Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 7067-7084 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

UM ALGORITMO CUDA EM DIFERENÇAS FINITAS PARA A DISCRETIZAÇÃO DAS EQUAÇÕES DE NAVIER-STOKES

Wesley dos S. Menenguci[a], Andrea M. P. Valli[a], Lucia Catabriga[a] e Lucas Veronese[a]

[a]*Laboratório de Computação de Alto Desempenho, Departamento de Informática, Universidade Federal do Espírito Santo, Av. Fernando Ferrari 514, 29075-910, Vitória, ES, Brazil, tiwesley@lcad.inf.ufes.br, avalli@inf.ufes.br, luciac@inf.ufes.br, lucas.veronese@lcad.inf.ufes.br (http://www.lcad.inf.ufes.br)*

Palavras Chave: C+CUDA, memória compartilhada, esquema Red-Black SOR, equações de Navier-Stokes.

Resumo. As unidades de processamento gráfico (*Graphics Processing Unit* – GPU) surgiram como um poderoso dispositivo computacional e a plataforma *Compute Unified Device Architecture* (CUDA) é um ambiente adequado que permite a tradução quase direta de um código C para uma GPU. Especializada inicialmente em processamento gráfico, a GPU vem sendo designada à otimização de cálculos lógicos e aritméticos beneficiando diversas áreas de pesquisa com a redução do tempo de computação. O objetico deste trabalho é mostrar como aplicações em mecânica dos fluidos, discretizadas pelo método das diferenças finitas, podem lucrar bastante com esta tecnologia. Implementações paralelas na GPU em C+CUDA das equações de Navier-Stokes são comparadas com uma versão sequencial implementada na CPU em C. É utilizada uma formulação em diferenças finitas implícita-explícita, sendo o algoritmo caracterizado como sendo explícito nas velocidades e implícito na pressão. 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. Além disso, é discutido neste trabalho os benefícios e dificuldades da utilização das memórias compartilhada (*shared memory*) e global (*global memory*) existentes na GPU. O algoritmo C+CUDA foi verificado para um conjunto de problemas conhecidos da literatura e o tempo de processamento comparado com o mesmo algoritmo implementado em C. Os resultados numéricos mostraram que o tempo de processamento pode ser reduzido significamente utilizando C+CUDA.

1 INTRODUÇÃO

Avanços científicos e inovações em hardware e software permitiram 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 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 e Patterson (2006). Até recentemente, a tendência na indústria de processadores era empregar transistores adicionais na implementação de CIs contendo sistemas com um único processador cada vez mais poderoso. No entanto, existem três obstáculos para a continuidade desta tendência: (i) o consumo de energia e a consequente necessidade da dissipação do calor (*the Power Wall*), Asanovic et al. (2006); (ii) a crescente latência da hierarquia de memória (*the Memory Wall*), Wulf e McKee (1995); e (iii) as dificuldades associadas a uma maior exploração do paralelismo no nível de instrução (*the ILP Wall*), Asanovic et al. (2006). David Patterson sumarizou estes três problemas na expressão: *the Power Wall + the Memory Wall + the ILP Wall = the Brick Wall for serial performance*, Irwin e Shena (2005). Em face dos obstáculos mencionados, a indústria mundial de computadores mudou de curso em 2005 com o desenvolvimento dos múltiplos núcleos de processamento (*multi-core*). Atualmente, processadores *multi-core* tiram proveito do número de transistores disponível em um único CI para disponibilizar hardware próprio para a exploração do paralelismo de grão grosso existente nas aplicações, De Souza (2008). 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).

Para aplicações em mecânica dos fluidos computacional destacam-se os trabalhos: Shinn e Vanka (2009); Senocak et al. (2009); Klockner et al. (2009);Thibault1 e Senocak (2009); Cohen e Molemaker (2010); Jacobsen et al. (2010); Veronese et al. (2010). Shinn e Vanka (2009) abordam a utilização de GPU para resolução de problemas em fluidos turbulentos tridimensionais utilizando técnicas multigrid. As discusões apresentadas mostram que o uso de precisão simples não influenciou nos resultados obtidos. Os autores apresentam um estudo sobre a latência da memória global do dispositivo e a utilização da memória compartilhada. Senocak et al. (2009); Thibault1 e Senocak (2009) estudam 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*). Uma discussão sobre precisão simples e dupla em GPUs é feito por Cohen e Molemaker (2010). Os resultados com GPU apresentam ganho utilizando ambas as precisões, porém o desempenho é menor quando a precisão dupla é utilizada. Recentemente, Jacobsen et al. (2010); Veronese et al. (2010) 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 lucrar bastante com esta tecnologia. Foi comparada implementações paralelas na GPU em C+CUDA das equações de Navier-Stokes com uma versão sequencial implementada na 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. Além disso, foi discutido os benefícios e as dificuldades da utilização das memórias compartilhada (*shared memory*) e global (*global memory*) existentes na GPU. Este trabalho está organizado como a seguir. Na Seção 2 é apresentado o tratamento numérico utilizado para as equações de Navier-Stokes. A Seção 3 introduz os conceitos do modelo de arquitetura e programação em CUDA. Na Seção 4 são discutidos e apresentados os principais algoritmos paralelos utilizados no trabalho, bem como a utilização da memória compartilhada. Em seguida, a Seção 5 são apresentados os resultados obtidos para os testes computacionais. Por fim, a Seção 6 apresenta as conclusões e considerações finais do trabalho.

2 O TRATAMENTO NUMÉRICO DAS EQUAÇÕES DE NAVIER-STOKES

2.1 A discretização por diferenças finitas

Os problemas tratados neste trabalho envolvem a simulação de escoamentos bidimensionais transientes de fluidos viscosos incompressíveis, governados pelas equações de conservação de massa e quantidade de movimento para fluidos Newtonianos com viscosidade e densidade constante. As equações de Navier-Stokes e continuidade que representam este tipo de escoamento podem ser escritas como:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{1}$$

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \tag{2}$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (v^2)}{\partial y} - \frac{\partial (uv)}{\partial x} + g_y \tag{3}$$

onde u e v são os componentes horizontal e vertical da velocidade, p é a pressão, g_x e g_y são os componentes da força, $Re = (\rho u_\infty L_\infty)/\mu$ é o número adimensional de Reynolds, ρ, u_∞ e L_∞ são constantes escalares (ou seja: densidade do fluido, velocidade característica e o comprimento característico, respectivamente) e μ é a viscosidade dinâmica. Para completar a formulação matemática do problema são necessárias as condições iniciais e as condições de contorno.

O tratamento numérico das equações de Navier-Stokes é baseado em um esquema de diferenças finitas como sugerido em Griebel (1998). O domínio é discretizado em $imax$ células na direção x e $jmax$ células na direção y, todas de mesmo tamanho. A região é discretizada usando uma malha deslocada (*staggered*), Figura 1, no qual 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. Esta disposição das incógnitas evita possíveis oscilações na pressão que poderiam ocorrer caso as três incognitas u, v e p fossem avaliadas no mesmo ponto na malha.

