos experimentos, a existência de memórias *cache* L1 e L2 nesta mesma placa e o pequeno número de pontos de reuso de dados encontrados no código podem ter contribuído para o baixo benefício proveniente do uso da memória *shared*. Porém, gostaríamos de destacar dois fatores: (*i*) a memória *shared* é uma importante e fácil ferramenta para a cooperação entre as *threads* de um mesmo bloco, como no código de redução do valor máximo, onde foi minimizado a necessidade de escrita no vetor da memória global do *device*; (*ii*) o desempenho da implementação apresentada nesse trabalho usando a memória *shared* é diretamente relacionado a placa utilizada. No trabalho (MENENGUCI *et al.*, 2010), onde foi utilizada a mesma formulação e implementação deste trabalho, também foi possível observar uma discreta melhora do *speedup* utilizando a memória *shared* em comparação com a memória global para testes realizados na placa NVIDIA Tesla C1060.

Como trabalhos futuros sugerimos: (*i*) o estudo da utilização de métodos iterativos não estacionários para a resolução dos sistemas lineares resultantes; (*ii*) a extensão do algoritmo para outras arquiteturas paralelas, como máquinas *multi-cores*, clusters de CPUs e máquinas multi-GPUs ou clusters de GPUs; (*iii*) o estudo do impacto da utilização de um esquema misto de precisão (*float* e *double*), como em (KURZAK; DONGARRA, 2007) e (*iv*) a divisão da malha para a execução de problemas maiores que o tamanho da memória da GPU. Pretendemos ainda, analisar o impacto do uso de GPUs em outros métodos de discretização, como por exemplo o método dos elementos finitos.

Referências

ASANOVIC, K.; BODIK, R.; CATANZARO, B. C.; GEBIS, J. J.; HUSBANDS, P.; KEUTZER, K.; PATTERSON, D. A.; PLISHKER, W. L.; SHALF, J.; WILLIAMS, S. W.; YELICK, K. A. *The Landscape of Parallel Computing Research: A View from Berkeley*. California, Dec 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

BULGARELLI, U.; CASULLI, V.; GREENSPAN, D. *Pressure Methods for the Numerical Solution of Free Surface Fluid Flows*. Swansea: Pineridge Press, 1984.

CASULLI, V. Numerical simulation of free surface thermally incluenced flows for nonhomogeneous fluids. *Appl. Math. Comp.*, v. 8, p. 261–280, 1981.

DE SOUZA, A. F. *Computing Unified Device Architecture (CUDA) A Mass-Produced High Performance Parallel Computing Platform*. SBAC-PAD'2008. Tutorial. Campo Grande, MS, Brasil, 2008.

EATON, B. Analysis of laminar vortex shedding behind a circular cylinder by computer-aided flow visualization. *J. Fluid Mech.*, v. 180, p. 117–145, 1987.

ERTURK, E.; CORKE, T.; COKCOL, C. Numerical solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *International Journal for Numerical Methods in Engineering*, v. 48, n. 7, p. 747–774, jul. 2005.

FERNANDES, E.; BARBOSA, V.; RAMOS, F. Instruction usage and the memory gap problem. In: *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2002. (SBAC-PAD '02), p. 169–175. ISBN 0-7695-1772-2. Disponível em: <http://portal.acm.org/citation.cfm?id=862895.883870>.

FORTUNA, A. de O. *Técnicas Computacionais para Dinâmica dos Fluidos: Conceitos Básicos e Aplicações*. São Paulo, SP, Brasil: Edusp - Editora da Universidade de São Paulo, 2000. ISBN 85-314-0526-2.

GARTLING, D. A test problem for outflow boundary conditions— flow over a backward-facing step. *J. Numer. Methods Fluids*, v. 11, p. 511–537, 1990.

GHIA, U.; GHIA, K.; SHIN, C. High-re solutions for incompressible flow using te navier-stokes and a multigrid method. *J. Comput. Phys.*, v. 48, p. 387–411, 1982.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. gprof: a call graph execution profiler. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 39, n. 4, p. 49–57, 2004. ISSN 0362-1340.

GRAY, D.; GIORGINI, A. On the validity of the boussinesq aproximation for liquids and gases. *Int. J. Heat Mass Transfer*, v. 19, p. 545–551, 1976.

GRIEBEL, M. *Numerical Simulation Fluid Dynamics*. Philadelphia, PA: SIAM, 1998.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Fourth edition. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2006.

IRWIN, M. J.; SHENA, J. P. Revitalizing computer architecture research. In: *Third in a Series of CRA Conferences on Grand Research Challenges in Computer Science and Engineeringg*. Monterey Bay, California: Washington, Computing Research Association, 2005.

JACOBSEN, D. A.; THIBAULT, J. C.; SENOCAK, I. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In: *Proceedings of the 48th AIAA Aerospace Sciences Meeting*. Orlando, Florida: American Institute of Aeronautics and Astronautics (AIAA), 2010.

JAGER, W. *Oszillatorische und turbulente Konvektion*. Dissertação (Mestrado) — Universitat Karlsruh, 1982.

