Universidade Tecnológica Federal do Paraná - UTFPR
Centro de Pesquisa em Reologia e Fluidos Não Newtonianos - CERNN

Engenharia de Computação

Implementação do LBM em GPU

Aluno:
Waine B. de Oliveira Junior

Relatório
Orientador: Dr. Admilson Teixeira Franco
Coorientador: MSc. Alan Lugarini de Souza

3 de Abril de 2019

Conteúdo

1 Introdução

O LBM é um método para CFD que tem entre suas vantagens a possibilidade de sua paralelização computacional, permitindo que sejam utilizadas arquiteturas de computação paralela para proveito desse fato. Dentre essas arquiteturas estão as GPUs que pelo seu baixo custo, alto processamento e largura de banda da memória, além da expectativa do aumento desses nos próximos anos [1], são uma das escolhas mais promissoras para o LBM.

Para a implementação em GPU foi utilizada a API CUDA. Os casos de validação utilizados foram o escoamento da cavidade com tampa deslizante e entre placas paralelas. Foram aplicadas técnicas de otimização para melhora do desempenho do código e verificado como a variação de parâmetros afeta o número de MLUPS (*Million Lattice Updates per Second*).

Os objetivos do relatório são: implementação do LBM em GPU e a otimização do código; comparação entre os valores obtidos e os analíticos ou obtidos por um método de maior precisão; validação pelo teste de erro de truncamento; análise do desempenho do código e do impacto da variação de certos parâmetros.

2 GPU's

As GPU's (*Graphics processing units*) são circuitos elétricos dedicados ao processamento gráfico em aparelhos eletrônicos. Para sua arquitetura, se aproveitam do fato de que a maior parte das operações feitas são as mesmas em todo domínio e paralelizáveis entre si, sendo consideravelmente mais rápidas que uma CPU para tal processamento.

Com o tempo, placas gráficas começaram a ser utilizadas para outros fins que permitem paralelização, dentre eles a computação científica. GPGPU (*General-purpose computing on graphics processing units*) consiste no uso de uma ou mais GPU's e CPU's para o processamento de um conjunto de dados [2].

Para tal, na maioria das vezes são utilizadas API's (*Application programming interface*) para o uso das funcionalidades da GPU e a conexão dessa com a CPU, sendo a CUDA [3], API para placas de vídeo da Nvidia, uma das mais utilizadas.

2.1 Arquitetura de GPU's da Nvidia

Antes de entender a CUDA e as escolhas quanto à implementação, é necessária uma noção de como é a arquitetura das GPUs da Nvidia. A primeira diferença dessas para as CPUs tradicionais, é seu modelo de execução. Enquanto as CPUs executam uma instrução apenas para um dado (*Single Instruction, Single Data*), as GPUs utilizam um modelo em que uma instrução é executada em um conjunto de dados. No caso da Nvidia, a operação é feita nos dados de várias *threads* (tarefas simultâneas), daí o nome do modelo, *Single Instrucion, Multiple Threads* [4].

Para fazer isso, a GPU precisa reunir várias *threads* que executam a mesma instrução. Esse conjunto de *threads* é chamado de *warp* e atualmente é composto por 32 daquelas. Durante a execução de um *warp* a GPU primeiro garante que os recursos necessários para as *threads* estão disponíveis e então executa a mesma instrução para todas elas ao mesmo tempo.

No caso de divergência entre *threads* em um *warp*, por meio de um `if` por exemplo, a GPU desativa as *threads* que não "entram"na condição e realiza as operações para as outras. Isso é um gargalo no desempenho, já que as tarefas desativadas ficam ociosas nesse meio tempo, desperdiçando recursos da GPU.