Uma vez que os termos convectivos, $\frac{\partial (u^2)}{\partial x}$, $\frac{\partial (v^2)}{\partial y}$, $\frac{\partial (uv)}{\partial x}$ e $\frac{\partial (uv)}{\partial y}$, nas equações de momento se tornam dominante para números de Reynolds grandes ou velocidades altas, problemas de

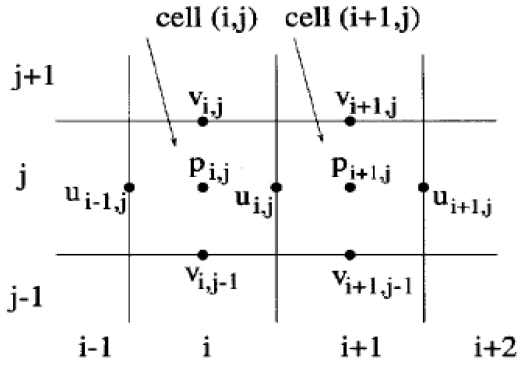

Figura 1: Malha deslocada.

estabilidade podem ocorrer quando o espaçamento na malha escolhido for muito grosseiro. Sendo assim, a discretização espacial das derivadas requer uma combinação de diferenças finitas centrais e *donor-cell* para manter a estabilidade. No caso das derivadas parciais de primeira ordem $\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial y}$ e de segunda ordem $\frac{\partial^2 u}{\partial x^2}$, $\frac{\partial^2 u}{\partial y^2}$, $\frac{\partial^2 v}{\partial x^2}$, $\frac{\partial^2 v}{\partial y^2}$, que formam o termo difusivo, pode ser utilizado diferenças finitas centrais com a metade do tamanho da malha. Maiores detalhes da discretização podem ser encontrados em Griebel (1998). Para obter a discretização no tempo das equações de momento (2) e (3), é utilizado o método de Euler para a aproximação das derivadas $\frac{\partial u}{\partial t}$ e $\frac{\partial v}{\partial t}$. Definindo as funções

$$F = u^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \right], \tag{4}$$

$$G = v^{(n)} + \delta t \left[\frac{1}{Re} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial (v^2)}{\partial y} - \frac{\partial (uv)}{\partial x} + g_y \right], \tag{5}$$

onde n significa a discretização no tempo. Considerando as definições dadas em (4) e (5) é possível obter as aproximações para as equações de momento

$$u^{(n+1)} = F_{i,j}^{(n)} - \frac{\delta t}{\delta x} \left[p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)} \right], \quad i = 1, \cdots, i_{max} - 1, \quad j = 1, \cdots, j_{max}, \tag{6}$$

$$v^{(n+1)} = G_{i,j}^{(n)} - \frac{\delta t}{\delta x} \left[p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)} \right], \quad i = 1, \cdots, i_{max}, \quad j = 1, \cdots, j_{max} - 1, \tag{7}$$

no qual podem ser caracterizadas como sendo explícitas nas velocidades e implícitas para a pressão, isto é, a velocidade no passo no tempo t_{n+1} pode ser calculada quando a correspondente pressão é conhecida. Substituindo as equações para as velocidades (6) e (7) na equação de continuidade (1) , é possível obter a equação de Poisson para a pressão $p^{(n+1)}$ no tempo t_{n+1}

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2}$$

$$= \frac{1}{\delta t} \left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y} \right), \quad i = 1, \cdots, i_{max}, \quad j = 1, \cdots, j_{max} \tag{8}$$

no qual requer valores na fronteira para a pressão. Nesse caso é assumido que $p_{0,j} = p_{1,j}$, $p_{i_{max}+1,j} = p_{i_{max},j}$, $p_{i,0} = p_{i,1}$, $p_{i,j_{max}+1} = p_{i,j_{max}}$, com $i = 1, \cdots, i_{max}$ e $j = 1, \cdots, j_{max}$. Além disso, são necessários os valores de F e G na fronteira para calcular o lado direito de

(8). Então, é definido $F_{0,j} = u_{0,j}$, $F_{i_{max},j} = u_{i_{max},j}$, $G_{i,0} = v_{i,0}$ e $G_{i,j_{max}} = v_{i,j_{max}}$, com $i = 1, \cdots, i_{max}$ e $j = 1, \cdots, j_{max}$.

Como resultado, é preciso resolver um sistema linear de equações para a pressão (8) contendo $i_{max} \times j_{max}$ incógnitas. Neste trabalho é utilizado o método *Red-Black Successive Over-Relaxation* (Red-Black-SOR) na resolução dos sistemas lineares na forma sequencial e paralela. Além disso, um controle adaptativo do tamanho do passo no tempo, baseado na condição Courant-Friedrichs-Lewy (CFL), é utilizado para evitar a geração de oscilações e garantir estabilidade do método numérico. O novo passo no tempo é dado por

$$\delta t = \tau_{min} \left(\frac{Re}{2} \left(\frac{1}{\delta x} + \frac{1}{\delta y} \right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|} \right). \tag{9}$$

onde $\tau \in (0, 1]$ é um fator de segurança. Resumindo, o processo sequencial global consiste dos passos mostrados na Figura 2. O algoritmo paralelo implementado em CUDA será discutido na próxima seção.

1. Set t = 0, n = 0
2. init_UVP(); // Initializa as variáveis: u, v, p
3. While $t < t_{end}$
   (a) comp_delt(); // calcula δt (use (9) se o algoritmo de controle for usado)
   (b) set_BondaryCond(); // valores de fronteira u e v
   (c) comp_FG(); // calcula $F^{(n)}$ e $G^{(n)}$ de acordo com (4) e (5)
   (d) poisson_system(); // constroi o sistema linear
   (e) MetodoSOR_RedBlack(); // resolve o sistema usando Red-Black-SOR
   (f) comp_UV(); // calcula $u^{(n+1)}$ e $v^{(n+1)}$ de acordo com (6) e (7)
   (g) $t = t + \delta t$;
   (h) $n = n + 1$;

Figura 2: Algoritmo Sequencial.

3 O SISTEMA DE COMPUTAÇÃO 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 do *host*, a tradicional unidade de processamento central (CPU), e de um ou vários *devices* (GPU), coprocessadores da CPU capazes de executar dezenas de milhares de *threads* simultaneamente. Trechos da aplicação com grandes demandas computacionais podem ser traduzidos para CUDA e executados em GPUs. 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.

