Extensão em CUDA para o framework waLBerla

José Aurimar Sepka Junior[1], **Daniel Weingaertner**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.031 – 81531-900 – Curitiba – PR – Brazil

{jasjunior,danielw}@inf.ufpr.br

Abstract. *This paper presents a CUDA extension to the waLBerla framework. waLBerla is a massively parallel framework for stencil based algorithms operating on block-structured grids, with the main application field being fluid simulations in complex geometries using the LBM. To increase the performance and allow the heterogeneous compute, a new CUDA module was created. Additionally, we compared the simulations' performance using linear memory and pitched memory. Moreover, different sizes of domain were analysed. The achieved results in double precision are according related works. The CUDA module reaches up to 612 MLUPS with ECC disabled and 489 MLUPS using ECC on GPU Tesla K40m.*

Resumo. *Este trabalho apresenta uma extensão em CUDA para o framework waLBerla. waLBerla é um framework massivamente paralelo que utiliza algoritmos baseados em stencil operando sobre uma grid estruturada de blocos com principal aplicação em simulações de fluido com geometria complexa usando o LBM. Para aumentar a performance e permitir o uso de computação heterogênea um novo módulo em CUDA foi criado. Além disso, esse trabalho também levou em conta o desempenho das simulações utilizando memória alocada de maneira linear e alinhada e também analisou diferentes tamanhos de domínio com a finalidade de definir um critério para alocação eficiente de grids de blocos de threads para a GPU. Os resultados obtidos, usando operações de ponto flutuante de dupla precisão, estão de acordo com a literatura. O módulo CUDA alcançou 612 MLUPS com ECC desabilitado e 489 MLUPS usando ECC na GPU Tesla K40m.*

1. Introdução

As Unidades de Processamento Gráfico (GPU) evoluíram consideravelmente nos últimos anos, entre as principais melhorias podemos citar: aumento do poder de processamento, aumento da largura de banda de memória e maior quantidade de memória global. Dessa maneira, deixaram de ser apenas processadores gráficos e passaram a ser usadas para resolver os mais variados problemas computacionais, tornando-se assim unidades de processamento gráfico de propósito geral (GPGPU).

Na Dinâmica de Fluidos Computacionais (DFC) os problemas abordados requerem um grande poder computacional, devido a grande quantidade de operações realizadas e o grande número de iterações, sendo assim, as atuais GPUs são adequadas para as simulações da DFC, pois possuem uma grande quantidade de processadores paralelos e sua capacidade em realizar operações de ponto flutuante, de precisão simples e dupla, é extremamente alta se comparado com uma CPU.

O uso de GPUs em simulações da DFC cresceu ao longo dos últimos anos, como visto nos trabalhos relacionados. Isso porque associado ao grande poder computacional das GPUs e métodos altamente paralelizáveis, como o Método de Lattice Boltzmann (LBM) [Mohamad 2011], produzem excelentes resultados.

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

Tendo em vista o poder computacional das GPUs, métodos altamente paralelizáveis e a falta de suporte à GPU do *framework* waLBerla, optou-se por desenvolver um novo módulo para o *framework* que utilize as propriedades das GPUs em simulações de fluidos que utilizam o LBM.

O restante do trabalho está organizado da seguinte maneira: Na seção 2 o método de Lattice Boltzmann e o modelo de *lattice* utilizado são apresentados. A tecnologia CUDA e o *framework* waLBerla são introduzidos nas seções 3 e 4, respectivamente. Os trabalhos relacionados estão na seção 5. Todo o desenvolvimento do módulo CUDA é descrito na seção 6. Os resultados obtidos são apresentados na seção 7. E, finalmente, a conclusão é apresentada na seção 8.

2. Método de Lattice Boltzmann

O Método de Lattice Boltzmann (LBM) surgiu como uma eficiente alternativa às equações de Navier-Stokes (NS) para soluções numérias na DFC. O LBM simula o comportamento de um fluido através da representação de suas partículas sobre uma malha (*lattice*). Cada ponto da malha representa uma célula contendo um número fixo de Funções de Distribuição de Partículas (PDF). Essas funções representam uma porção do fluido que se propaga em direções específicas, em relação as células vizinhas, e com velocidade definida. Durante a propagação o sentido das partículas pode ser alterado quando ocorrerem colisões entre as partículas do fluido ou colisões com as bordas do recipiente no qual o fluido está mantido [Succi 2001].