KAIKTSIS, L.; KARNIADAKIS, G.; ORSZAG, S. Onset of three-dimensionality, equilibra, and early transition in flow over a backward-facing step. *J. Fluid Mech.*, v. 231, p. 501–528, 1991.

KIMURA, S.; BEJAN, A. The heatline visualization of convective heat transfer. *Trans. ASME: J. Heat Transfer*, v. 105, p. 916–919, 1983.

KONGETIRA, P.; AINGARAN, K.; OLUKOTUN, K. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, v. 25, n. 2, p. 21–29, 2005.

KURZAK, J.; DONGARRA, J. Implementation of mixed precision in solving systems of linear equations on the cell processor: Research articles. *Concurr. Comput. : Pract. Exper.*, John Wiley and Sons Ltd., Chichester, UK, v. 19, p. 1371–1385, July 2007. ISSN 1532-0626. Disponível em: <http://dl.acm.org/citation.cfm?id=1272430.1272431>.

LINDHOLM, E.; NICKOLLS, J.; OBERMAN, S.; MONTRYM, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, IEEE Computer Society, Los Alamitos, CA, USA, v. 28, p. 39–55, 2008. ISSN 0272-1732.

MALISKA, R. C. *Transferência de calor e mecânica dos fluidos computacional*. Rio de Janeiro: LTC, 1995.

MANFERDELLI, J. L. The many-core inflection point for mass market computer systems. *CTWatch (Cyberinfrastructure Technology Watch) Quartely*, v. 3, p. 11–17, 2007.

MCCALPIN, J.; MOORE, C.; HESTER, P. The role of multicore processors in the evolution of general-purpose computing. *CTWatch (Cyberinfrastructure Technology Watch) Quartely*, v. 3, n. 1, p. 18–30, 2007.

MENENGUCI, W. S.; VALLI, A. M. P.; CATABRIGA, L.; VERONESE, L. Um algoritmo cuda em diferenças finitas para a discretização das equações de Navier-Stokes. *Mecánica Computacional*, XXIX, n. 71, p. 7067–7084, 2010.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 8, p. 1–4, 1965.

NICKOLLS, J.; BUCK, I.; GARLAND, M.; SKADRON, K. Scalable parallel programming with cuda. *ACM Queue*, v. 6, n. 2, p. 14–53, 2008.

NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. Santa Clara, CA, 2008.

NVIDIA. *Whitepaper – NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Santa Clara, CA, 2009.

NVIDIA. *CUDA API Reference Manual – Version 4.0*. Santa Clara, CA, fev. 2011.

NVIDIA. *NVIDIA CUDA C Programming Guide 4.0*. Santa Clara, CA, 2011.

OZAL, H.; HARA, T. Numerical analysis for oscillatory natural convection of low prandtl number fluid heated from below. *Numer. Heat Transfer A*, v. 27, p. 307–318., 1995.

PINELLI, A.; VACCA, A. Chebyshev collocation method and multidomain decomposition for the incompressible navier-stokes-equations. *internat. J. Number Methods Fluids*, v. 18, p. 781–799, 1994.

RANDIMA, F.; MARK, K. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston: Addison-Wesley Longman Publishing Co., 2003.

SAABAS, H. J. *A control volume finite element method for three-dimensional, incompressible, viscous fluid flow*. Tese (Doutorado) — McGill University, Montreal, Que, Canada., 1991.

SENOCAK, I.; THIBAULT, J.; CAYLOR, M. Rapid-response urban CFD simulations using a GPU computing paradigm on desktop supercomputer. In: *Proceedings of the Eighth Symposium on the Urban Environment*. Phoenix, Arizona: American Meteorological Society, 2009.

SHINN, A. F.; VANKA, S. P. Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit. *ASME Conference Proceedings*, ASME, v. 2009, n. 43864, p. 125–133, 2009. Disponível em: <http://link.aip.org/link/abstract/ASMECP/v2009/i43864/p125/s1>.

TENDLER, J.; DODSON, J.; JR., J. F.; LE, H.; SINHAROY, B. Power4 system microarchitecture. *IBM Journal of Research and Development*, v. 46, n. 1, p. 5–26, 2002.

THIBAULT, J. C. *Implementation of a cartesian grid incompressible Navier-Stokes solver on multi-GPU desktop plataforms using CUDA*. Dissertação (Mestrado) — Boise State University, april 2009.

THIBAULT, J. C.; SENOCAK, I. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. In: *Proceedings of the 7th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*. Orlando, Florida: American Institute of Aeronautics and Astronautics (AIAA), 2009.

TRITTON, D. Experiments on the flow past a circular cylinder at low reynolds numbers. *J. Fluid Mech.*, v. 6, p. 547–567, 1959.

VERONESE, L.; LIMA, L. M.; DE SOUZA, A. F.; CATABRIGA, L. Evaluation of two parallel finite element implementations of the time-dependent advection diffusion problem: CUDA $\times$ MPI. In: *Proceedings of the Supercomputing 2010 - Poster Session*. New Orleans, Lousiana: Supercomputing, 2010.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, v. 23, n. 1, p. 20–24, 1995.