Um *kernel* comanda a execução, na GPU, de um conjunto de *threads*, que são organizadas em grades (*grids*) de blocos de *threads* (*thread blocks*). Uma *grid* é um conjunto de *thread*

blocks que executam independentemente, enquanto que, um *thread block* é um conjunto de *threads* que podem cooperar por meio de sincronização do tipo barreira e acesso compartilhado a um espaço de memória exclusivo de cada *thread block*. Em C+CUDA existem três tipos de funções: *host*, *kernel* e *device*. As funções *host* são chamadas e executadas somente pela CPU e são semelhantes as funções implementadas em C. As funções *kernel* são chamadas pelo *host* e executadas somente pela GPU. Possuem um qualificador `__global__`, que deve ser inserido antes do tipo de retorno da função que é sempre do tipo `void`. Por fim, as funções *device* são chamadas e executadas somente pela GPU. Neste tipo também existe um qualificador definido como `__device__`, que deve ser declarado antes do tipo de retorno da função; e neste caso, é permitido o retorno de qualquer tipo.

O grande número de *threads* por processador, as centenas de processadores por GPU e o baixo custo do chaveamento entre *threads*, ocultam a latência de memória global da GPU viabilizando o modelo de computação massivamente paralela de alto desempenho. De uma forma geral, a arquitetura de uma GPU CUDA é composta por vários processadores (*Streaming Processors* - SP) para cálculos de inteiros e ponto flutuante, agrupados em *Streaming Multiprocessors* (SM). Além disso, CUDA possui uma profunda hierarquia de memória, que é composta por: registradores, memória compartilhada (*shared memory*), memória de constantes (*constant memory*), memória de textura (*texture memory*) e memória global do dispositivo (*device memory*), representadas na Figura 3.

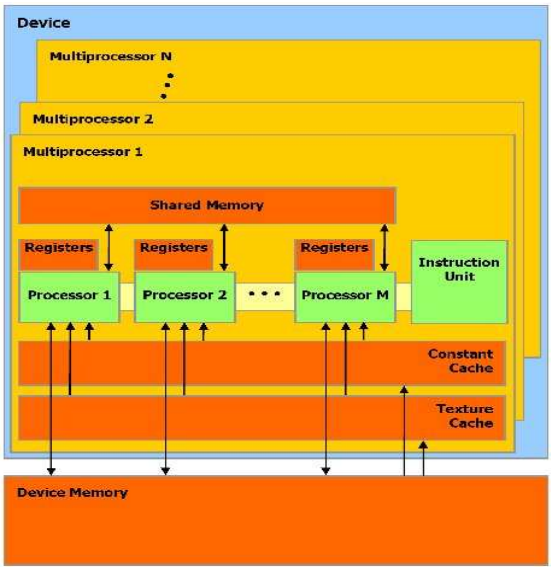

Figura 3: Modelo da hierarquia de memória CUDA.

Os registradores armazenam as variáveis e a memória compartilhada armazena os dados compartilhados entre *threads* de uma mesmo bloco. Portanto, como ambas ficam localizadas dentro dos SM, possuem uma baixa latência. Por outro lado, a memória global do dispositivo é a maior e também é aquela que apresenta a maior latência. Com isto, um bom uso dos registradores e da memória compatilhada é de extrema importância para o desempenho da aplicação. As memórias de constantes e de textura também possuem baixas latências, porém são memórias apenas de leitura. Os registradores são alocados individualmente para cada *thread*, que acessam apenas seus próprios registradores. A memória compartilhada é alocada para um bloco de *threads*, sendo um meio eficiente para o compartilhamento dos dados. Uma estratégia para minimizar a latência da memória do dispositivo é a cópia de porções de dados reusáveis para

memórias de acesso rápido. No entanto, o uso de tais memórias deve ser cuidadoso, pois o mau uso pode prejudicar o desempenho da aplicação.

Neste trabalho executamos os seguintes passos para obter o código CUDA: (1) otimização do código sequencial para CPU, com o objetivo de obter um bom código sequencial que sirva de base para as comparações de *speedup*; (2) uso da ferramenta de *profile* `gprof` (Graham et al. (2004)) para determinar as partes mais custosas do código em termos de tempo de execução; (3) o melhor código paralelo CUDA em termos de desempenho; e (4) uma versão CUDA usando memória compartilhada. Ressaltamos que na otimização do código sequencial temos que decidir quando o código está suficientemente bom ou muito tempo de trabalho será gasto nesta etapa.

4 O ALGORITMO PARALELO

Os passos do algoritmo principal em C+CUDA implementados no *host* são similares aos passos mostrados no algoritmo sequencial (Figura 2). Todas as funções dentro do *loop* no tempo foram executadas na GPU, a ideia principal é evitar cópias de dados entre a CPU e a GPU em 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 as constantes relacionadas com 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 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. Nos experimentos, definimos empiricamente `NUM_THREADS` $= 256$ e `NUM_BLOCKS` $= 30$. A função do método SOR será detalhada na próxima seção considerando implementações usando memórias *shared* e global.

Algoritmo 1: Loop no tempo do algoritmo principal no *host*.

```
...
while( t < t_end){
    set_BondaryCond_GPU <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
    set_InputCondition_GPU <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
    comp_FG_GPU <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
    poisson_system_GPU <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
    // Resolucao do sistema linear
    MetodoSOR_Red_Black_GPU ( ... );
    total_time += time;
    comp_UV_GPU <<< NUM_BLOCKS, NUM_THREADS >>> ( ... );
    t += delt;
}
...
```

4.1 O método Red-Black Successive Over-Relaxation paralelo

Na implementação do método Red-Black-SOR, definido na função MetodoSOR_RedBlack() no Algoritmo 2, o sistema foi construído e armazenado em uma matriz com 5 colunas e $i_{max} \times j_{max}$ linhas. O seguinte ordenamento foi escolhido na implementação do método: a ordem começa no canto inferior esquerdo da malha, no sentido da esquerda para direita, move para a próxima linha acima e continua no sentido da esquerda para a direita, até terminar no canto

superior direito da malha. Os vetores solução (u,v,p) foram armazenados por linha em vetores de tamanho $i_{max} \times j_{max}$, ou seja, nas primeiras posições entram os valores relativos a primeira linha da malha, depois os valores da segunda linha e assim por diante.

No método a malha é colorida como um tabuleiro de xadrez, veja Figura 4, e as iterações divididas pelas células vermelhas e pretas. No algoritmo, primeiro as células em vermelho são atualizadas de it para $it+1$, sendo dependentes somente das células pretas na iteração it. Depois as células pretas são atualizadas de it para $it+1$, sendo dependentes somente das células vermelhas na iteração $it+1$, como mostrado a seguir.

Loop nas células vermelhas:

$$p_P^{it+1} = (1-w)p_P^{it} + \frac{w}{a_P}\left(a_E p_E^{it} + a_W p_W^{it} + a_N p_N^{it} + a_S p_S^{it} - r_P\right) \tag{10}$$

Loop nas células pretas:

$$p_P^{it+1} = (1-w)p_P^{it+1} + \frac{w}{a_P}\left(a_E p_E^{it+1} + a_W p_W^{it+1} + a_N p_N^{it+1} + a_S p_S^{it+1} - r_P\right) \tag{11}$$

A abreviação r_P representa o lado direito na equação da pressão (8) na célula P, N = norte, S = sul, E = leste e W = oeste, conforme Figura 4. 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.