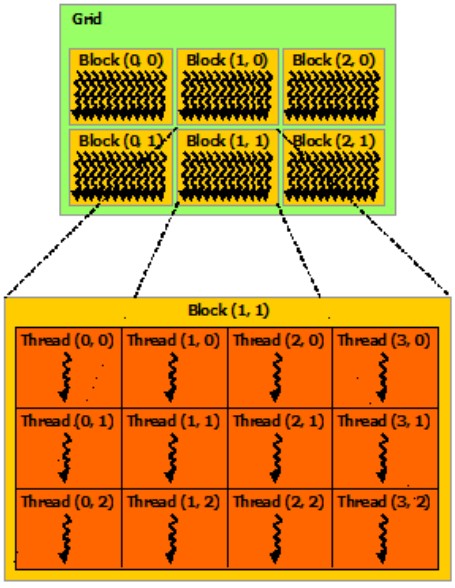

Figura 1: Hierarquia de *threads* [4]

Como mostra a figura 1, as *threads* são divididas em três níveis de hierarquia: *grid*, bloco e a própria *thread*. O *grid* é uma matriz com todos os blocos e esses

são matrizes de *threads*, todos do mesmo tamanho. As dimensões do *grid* e dos blocos são definidas no código e escolhidas de modo a uma *thread* corresponder a um elemento do domínio. A equação 1 é utilizada para definir as dimensões do bloco e do *grid*.

$$DimGrid \cdot DimBloco = DimDominio \tag{1}$$

Há a limitação de no máximo 1024 *threads* por bloco para a Nvidia Tesla K20x. O procedimento para escolher o tamanho do bloco e do *grid* é o seguinte: escolher dimensões para o bloco de modo que seja possível completar o domínio com ele; calcular as dimensões do *grid* em função das dimensões do bloco e do domínio.

Por exemplo, para um domínio de $(256, 128, 256)$, é possível escolher um bloco de $(128, 1, 4)$. Assim a dimensão do *grid* seria de $(256, 128, 256) \div (128, 1, 4) = (2, 128, 64)$. Um exemplo de má escolha para o tamanho do bloco seria de $(96, 4, 2)$ já que assim não seria possível completar o domínio com ele.

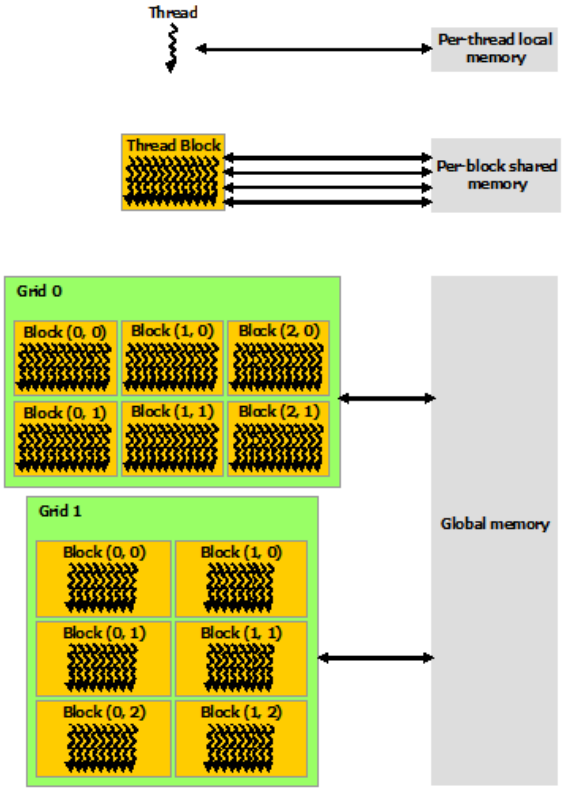

Figura 2: Hierarquia de memória [4]

Threads e blocos tem acesso a certos recursos, sendo os mais relevantes a

memória, como mostra a figura 2, e a capacidade de comunicação. Cada *thread* tem certo número de registradores e memória *on-chip* privado, que outras *threads* não podem acessar. O conjunto de *threads* de cada bloco tem acesso a memória compartilhada do bloco (*shared memory*), além de capacidade de se comunicar com outras por meio de pontos de sincronização (*i.e.* `__syncthreads()`). Já o *grid* têm acesso a memória global da GPU, a que possui maior capacidade, porém também a maior latência de acesso e escrita.