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

No LBM o domínio da solução é dividido em *lattices* de uma, duas ou três dimensões, sendo que cada ponto está associado a um conjunto de valores que representam as PDFs. Cada *lattice* possui uma dimensão e o número de velocidades microscópicas (e_i), e é representado por DdQb, onde d é a dimensão do problema e b é a quantidade de velocidades. Nesse trabalho o modelo de *lattice* utilizado foi o D3Q19, como mostra a Figura 1.

3. CUDA

Em novembro de 2006, a empresa NVIDIA apresentou a tecnologia *Compute Unified Device Architecture* (CUDA) com o objetivo de atender a demanda por processamento de

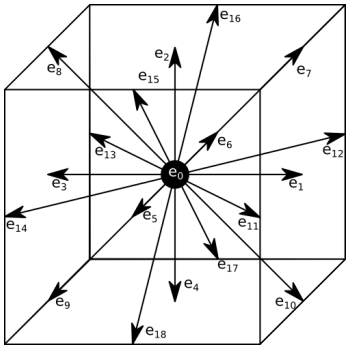

Figura 1. Lattice D3Q19

propósito geral em placas gráficas. CUDA é uma plataforma de computação paralela de propósito geral e um modelo de programação que aproveita a arquitetura das GPUs para resolver problemas computacionais de maneira altamente paralela [NVIDIA 2015a].

O modelo de arquitetura paralela utilizado pelas GPUs é conhecido como *Single-Instruction, Multiple-Thread* (SIMT), ou seja, uma única instrução é executada paralelamente por diversas *threads* independentes [NVIDIA 2015b]. Esse modelo é semelhante ao modelo *Single-Instruction, Multiple-Data* (SIMD), entretanto as instruções SIMT são específicas para execução e comportamento de uma única *thread* [NVIDIA 2015a]. O modelo de programação CUDA assume um sistema composto por *host* (CPU), *device* (GPU) e um conjunto de extensões das linguagens de programação C e C++. CUDA também introduz o conceito de *kernel*, definido como uma função executada N vezes em paralelo por um conjunto de N *threads*, ou seja, todas as *threads* executam o mesmo código simultaneamente.

CUDA possui uma hierarquia de grupos de *threads* na qual um conjunto de *threads* é organizado em forma de blocos. *Threads* do mesmo bloco podem cooperar através do compartilhamento de informação, por meio da memória compartilhada, e sincronizar sua execução pelo acesso coordenado à memória através de *__syncthreads()*. Essa sincronização atua como uma barreira na qual as *threads* do mesmo bloco devem aguardar até que todas as *threads* tenham terminado sua execução. Cada *thread* executa uma instância do *kernel* e possui um identificador único [NVIDIA 2015a].

4. waLBerla

waLBerla (*widely applicable Lattice Boltzmann solver from Erlangen*) é um *framework* massivamente paralelo desenvolvido pelo Laboratório de Simulação de Sistemas (LSS) do departamento de informática da universidade de Erlangen-Nürnberg na Alemanha para simulações de fluidos que utilizam o LBM [Götz et al. 2007]. A atual versão do *framework* é capaz de simular vários fenômenos físicos como: fluxo em um canal aberto, fluxo em um canal aberto com objetos flutuando, fluxo através de meios porosos [Donath et al. 2011] e permite ainda, simular o processo de coagulação em vasos sanguíneos, como apresentado no trabalho de Feichtinger et al. (2011).

Além desses aspectos, outras características relevantes do *framework* são sua escalabilidade e alto desempenho, como visto no trabalho de Godenschwager et al. (2013), alcançando a marca de 1,93 trilhões de células de fluidos atualizadas por segundo

usando 1,8 milhões de *threads* em supercomputadores como o JUQUEEN e o SuperMUC para uma simulação do fluxo sanguíneo.

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

5. Trabalhos Relacionados

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

Tölke e Krafczyk [Tölke and Krafczyk 2008] publicaram um artigo apresentando uma nova abordagem para o LBM, primeiro, utilizaram o tipo de *lattice* D3Q13, e segundo, utilizaram um *lattice* onde cada célula é representada por um dodecaedro, ou seja, cada face do dodecaedro possui uma PDF. Além disso, também utilizaram memória compartilhada para armazenar os valores das PDFs.