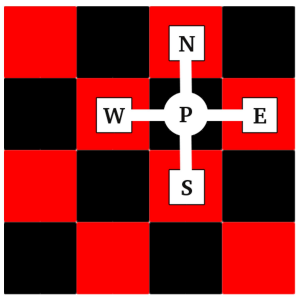

Figura 4: Esquema de coloração Red-Black para as células internas da malha.

O Algoritmo 2 mostra uma visão geral do código do método iterativo Red-Black-SOR implementado no *host*, considerando um número fixo de iterações em cada passo no tempo. O código possui a chamada de dois *kernels*: `red_GPU` e `black_GPU`. A opção de considerar somente um *kernel* com duas chamadas de funções do tipo *device* dentro da própria GPU, não produzirá valores corretos, pois o sincronismo não é garantido para *threads* de blocos diferentes. Então, o algoritmo se resume na execução das funções para as células vermelhas, *kernel* `red_GPU`, referente à Equação (10), e para as células pretas, *kernel* `black_GPU`, referente à Equação (11).

Algoritmo 2: Método SOR Red-Black no host.

```
...
for (iter = 0; iter < IterMax; iter++) {
// executa as posicoes Vermelhas, em seguida as pretas
    red_GPU   <<< NUM_BLOCKS, NUM_THREADS >>> ( imax, jmax, w, ... );
    black_GPU <<< NUM_BLOCKS, NUM_THREADS >>> ( imax, jmax, w, ... );
}
```

O Algoritmo 3 mostra a implementação do *kernel* das células vermelhas (`red_GPU`). O *kernel* relativo as células pretas, `black_GPU`, é similar ao Algoritmo 3, portanto será omitido. A variável `idx` (linha 2) recebe inicialmente o 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 k *threads* devem executar sobre uma malha com $3k$ posições, cada *thread* executará em 3 posições, onde a primeira *thread* computará as posições 0, k e $2k$; a segunda *thread* computará as posições 1, $k+1$ e $2k+1$, e assim sucessivamente. O incremento das posições é calculado com a variável `inc` (linha 3), que recebe o número total de *threads* disparadas pelo *kernel*, e um *loop* (linha 8) garante a execução de todas as posições. Como a malha está separada entre células vermelhas e pretas, cabe a cada *thread* realizar o cálculo da posição na malha que deverá ser computada (linhas 5 e 6). A variável `N` (linha 4) é inicializada com o tamanho da malha. A variável `linha` (linha 5) recebe o valor da linha a qual a célula vermelha está na malha. O cálculo da posição das células vermelhas, `pos` (linha 6), utiliza o seguinte critério: `pos` $= (2 *$ `idx` $+ 1)$ se `linha` $= par$, senão, `pos` $= (2 *$ `idx`$)$. O corpo da iteração do método apresentado é o mesmo para o método SOR sequencial (linhas 10 a 12). A linha 12 calcula o incremento das posições da células e nas linhas 13 e 14 são calculadas as próximas posições das células vermelhas.

Algoritmo 3: Kernel Red na GPU.

```
1  __global__ void red_GPU (int imax, int jmax, ... ) {
2      int idx   = blockDim.x*blockIdx.x + threadIdx.x;
3      int inc   = gridDim.x*blockDim.x;
4      int N     = imax*jmax;
5      int linha = (2*idx) / imax;
6      int pos   = (linha % 2) ? (2*idx + 1) : (2*idx);
7      float aux;
8      while (pos < N) {
9          aux = ( diag_s[pos]*VetorSolucao[pos-imax] +
                   diag_w[pos]*VetorSolucao[pos-1] +
                   diag_e[pos]*VetorSolucao[pos+1] +
                   diag_n[pos]*VetorSolucao[pos+imax] );
10         VetorAnt[pos]     = VetorSolucao[pos];
11         VetorSolucao[pos] = w*(Vetor[pos]-aux)/diag_p[pos] +
                               (1-w)*VetorSolucao[pos];
12         idx  += inc;
13         linha = (2*idx) / imax;
14         pos   = (linha % 2) ? (2*idx + 1) : (2*idx);
15     }
16 }
```

4.2 Utilização da memória compartilhada

Normalmente o uso de uma memória com menor latência é o melhor caminho para obter maior desempenho. O contexto da memória compartilhada em CUDA traz essa ideia, porém é um recurso limitado, visto que seu tamanho é de apenas 16KB por bloco de *threads*. Desta forma é preciso modelar o problema de forma que os dados reutilizáveis sejam armazenados por essa memória. Para a utilização da memória compartilhada neste trabalho a opção foi feita pela utilização de vetores compartilhados de tamanho `NUM_THREADS`. Considerando que o tipo de dado utilizado é `float`, que utiliza 4bytes para ser armazenado na memória, podemos contabilizar o número máximo de vetores utilizáveis na memória compartilhada da seguinte forma:

$$\text{Número de vetores} = \frac{\text{tamanho da memória compartilhada}}{\text{tamanho do vetor float}} = \frac{16\mathbf{KB}}{\texttt{NUM\_THREADS} * 4\mathbf{B}}$$
$$= \frac{16\mathbf{KB}}{256 * 4\mathbf{B}} = \frac{2^{14}\mathbf{B}}{2^{10}\mathbf{B}} = 16 \text{ vetores}$$

4.2.1 *Kernel* das células vermelhas

O Algoritmo 4 apresenta um *kernel* de células vermelhas utilizando memória compartilhada, `red_shared_GPU`. Na versão com memória compartilhada, utilizamos apenas um vetor aproveitando do 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 5. As diferenças entre o Algoritmo 4 e o Algoritmo 3 são: (i) declaração do vetor na memória compartilhada (linha 8); (ii) inicialização do vetor compartilhado (linhas 10 e 11) e (iii) chamada ao vetor compartilhado no lugar de uma chamada a memória global do dispositivo (linha 14). Para a inicialização do vetor compartilhado é necessário o acerto da última posição de cada linha, o que é controlado pelo `if` da linha 11. Assim o vetor, que possui tamanho `NUM_THREADS` $+ 1$, é preenchido em todas as posições com os valores correspondentes às células oeste (W), exceto a última posição da linha, que corresponde a célula leste (E).