Como todo bloco deve ser executado em apenas um multiprocessador, a execução de dois blocos ao mesmo tempo é limitada pelo número de registradores e a quantidade de memória compartilhada em cada um. Caso os recursos sejam o bastante, vários blocos podem ser executados concomitamente em um mesmo multiprocessador.

2.2 CUDA

A CUDA se apresenta como uma extensão de linguagens de programação como C, C++ e Fortran, e dentre suas funções estão especificar se certa função será utilizada pela GPU ou pela CPU, quais variáveis estarão em uma ou em outra, funções para coletar informações da GPU, dentre várias outras. Porém seu conceito chave é a paralelização do código, que é feita por meio de uma função da GPU chamada pela CPU, denominada *kernel*.

O *kernel* consiste em uma função que retorna `void` e deve possuir o classificador `__global__` em sua declaração. Ele pode ser visto como uma função dentro de um ou vários `for`, em que cada iteração desse `for` é executada paralelamente.

A sintaxe de chamada de um *kernel* é a seguinte: `nome_do_kernel<<<dimGrid, dimBloco, sharedMem>>>(args)`. O `dimGrid` é a variável de tipo `dim3` que fornece o tamanho do *grid*, o `dimBloco` exerce a mesma função, mas com relação ao bloco. Já `sharedMem` é um argumento optativo que é o quanto deverá ser reservado para memória compartilhada por cada bloco, sendo zero quando omitida.

Para a identificação do elemento tratado por cada thread, são utilizadas as variáveis `threadIdx`, `blockIdx`, `blockDim` e `gridDim`, sendo que as com sufixo `Idx` são os identificadores únicos de cada bloco ou *thread*, e aquelas com sufixo `Dim` informam a dimensão do bloco ou do *grid*. Os tipos delas são também `dim3`.

```
i = blockIdx.x*blockDim.x + threadIdx.x          (2)
```

A equação 2 mostra como é feito o cálculo do índice no eixo x e pode ser estendida para os eixos y e z, alterando apenas o sufixo das variáveis.

3 Implemetação do LBM em GPU

Para a implementação do LBM utilizando CUDA é possível reaproveitar grande parte do código sequencial, visto que as operações são as mesmas. Para migração, o que está dentro do `for` se torna um *kernel*. As maiores alterações são com relação a `main`, já que agora é necessário inicializar o dispositivo, alocar e desalocar memória na GPU, etc.

O fluxo do código na `main` é o seguinte: inicialização da GPU; alocação de memória na GPU e na CPU; definição das condições de contorno; inicialização das populações e dos macroscópicos; aplicação da colisão, propagação e das condições de contorno até que convergência ou número máximo de passos seja atingido; salvar as variáveis do último passo; desalocação dos recursos alocados; liberação dos recursos da GPU. Há também saídas de texto com informações sobre a simulação e a GPU durante a execução do programa.

Os passos do LBM (colisão, propagação, aplicação das condições de contorno, macroscópicos) foram unidos em apenas um *kernel*, chamado de `gpu_bc_macr_collision_streaming` que primeiro aplica condições de contorno, então atualiza os macroscópicos e depois faz a colisão e propagação [5]. O algoritmo utilizado para colisão e propagação foi o tradicional *push*, pois apesar do *pull* ser mais rápido [5], não foi observada grande diferença de desempenho entre um e outro.

Visando melhorar a leitura do código, foram definidas funções que apenas executam um *kernel*, assim a `main` não apresenta a sintaxe especial para esse. Também as funções foram divididas em vários arquivos para maior coesão, e aquelas da GPU (com classificadores `__device__` ou `__global__`) foram definidas com o prefixo `gpu` para melhor clareza no código.