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

Visto que a memória global das placas gráficas apresenta alta latência e o LBM possui uma grande quantidade de dados que precisam ser trocados entre a memória global e a GPU, o acesso padrão à esses dados em memória pode contribuir para um baixo desempenho da simulação. Uma forma de resolver esse problema é a utilização de memória compartilhada, entretanto no artigo de Obrecht et al. (2011), os autores apresentam uma implementação em CUDA do LBM em três dimensões (D3Q19) usando apenas memória global. Nesse artigo os autores mostram uma forma eficiente de acesso aos dados em memória e, além disso, há uma seção que trata dos princípios de otimização para algoritmos em CUDA.

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

de colisão, com isso a colisão é calculada com base nos valores dos PDFs das células vizinhas e armazenado na célula atual, semelhante ao trabalho de Rinaldi et al. (2012).

Segundo os autores essa abordagem apresentou um desempenho melhor pois a leitura dos dados desalinhados tem impacto menor no desempenho do algoritmo com relação a escrita desalinhada. Além dessa estratégia, os autores também utilizaram otimizações aritméticas, redução de variáveis temporárias e uma técnica de otimização de *index*, que reduziu a quantidade de registradores, porém está última otimização não fica claro como foi realizada.

6. Modulo CUDA

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

A principal classe do módulo CUDA é a classe *GPUField*. Essa classe representa a estrutura de dados que será processada pela GPU. Possui informações como tamanho do domínio nas dimensões x, y e z e o número de *ghost layers*, além de suportar os *layouts Structure-of-Array* (SoA) e *Array-of-Structure* (AoS). Além da classe *GPUField*, foram desenvolvidas as classes *GPUFieldLinearMem* e *GPUFieldMemPitch* para alocação de memória linear e memória alinhada, respectivamente.

Para copiar os dados entre a CPU e a GPU foram criadas as funções de cópia *fieldCpyLinearMem()* e *fieldCpyMemPitch()* com base nas funções já definidas pela API CUDA. Para percorrer os dados em memória foram implementadas duas classes, a primeira, definida como *FieldIndexingLinearMem*, percorre os dados alocados através da classe *GPUFieldLinearMem* e a segunda classe foi definida como *FieldIndexingMemPitch* para percorrer os dados alocados pela classe *GPUFieldMemPitch*. Essas classes possuem métodos similares a iteradores, que definem qual o espaço do domínio será percorrido durante a simulação. Os métodos dessas classes são os mesmos, a única diferença é que os métodos da classe *GPUFieldMemPitch* devem considerar o alinhamento de memória.

Para acessar os dados na memória da GPU foram implementadas as classes *FieldAccessorLinearMem* e *FieldAccessorMemPitch* que possuem métodos para manipulação dos dados. Para definir novos valores para as PDFs temos o método *get()*, que retorna o endereço de memória de cada PDF. Visto que o LBM depende dos valores de seus vizinhos, foi implementado o método *getNeighbor()* para auxiliar a atualização de cada PDF de acordo com a posição de cada célula do *lattice*. E para definir a quantidade de blocos e *threads* em cada *kernel* foi implementado o método *set()*.

Para que fosse possível integrar o novo módulo CUDA ao *framework* waLBerla foi necessário criar a classe *Kernel*. A classe *Kernel* foi implementada usando o conceito

Adapter de padrões de projeto [Gamma et al. 1994]. Essa classe é responsável por configurar o *kernel* da GPU, ou seja, definir a quantidade de blocos em cada *grid* e a quantidade de *threads* em cada bloco. Além disso, é responsável também por receber os parâmetros que serão enviados ao *kernel* e lançar o *kernel* para a GPU.

Como o waLBerla é desenvolvido utilizando o padrão C++11 da linguagem de programação C++, a classe *Kernel* se tornou fundamental para o módulo CUDA, pois durante o período de desenvolvimento do módulo o compilador da NVIDIA (nvcc) ainda não suportava nenhuma funcionalidade do C++11, por isso todos os códigos com a extensão ".cu" e todos os arquivos de cabeçalho (*headers*), incluídos nesses arquivos, não podem conter nenhum elemento do C++11. Com isso, a forma tradicional de chamar *kernels* não pode ser utilizada nesse módulo.