Algoritmo 4: Kernel para as células vermelhas na GPU com memória *shared*.

```
1  __global__ void red_shared_GPU (int imax, int jmax, ... ) {
2      int idx   = blockDim.x*blockIdx.x + threadIdx.x;
3      int inc   = gridDim.x*blockDim.x;
4      int N     = imax*jmax;
5      int linha = (2*idx) / imax;
6      int pos   = (linha % 2) ? (2*idx + 1) : (2*idx);
7      float aux;
8      __shared__ float s_sol[threadIdx.x + 1];
9      while (pos < N) {
10         s_sol[threadIdx.x] = VetorSolucao[pos-1];
11         if (threadIdx.x == (imax-1))
12             s_sol[threadIdx.x+1] = vetorSolucao[pos+1];
13         __syncthreads();
14         aux = ( diag_s[pos]*VetorSolucao[pos-imax] +
                   diag_w[pos]*s_sol[threadIdx.x] +
                   diag_e[pos]*s_sol[threadIdx.x+1] +
                   diag_n[pos]*VetorSolucao[pos+imax] );
15         VetorAnt[pos]     = VetorSolucao[pos];
16         VetorSolucao[pos] = w*(Vetor[pos]-aux)/diag_p[pos] +
                               (1-w)*VetorSolucao[pos];
17         idx  += inc;
18         linha = (2*idx) / imax;
19         pos   = (linha % 2) ? (2*idx + 1) : (2*idx);
20     }
21 }
```

4.2.2 Cálculo do critério de parada da norma do máximo

Uma forma usual para verificar a convergência é calcular a norma do máximo para a diferença entre duas iterações consecutivas. Uma maneira eficiente e facilmente paralelizável em

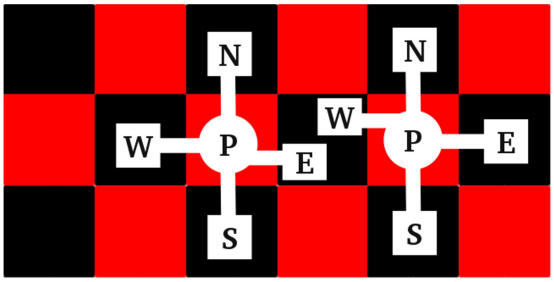

Figura 5: Exemplo de reuso para a memória compartilhada.

C+CUDA para encontrar o maior valor em um vetor é fazendo uma redução em árvore binária. Para melhor entendimento da redução em árvore binária, suponha que queremos encontrar o maior valor de um vetor com 8 posições. Primeiramente dividimos os elementos em 2 grupos, conforme visto na Figura 6. Os elementos 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 as posições 1 e 3 é o maior 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.

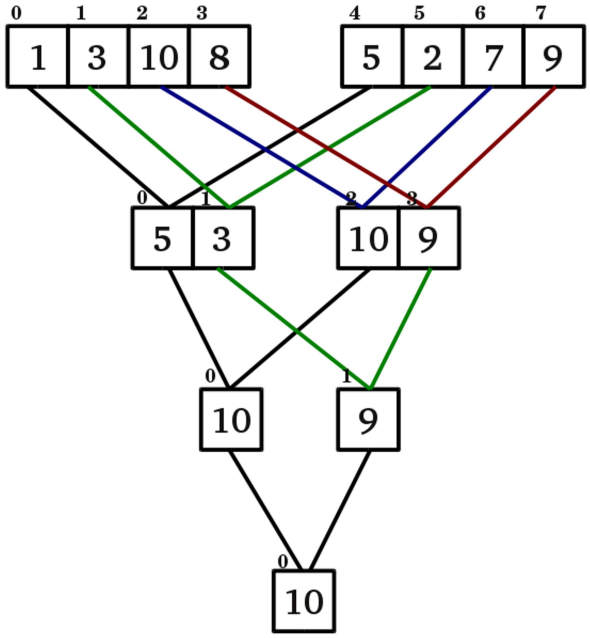

Figura 6: Exemplo de redução em árvore binária.

O Algoritmo 5, *kernel* `reductionMax`, apresenta uma redução binária com uso de memória compartilhada. Os parâmetros deste *kernel* são: (i) o tamanho do vetor (`N`); (ii) o vetor com os valores a serem avaliados (`vetor`) e (iii) um vetor auxiliar (`parcial`), do tamanho do número de blocos, que receberá resultados da redução de cada bloco. Caso o número de *threads* seja

maior que o tamanho do vetor, cada *thread* avaliará mais de uma posição de forma similar ao explicado para o Algoritmo 3. Cada *thread* armazenará o maior valor encontrado por ela em um vetor `cache` compartilhado (linha 11). Em seguida, é realizado uma redução em árvore binária no vetor `cache` (linhas 15 a 22); observe que antes de cada iteração do *loop* de redução é necessário a sincronização das *threads* (linhas 13 e 21). Em seguida, uma única *thread* de cada bloco (linha 23) escreve o resultado de sua redução em um vetor denominado `parcial` na memória global do dispositivo (linha 24). Para finalizar a redução, o vetor `parcial` deve ser analisado pelo *host*, ou seja, copiado para o *host* e finalizar sua execução pela CPU. Como o número de blocos é pequeno, a finalização da redução na CPU não se torna um ponto crítico para o desempenho.

Algoritmo 5: Algoritmo de redução do máximo.

```
1  __global__ void reductionMax (int N, float *vetor, float *parcial) {
2      __shared__ float cache[NUM_THREADS];
3      int idx        = blockIdx.x*blockDim.x + threadIdx.x;
4      int inc        = blockDim.x*gridDim.x;
5      int i, k;
6      float temp = 0;
7      for (i = idx; i < N; i += inc) {
8          if (temp < vetor[i]) temp = vetor[i];
9      }
10     // setar valores do cache
11     cache[threadIdx.x] = temp;
12     // sinconizar as threads no bloco
13     __syncthreads();
14     // para a reducao, threads por bloco deve ser potencia de 2
15     for ( k = (blockDim.x >> 1); k > 0; k >>= 1) {
16         if (cacheIndex < k) {
17             if (cache[thredIdx.x] < cache[threadIdx.x+k]) {
18                 cache[threadIdx.x] = cache[threadIdx.x+k];
19             }
20         }
21         __syncthreads();
22     }
23     if (threadIdx.x == 0) {
24         parcial[blockIdx.x] = cache[0];
25     }
26 }
```

5 RESULTADOS COMPUTACIONAIS

Os experimentos foram executados em uma CPU AMD Phenon(tm) 9950 (Quad-Core) de 2600MHz, com 512KB de cache L2 por core e 8GB de DRAM DDR2 de 800 MHz. O sistema operacional empregado foi o Linux Ubuntu 9.10 (karmic) 64 bits e o compilador de C o gcc 4.3.0. A placa de vìdeo utilizada foi uma NVIDIA Tesla C1060 com 4GB de DRAM GDDR3. A versão do compilador CUDA foi o nvcc 2.1. A placa de video Tesla C1060 é equipada com uma GPU GT200 que possui 10 *Thread Processing Cluster* (TCPs) de 3 SMs cada, e cada SM comtêm 8SPs, totalizando 240 SPs (CUDA *Cores*). Os SPs operam com um clock máximo de 1,3 GHz. A GPU GT200 permite criar até 512 threads por bloco de *threads* e é capaz de manter o estado de 32K threads simultaneamente (uma grid de 32K threads). Duas aplicações foram implementadas neste trabalho, o problema da cavidade com cobertura deslizante e o do escoamento sobre um degrau.