3.1 Otimização

Os nortes com relação a otimização do código sequencial seguem valendo para a GPU, por exemplo o acesso contínuo à memória e evitar redundância de contas. Mas surgem questões específicas do GPGPU e da CUDA, como evitar que o código tenha ramificações, minimizar as transfrências de memória entre GPU e CPU, escolher um bom arranjo de *threads* e blocos, dentre outras.

O arranjo de memória foi feito de modo a aproveitar o carregamento de linhas de cache para várias *threads* em um *warp*. Assim a latência de memória para operações no kernel é minimizada [1]. As populações do domínio então são primeiro indexadas pelo seu valor em x, depois pelo seu valor em y, e só então pelo seu

número, fazendo com que sejam contíguas na memória na direção de x. A escolha de x em detrimento de y se deve ao fato do arranjo das *threads* nos blocos, pois essas são contínuas em x, logo devem acessar memória contígua na mesma direção.

A transferência de memória entre GPU e CPU toma um grande tempo, então essa só deve ser feita quando for imprescindível. Também tendo em vista que cada transferência de memória é independente de outra para os macroscópicos, elas podem ser realizadas em paralelo. Mas isso não apresentou melhora considerável no desempenho para os testes realizados.

Um fator que afeta o desempenho é a quantidade de memória utilizada por cada *kernel*, devido a fatores como ocupância (que trocando em miúdos, relaciona o quanto da GPU está sendo utilizado com o quanto poderia ser) e linhas de cache. Então uma estratégia adotada foi trocar tipos como `int` por `short unsigned int` quando possível.

Quanto ao número de *threads* por bloco, é recomendado utilizar um múltiplo de 32 (número de *threads* por *warp*) [4]. Também são usados classificadores que permitem otimizações por parte do compilador, como `__restrict__` e variáveis `const` ou `constexpr`. O cache foi configurado com a opção `cudaFuncCachePreferL1`, em que 16kB do cache são reservados para a *shared memory* e 48kB para o L1 (nível mais rápido de memória), sendo o impacto dessa alteração analisado nos resultados.

Algo que foi observado também é a importância de uma boa implementação das condições de contorno, tanto para malhas grandes quanto para pequenas. Isso ocorre pelos nós de fronteira serem casos especiais, sofrendo ramificação no código e também por não tomarem proveito do arranjo de memória durante a execução. Mesmo que os nós de fronteira sejam uma pequena fração do domínio, para a implementação em GPU podem influenciar consideravelmente o desempenho do programa.

3.2 Implementação das condições de contorno

Devido a relevância das condições de contorno, foi necessário definir um modelo para sua implementação de modo que elas pudessem ser generalizadas e facilmente alteradas entre um e outro caso. Há duas estratégias principais para tal [1]. Uma delas é definir uma lista com nós de fronteira e qual condição de contorno utilizar e então percorrer a lista aplicando-as. A outra é definir um vetor que mapeia e classifica todos os nós no domínio e então percorrê-lo para a aplicação ou não das condições de contorno em cada nó.

Inicialmente foi adotada a primeira estratégia, entretanto ela se apresentou muito ineficiente, possivelmente devido a fatores como necessitar de um *kernel* es-

pecífico para tal, o número de argumentos que necessitava ser passado a cada função, o uso de uma lista para cada condição de contorno, dentre outros.

Então foi utilizada a segunda estratégia, implementando o modelo de mapeamento dos nós semelhante ao utilizado pelo código aberto *Sailfish* [6]. Para a classificação dos nós foi definido o tipo `nodeTypeMap`, que é uma variável de 16 bits composta por "campos de bits", os quais cada um define certa propriedade do nó.

Essas propriedades são utilizadas para a aplicação ou não aplicação de condições de contorno. Dentre elas estão o uso ou não do nó, o esquema de condição de contorno (incluindo o esquema nulo para não aplicação), a direção da normal do nó, e índices para vetores globais que fornecem os valores de velocidade e densidade do nó.