Para o módulo CUDA foram implementados três *kernels* na GPU, o primeiro para tratar as condições de contorno estáticas, o segundo para as bordas com velocidade e o terceiro para realizar os passos de colisão e propagação. É importante ressaltar que cada *thread* da GPU é responsável por uma célula do *lattice*, essa abordagem é semelhante ao trabalho de Tölke e Krafczyk [Tölke and Krafczyk 2008] e foi adotada para todos os *kernels* implementados. Outro detalhe importante, com relação ao primeiro *kernel*, é que todas as PDFs de cada célula do *lattice* são atualizadas em uma única iteração, isso só é possível graças a camada *ghost layer*.

Com relação ao modo de propagação das partículas o *kernel* foi implementado utilizando a abordagem feita por Rinaldi et al. (2012) e o operador de colisão utilizado foi o operador *Single Relaxation Time* (SRT). Essa primeira versão do módulo foi desenvolvida para executar as simulações em apenas uma única GPU.

7. Resultados Experimentais

Para validação do módulo CUDA foi utilizado o caso de testes conhecido como *lid-driven cavity* [Ghia et al. 1982]. O objetivo desse teste é comparar o resultado de uma simulação do caso de testes realizada pelo *framework* waLBerla em CPU, pois este já foi devidamente validado, com o resultado do módulo CUDA. Para isso, foram comparados os valores das velocidades (V_x e V_y), obtidos pela GPU, com os resultados da CPU, como pode ser visto nos gráficos da Figura 2. As velocidades obtidas pela simulação na GPU (linha azul) estão de acordo com os valores encontrados pelo *framework*. Esse teste é bastante utilizando para validação do *lid-driven cavity*, como pode ser visto nos trabalhos de Ghia et al. (1982) e Rinaldi et al. (2012). O tamanho de domínio e o valor da velocidade da borda superior, usados nesse teste, foram 128^3 e 0,1, respectivamente. Para esse teste foram executadas 10.000 iterações usando a GPU Tesla C2075 e o tempo total da simulação não foi considerado.

Em nossos testes de desempenho foram usadas três GPUs NVIDIA de arquiteturas diferentes (Tesla C2075, Tesla K40m e GTX 750 Ti). O objetivo desse teste é medir a quantidade de MLUPS (*Million Lattices Cells Updates per Second*), ou seja, a quantidade de células de *lattice* atualizadas por segundo. O tamanho de domínio utilizado foi $N_X \times 256 \times 256$ células em cada direção do *lattice*, variando N_X em 32, 64, 128, 256 e 512, pois a quantidade de células na direção x do domínio define a quantidade de *threads* em cada bloco. Essa abordagem foi adotada para avaliar qual a quantidade de *threads* apresenta o melhor desempenho.

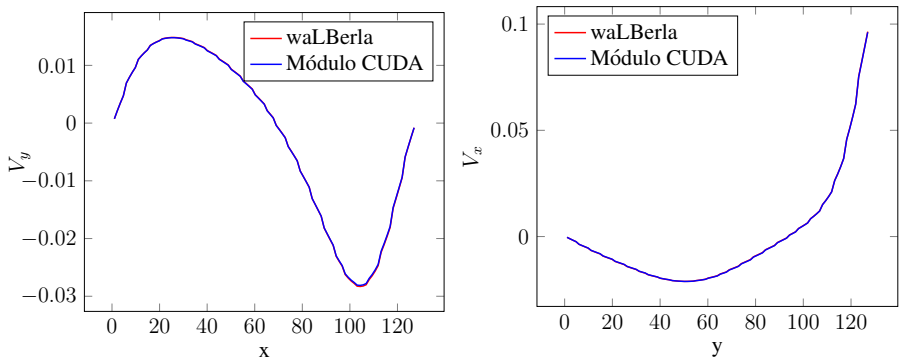

Figura 2. Comparação entre os resultados obtidos pelo *framework* waLBerla e o módulo CUDA. (a) Representa as velocidades na direção y (V_y) sobre a direção x do domínio e (b) representa as velocidades na direção x (V_x) sobre a direção y do domínio.