5.1 O problema da cavidade com cobertura deslizante

Para verificar a eficiência do algoritmo paralelo CUDA implementado neste trabalho, foi simulado o problema da cavidade com cobertura deslizante, Figura 7, que consiste de um recipiente quadrado com lados de tamanhos iguais a $1\ metro$, e preenchido com um fluido. A cobertura do recipiente se move com velocidade constante, causando movimentação do fluido. Condições de contorno de não deslizamento são impostas em todos os quatro segmentos das fronteiras, isto é, velocidade na fronteira superior igual à velocidade da cobertura do compartimento ($u = 1$ e $v = 0$) e velocidade nula nas outras três fronteiras ($u = 0$ e $v = 0$). Para o exemplo simulado o número de Reynolds é igual a 1000. O problema da cavidade representa um escoamento que parte de uma condição inicial, sendo gradualmente acelerado até uma condição de regime permanente, isto é, a solução torna-se estacionária no tempo a partir de um determinado número de iterações. Nesse exemplo, o tempo final de computação é $26\ seg$ para todos os experimentos sendo utilizado um tamanho de passo fixo que obedece a condição de estabilidade (9).

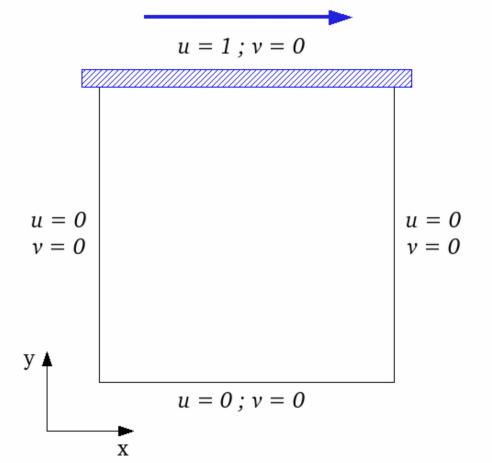

Figura 7: Descrição do domínio - Escoamento em uma cavidade para $Re = 1000$.

A Figura 8 apresenta as linhas de correntes para uma malha 512×512 usando o algoritmo sequencial, C+CUDA e C+CUDA com memória compartilhada. É possível observar que os resultados são virtualmente iguais.

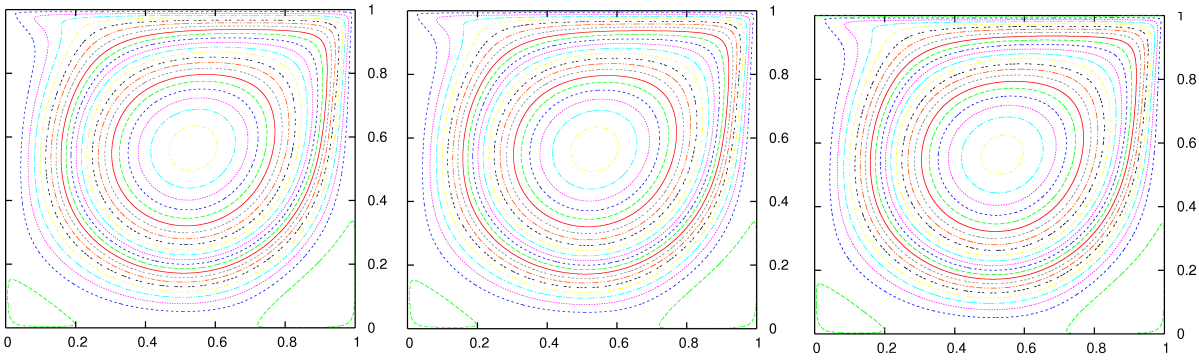

Figura 8: Linhas de corrente para o problema da cavidade usando o algoritmo sequencial (esquerda), C+CUDA (meio) e C+CUDA com memória compartilhada (direita).

A Figura 9 mostra os perfis das velocidade para uma malha 512×512 considerando uma linha horizontal, para a direção x, e uma linha vertical, para a direção y, que passa pelo centro geométrico da cavidade para: (i) o algoritmo C+CUDA; (ii) sequencial usando tipo *float*; (iii) sequencial usando tipo *double* e (iv) solução semi-analítica apresentada em Erturk et al. (2005). Observe que todos os algoritmos desenvolvidos nesse trabalho (itens (i), (ii) e (iii)) produzem resultados equivalentes. A solução semi-analítica em Erturk et al. (2005) utiliza uma malha mais refinada, 640×640, que a definida nos experimentos da Figura 9, por isso a diferença observada. Observe também que os resultados sequenciais utilizando tipos *float* e *double* não diferem significamente, portanto a acuidade da aproximação tanto para o algoritmo sequencial quanto para o algoritmo C+CUDA está garantida.

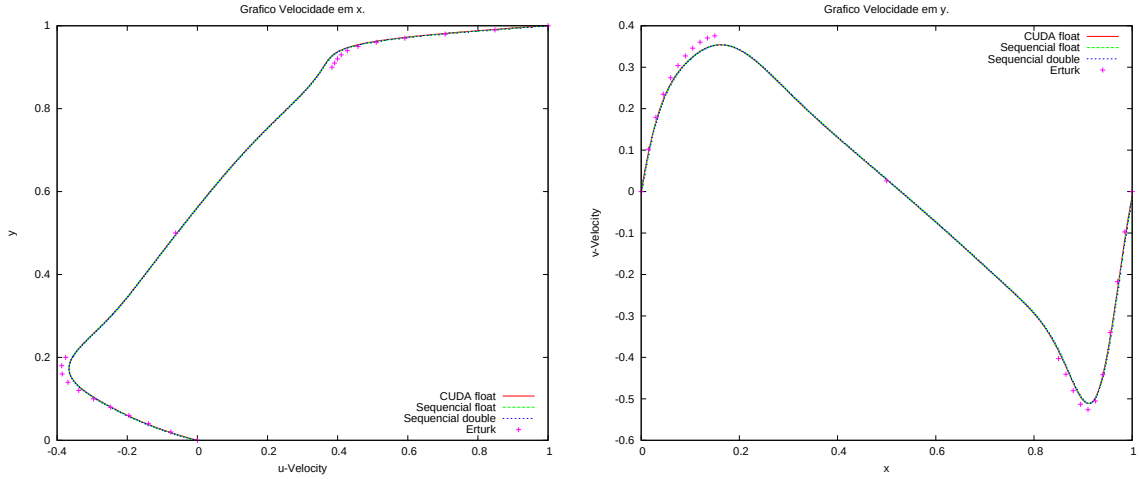

Figura 9: Perfis das velocidades para o problema da cavidade na direção x (esquerda) e direção y (direita).