Para as condições de contorno é definida uma função que recebe como parâmetros os dados necessários para sua aplicação. Uma escolha feita foi a passagem das coordenadas do nó para a função, o que tem como vantagem a possibilidade de acesso a todo o domínio pela função, permitindo que valores de densidade ou velocidade sejam extrapolados de nó vizinhos. Porém, devido ao aumento de memória necessária para a função, há uma pequena queda de desempenho.

Apesar do revés, o ganho com relação à gama de condições de contorno que necessitam de extrapolação ou de informação de outros nós é muito grande, então foi escolhido passar as coordenadas para as funções de condição de contorno.

4 Resultados

A validação dos dados foi feita por meio do teste da ordem de erro de truncamento e comparação com a literatura. Já para análise de desempenho foram feitos testes com a variação de alguns parâmetros para quantificar o impacto de cada um. A seguir são apresentadas a validação e a análise de desempenho do código.

4.1 Validação dos dados

A validação dos dados do código próprio foi realizada a partir do teste da ordem de erro de truncamento para o escoamento entre placas paralelas. Já para a cavidade com tampa deslizante, foi feita a comparação com valores da literatura e depois demonstrado que os valores tendem a convergir conforme o aumento da malha. A seguir são apresentadas as validações dos dois casos.

4.1.1 Cavidade com tampa deslizante

A comparação dos dados no escoamento em cavidade com tampa deslizante foi feita tomando como referência dois perfis, o primeiro da velocidade em x em x=0.5 e o segundo da velocidade em y em y=0.5. Pelo escoamento não possuir solução analítica, foi utilizado como referência um método de maior precisão [7].

A tabela 1 mostra os valores da simulação para comparação com os valores da literatura [7].

Reynolds	**N**	**Vel. mesoscópica**	**Resíduo**
1000	512	0.05	$9.92 \cdot 10^{-6}$

Tabela 1: Valores da simulação do escoamento em cavidade Re=1000

A figura 3 mostra o perfil de velocidade em x em x=0.5 do código próprio e do método espectral.

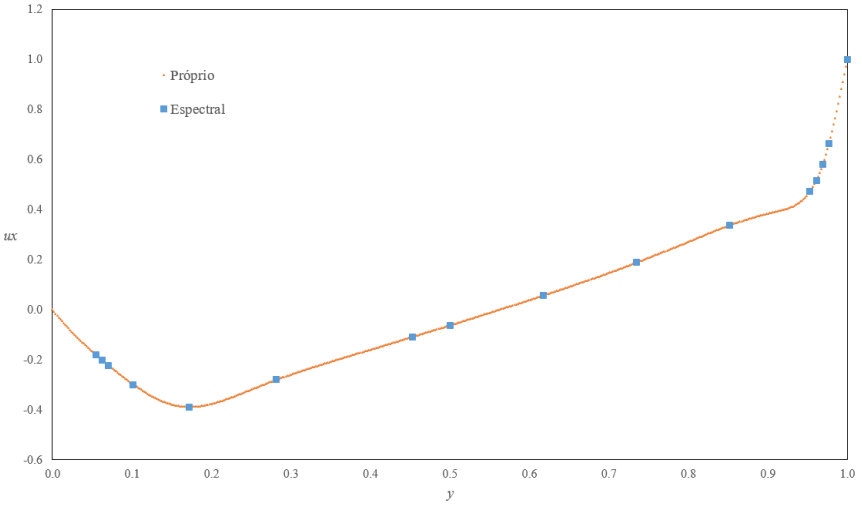

Figura 3: Perfil de velocidade em x em x=0.5 no escoamento de tampa deslizante

A figura 4 mostra o perfil de velocidade em y em y=0.5 do código próprio e do método espectral.