Além de diferentes tamanhos de domínio também foi levado em consideração o uso de ECC nas GPUs. Esses testes foram feitos com o intuito de poder comparar as GPUs, visto que a GPU GTX 750 Ti não possui suporte à ECC, com isso os testes descritos anteriormente foram feitos com ECC habilitado e desabilitado.

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

O melhor resultado obtido foi de 612 MLUPS utilizando a GPU Tesla K40m para um tamanho de domínio 256^3 com ECC desabilitado e memória linear, como mostra o gráfico da Figura 3. Como é observado no gráfico (Figura 3) houve uma queda de desempenho na simulação usando 512 *threads*, isso porque a taxa de ocupação com 512 *threads* foi inferior a taxa de ocupação com 256 *threads*.

Entretanto, o uso de ECC em simulações de fluidos utilizando o LBM deve ser considerado, pois, como o LBM é uma automato celular, isto é, cada célula do *lattice* interage com as células vizinhas, se um valor incorreto for obtido esse valor será propagado para as demais células do *lattice* invalidando a simulação. Com ECC habilitado o melhor resultado foi de 489 MLUPS com a GPU Tesla K40m com um tamanho de domínio de $512 \times 256 \times 256$ células, como mostra o gráfico da Figura 4.

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

Além dos testes de desempenho, foi realizada uma comparação com o trabalho de Habich et al. (2011), pois a abordagem apresentada pelos autores foi implementada na versão anterior do *framework* waLBerla. O gráfico da Figura 5 mostra os resultados obtidos no trabalho relacionado e os resultados alcançados pelo módulo CUDA. Com o ECC desabilitado o resultado obtido pelos autores foi superior ao nosso trabalho, porém com o uso de ECC o resultado do módulo CUDA foi um tanto superior. Esse resultado

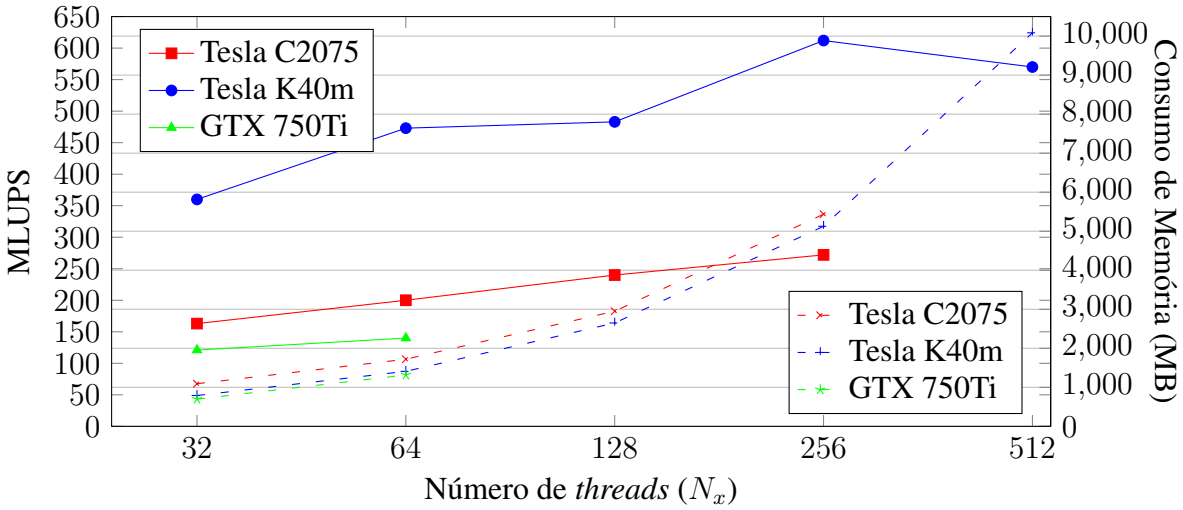

Figura 3. Desempenho das simulações usando memória linear e com ECC desabilitado para um domínio de tamanho N_x×256×256. O eixo secundário y mostra o consumo de memória nos diferentes tamanhos de domínio.

superior, obtido pelos autores, pode ser explicado pelo fato deles terem alcançado uma taxa de ocupação de 50%.