A Tabela 1 apresenta os tempos de execução para as três versões do algoritmo e diferentes tamanhos de malhas. No problema da cavidade foi considerado um número fixo de iterações em cada passo no tempo, 100 iterações para os primeiros 100 passos no tempo e 5 iterações nos demais passos. Na tabela é apresentado o número total de iterações ($Iter_{total}$) e os tempos em segundos obtido no algoritmo sequencial (T_{Seq}), C+CUDA (T_{CUDA}) e C+CUDA com memória compartilhada ($T_{CUDA_{sh}}$). As duas últimas colunas mostram, respectivamente, o *speedup* para o algoritmo C+CUDA (Sp) e o *speedup* para os algoritmo C+CUDA com memória compartilhada (Sp_{Sh}).

$Malha$	$Iter_{total}$	T_{Seq} (seg)	T_{CUDA} (seg)	$T_{CUDA_{sh}}$ (seg)	Sp	Sp_{Sh}
32×32	25105	2.67	1.60	1.61	1.67	1.66
64×64	41745	20.50	2.96	2.99	6.92	6.85
128×128	75025	189.34	9.18	9.16	20.64	20.67
256×256	141590	1374.07	39.82	38.37	34.51	35.81
512×512	279970	11124.19	281.22	259.01	39.56	42.95
1024×1024	1095355	182426.98	3938.06	3609.52	46.32	50.54

Tabela 1: Tempos de execução (T) e *SpeedUp* (Sp) – Problema da cavidade.

Observe na Tabela 1 que, a medida que aumentamos o número de células, o *speedup* aumenta. O uso da memória compartilhada também proporciona um aumento do *speedup*, porém

este ganho é menor para malhas menores pois o algoritmo não apresenta muitos pontos de reuso dos dados utilizáveis pela arquitetura CUDA. A Figura 10 apresenta os *speedups* Sp e Sp_{Sh} para as malhas 512×512 e 1024×1024, onde percebemos a melhora do desempenho que o uso da memória compartilhada proporciona.

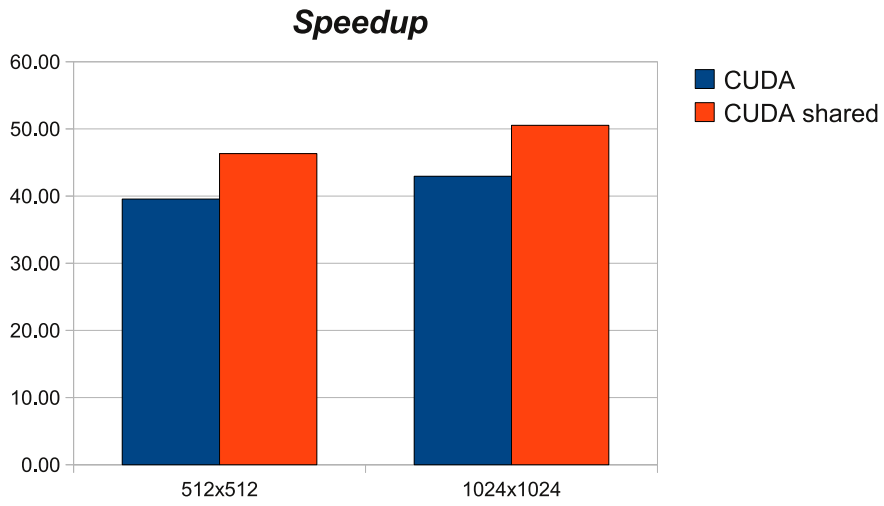

Figura 10: Comparação do *Speedup* sem e com utilização da memória *shared* – Problema da cavidade.

5.2 Escoamento sobre um Degrau

Outro problema bastante popular no processo de validação de códigos de simulação é a análise do comportamento do fluido sobre um degrau, como mostra a Fig. 11. O fluido entra no domínio com velocidade horizontal u_0 e seu comportamento é alterado devido ao alargamento do canal. O fluido entra na fronteira da direita com condição de contorno tipo *inflow* e sai na fronteira da esquerda com condição de controno tipo *outflow*. Nas fronteiras superior e inferior é considerado a condição de contorno tipo *no-slip*. O domínio, com dimensões de $(0; 29) \times (0; 1,5)$ e o degrau está localizado em $(7,5; 0,75)$. Os valores iniciais de pressão e velocidades são $p = 0$, $u = 0$ e $v = 0$, entretanto é considerado $u = 1$ na metade superior do domínio para que, no instante inicial, o fluido dentro do domínio satisfaça a equação da continuidade (Griebel, 1998). O número de Reynolds utilizado é igual a 100.

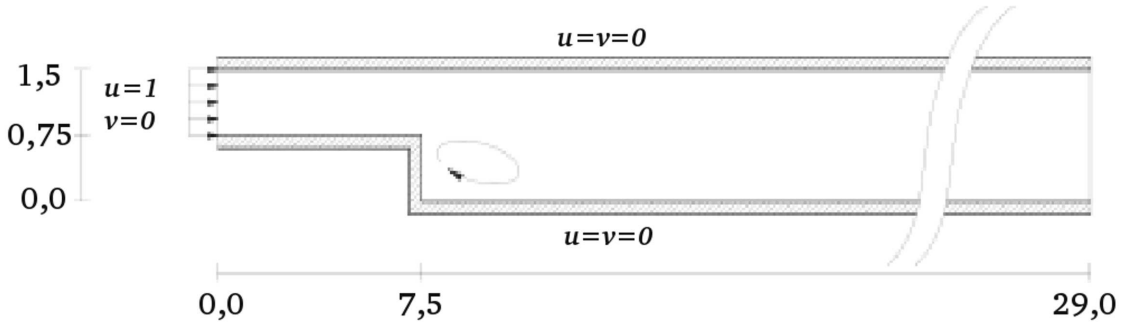

Figura 11: Descrição do domínio - Escoamento sobre um degrau.

No problema do degrau foi considerado como critério de parada a norma do máximo para a diferença entre duas iterações consecutivas do método Iterativo Red-Black-SOR. Como o cálculo da norma necessita de varrer, pelo menos uma vez, um vetor na memória do dispositivo, optou-se por calcular a norma a cada cinco iterações do método Red-Black-SOR a fim de minimizar o acesso à memória. Para efeito de comparação, o algoritmo sequencial também calcula a norma a cada cinco iterações. A Figura 12 apresenta as linhas de corrente usando o algoritmo sequencial, C+CUDA e C+CUDA com memória compartilhada para o domínio discretizado em uma malha com 512×128 células, sendo os resultados virtualmente iguais.