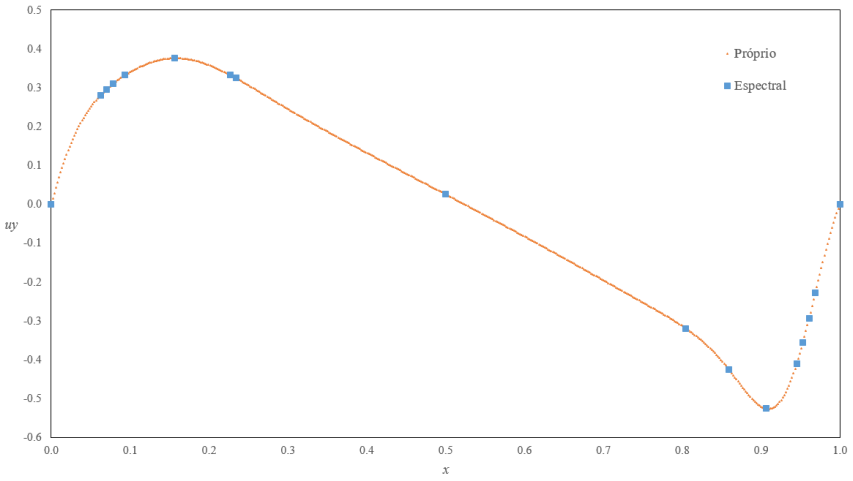

Figura 4: Perfil de velocidade em y em y=0.5 no escoamento de tampa deslizante

É possível perceber que os perfis de velocidade do código próprio se aproximam muito daquelas do método espectral. Sendo que o erro L2 foi de $2.04 \cdot 10^{-3}$ para a velocidade em x e $1.94 \cdot 10^{-3}$ para a em y, valores que estão dentro do aceitável. Logo, pode-se concluir que a simulação foi um sucesso.

Devido ao aquecimento da GPU, não foi possível aumentar a malha e o número de passos conforme necessário para o teste da ordem de truncamento, visto que a malha inicial já era grande. Então o foi utilizado um número de Reynolds igual a 10, para possibilitar a utilização de malhas menores.

A partir disso, foram feitas cinco simulações, dobrando o número de nós, dividindo a velocidade mesoscópica por dois e quadruplicando o número de passos no tempo de uma para outra, para manutenção do tempo físico.

Utilizando a simulação de maior malha como referência, foi calculado o erro L2 de cada simulação. Assim é possível mostrar que os valores tendem a convergir, ficando cada vez mais próximos da solução com o aumento da malha. Os valores dessas simulações são apresentados na tabela 2.

Simulação	**1**	**2**	**3**	**4**	**5**
Reynolds	10	10	10	10	10
N	32	64	128	256	512
Vel. mesoscópica	0.05	0.025	0.0125	0.00625	0.003125
Passos no tempo	4000	16000	64000	256000	1024000
Resíduo	$1.91 \cdot 10^{-7}$	$2.79 \cdot 10^{-10}$	$7.60 \cdot 10^{-13}$	$9.05 \cdot 10^{-14}$	$1.92 \cdot 10^{-13}$

Tabela 2: Valores das simulações do escoamento em cavidade com tampa deslizante

A tabela 3 mostra os erros das simulações de um a quatro com relação à de número cinco.

Simulação	**1**	**2**	**3**	**4**
Vel. em x	$2.45 \cdot 10^{-3}$	$1.39 \cdot 10^{-3}$	$6.88 \cdot 10^{-4}$	$2.29 \cdot 10^{-4}$
Vel. em y	$5.09 \cdot 10^{-2}$	$2.25 \cdot 10^{-2}$	$9.59 \cdot 10^{-3}$	$3.23 \cdot 10^{-3}$

Tabela 3: Erros para simulação de escoamento de cavidade com tampa deslizante

A partir da diminuição do erro conforme o aumento de malha tanto para velocidade em x quanto em y, é possível afirmar que os valores tendem a convergir conforme o aumento da malha.

4.1.2 Placas Paralelas

A validação dos dados no escoamento entre placas paralelas foi feita a partir do perfil da velocidade em x em x=0.5, utilizando como referência a solução analítica, dada pela equação 3.

$$u = 6(y - y^2) \tag{3}$$

Sendo u a velocidade normalizada pela velocidade média e y o valor normalizado pela distância entre placas.