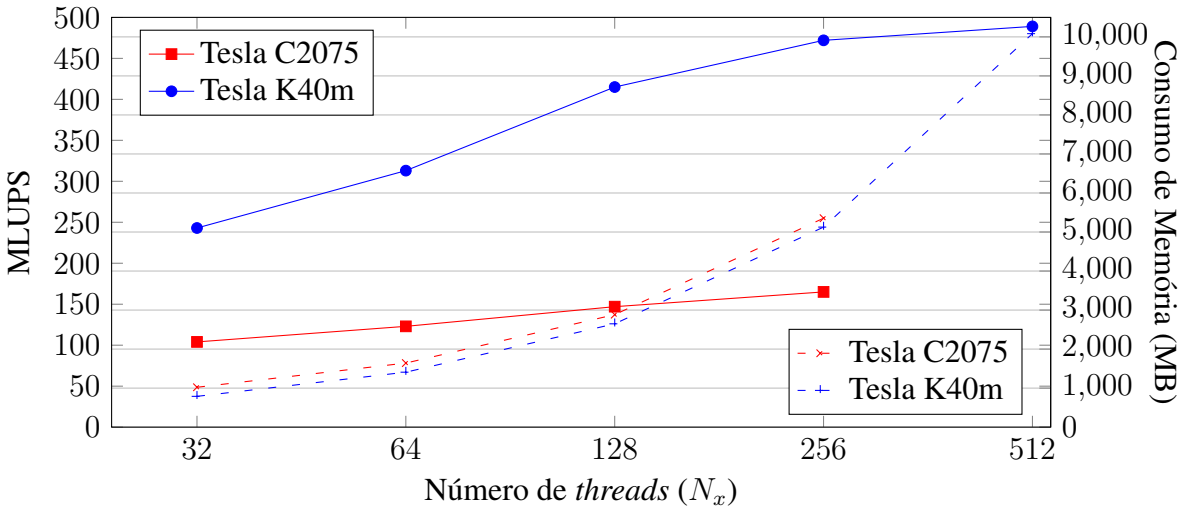

Figura 4. Desempenho das simulações usando memória linear e com ECC habilitado para um domínio de tamanho N_x×256×256. O eixo secundário y mostra o consumo de memória nos diferentes tamanhos de domínio

8. Conclusão

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

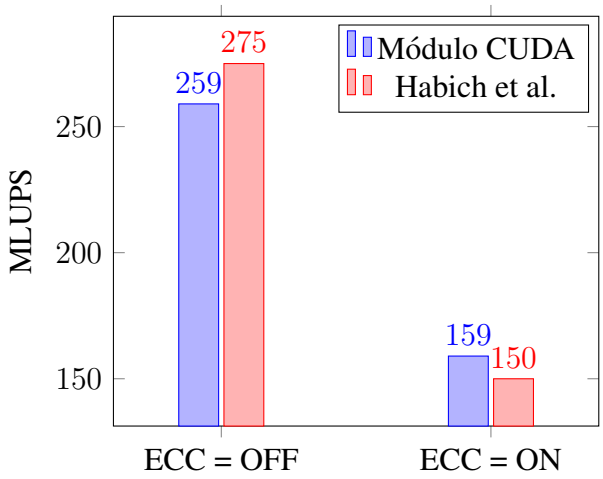

Figura 5. Comparação com o trabalho de [Habich et al. 2011] usando memória linear para um tamanho de domínio de 200×200×200.

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

Na comparação entre memória linear e memória alinhada os resultados obtidos utilizando memória linear foram superiores aos resultados com memória alinhada, ou seja, a maneira como é feito o alinhamento dos dados usando a API *cudaMalloc3D()* do CUDA não trouxe ganho no desempenho das simulações do LBM em três dimensões.

Durante o desenvolvimento desse trabalho foi possível observar que o LBM é um método altamente paralelizável e associado as placas gráficas da NVIDIA produzem excelentes resultados para simulações da DFC. Entretanto, a cópia dos dados entre a CPU e a GPU é um fator que limita o desempenho das simulações. E, apesar do *framework* waLBerla ser uma ferramenta bastante robusta, sua curva de aprendizado é alta se compararmos com o LBM e CUDA.

Referências

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

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

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

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

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

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

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

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

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

NVIDIA (2015b). Whitepaper Fermi. http://www.nvidia.com/content/pdf/fermi_white_ papers/nvidiafermic Acessado em 25/10/2015.

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

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

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

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