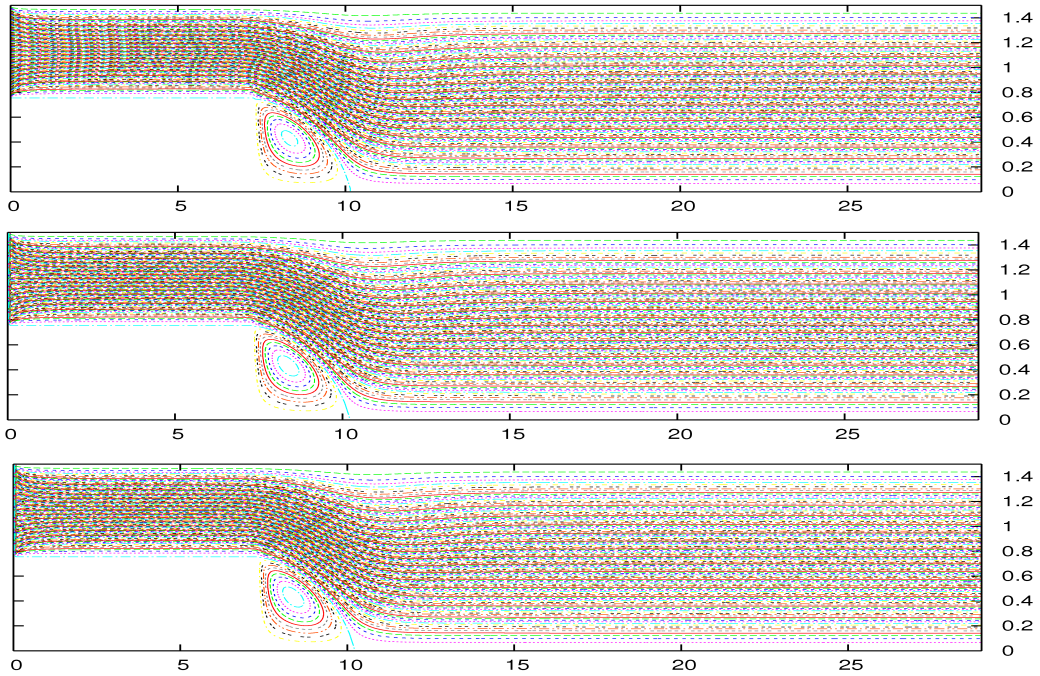

Figura 12: Linhas de corrente do escoamento sobre um degrau usando o algoritmo sequencial (topo), C+CUDA (meio) e C+CUDA com memória compartilhada (base).

A Tabela 2 mostra os tempos de execução dos três algoritmos para a malha 512×128. Na tabela é apresentado o número total de iterações ($Iter_{total}$), os tempos em segundos (T) e o *SpeedUp*. Observe que a utilização de C+CUDA diminuiu consideravelmente o tempo de execução, mesmo realizando um número maior de iterações. A diferença entre o número de iterações do algoritmo sequencial e o CUDA ocorre devido a diferença do hardware CUDA em relação ao *hardware* da CPU para a realização dos cálculos. Em alguns passos no tempo o algoritmo CUDA necessita de mais iterações para convergir, no entanto essa diferença não é muito significativa considerando o número de iterações por passo no tempo e o fato que o teste de convergência é realizado a cada 5 iterações.

$Algoritmo$	$Iter_{total}$	T (seg)	$SpeedUp$
$Sequencial$	147280	$1352.50s$	$-$
$CUDA$	148515	$26.76s$	50.54
$CUDA_{shared}$	148515	$24.43s$	55.36

Tabela 2: Total de Iterações ($Iter_{total}$), tempos de execução (T) e *SpeedUp* - Problema do degrau.

6 CONCLUSÃO

Neste trabalho foi examinado os benefícios do uso de GPUs para aplicações em mecânica dos fluidos computacional discretizadas pelo método das diferenças finitas, utilizando uma formulação implícita-explícita das equações de Navier-Stokes. A resolução dos sistemas lineares resultantes foi feita usando um esquema de coloração Red-Black para as células internas da malha e o método iterativo Red-Black-SOR. Foram implementadas duas versões paralelas em C+CUDA para as Equações de Navier-Stokes, uma com memória global e outra com memória compartilhada. As duas versões foram executadas em uma placa de vídeo NVIDIA Tesla C1060. Os resultados experimentais mostram que *SpeedUps* da ordem de 55 vezes podem ser obtidos, constatando que o uso de GPUs é uma boa alternativa para a paralelizaçao de códigos de problemas em mecânica dos fluidos computacional discretizados pelo método das diferenças finitas. Além disso, foi discutido uma implementação que utiliza memórias com menor latência para os algoritmos implementados assim como suas limitações e vantagens. Nos experimentos a utilização da memória compartilhada reduziu o tempo de computação e produziu um aumento do *speedup*.

AGRADECIMENTOS

Os autores Wesley Menenguci e Lucas Veronese agradecem, respectivamente, ao Fundo de Apoio à Ciência e Tecnologia do Município de Vitória (FACITEC) e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelas bolsas de mestrado concedidas. Os demais autores agradecem ao CNPq pelo apoio recebido dentro do escopo dos Projetos CNPq 620185/2008-2 e 309172/2009-8.

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., e Yelick K.A. The landscape of parallel computing research: A view from berkeley. Relatório Técnico UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

Cohen J.M. e Molemaker M.J. A fast double precision CFD code using CUDA. In *Proceedings of the 21st Parallel Computational Fluid Dynamics*. Monffett Fiel, California, 2010.

De Souza A.F. Computing unified device architecture (cuda) a mass-produced high performance parallel computing platform. Relatório Técnico, SBAC-PAD 2008, 2008.

Erturk E., Erturk E., Gokcol C., Corke T.C., e Gökçöl C. Numerical solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *International Journal for Numerical Methods in Engineering*, 48(7):747–774, 2005.

Graham S.L., Kessler P.B., e McKusick M.K. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39(4):49–57, 2004. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/989393.989401.

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

Hennessy J.L. e Patterson D.A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., fourth edição, 2006.

Irwin M.J. e Shena J.P. Revitalizing computer architecture research. In *Third in a Series of CRA Conferences on Grand Research Challenges in Computer Science and Engineeringg*. 2005.

Jacobsen D.A., Thibault J.C., e 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, 2010.

Klockner A., Warburton T., Bridge J., e Hesthaven J. Nodal discontinuous Galerkin methods

on graphics processors. *J. Comput. Phys.*, 228:7863–7882, 2009.

Moore G.E. Cramming more components onto integrated circuits. *Electronics*, 38(8):1–4, 1965.

Nickolls J., Buck I., Garland M., e Skadron K. Scalable parallel programming with cuda. *ACM Queue*, 6(2):14–53, 2008.

NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. NVIDIA, 2008.

Senocak I., Thibault J., e 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, 2009.

Shinn A.F. e Vanka S.P. Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit. *ASME Conference Proceedings*, 2009(43864):125–133, 2009. doi:10.1115/IMECE2009-11587.

Thibault1 J.C. e 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, 2009.

Veronese L., Lima L.M., Souza A.F.D., e 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, 2010.

Wulf W.A. e McKee S.A. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.