Foram realizadas quatro simulações visando a comparação dos erros obtidos a partir de cada uma. Utilizou-se a mesma relação entre número de nós, velocidade mesoscópica e passos no tempo usada nas simulações de cavidade com tampa deslizante. Os valores das simulações são apresentados na tabela 4.

Simulação	**1**	**2**	**3**	**4**
Reynolds	10	10	10	10
N	64	128	256	512
Vel. mesoscópica	0.1	0.05	0.025	0.0125
Passos no tempo	8000	32000	128000	512000
Resíduo	$4.85 \cdot 10^{-6}$	$1.34 \cdot 10^{-6}$	$3.77 \cdot 10^{-7}$	$9.93 \cdot 10^{-8}$

Tabela 4: Valores das simulações do escoamento entre placas paralelas

A figura 5 mostra um perfil de velocidade obtido a partir da simulação do escoamento entre placas paralelas em comparação com o perfil teórico.

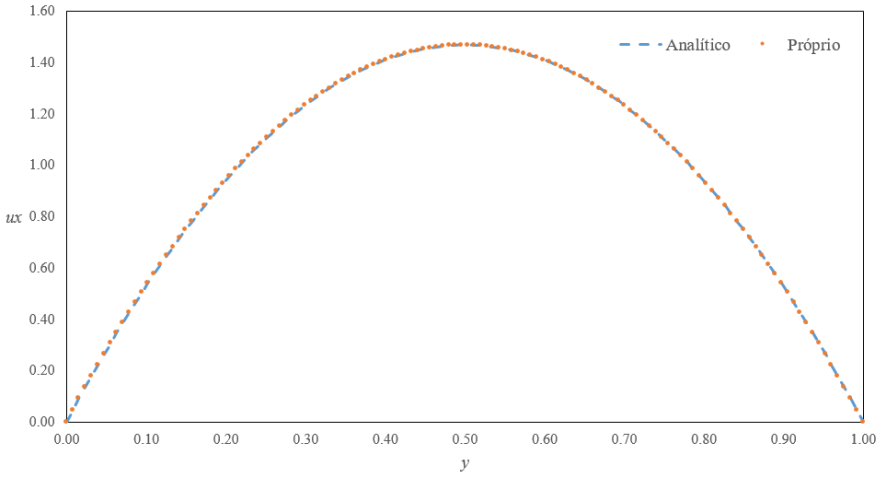

Figura 5: Perfil da velocidade em x no escoamento entre placas paralelas para N=128

Os erros L2 obtidos em cada solução com relação à equação 3 são apresentados na tabela 5.

Simulação	**1**	**2**	**3**	**4**
Erro	$3.27 \cdot 10^{-3}$	$8.28 \cdot 10^{-4}$	$2.07 \cdot 10^{-4}$	$5.18 \cdot 10^{-5}$

Tabela 5: Erros para simulação de escoamento entre placas paralelas

A figura 6 mostra os erros para o escoamento entre placas paralelas e a curva esperada desses.

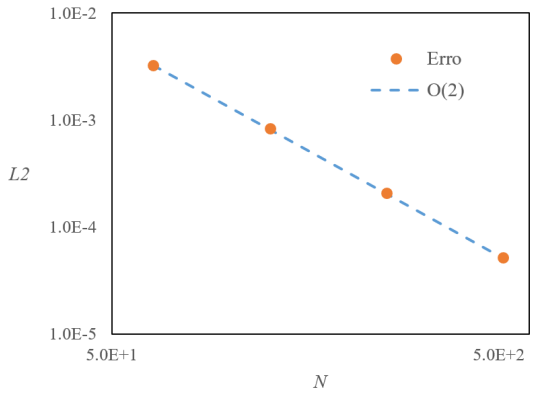

Figura 6: Erro para simulação entre placas paralelas em função do número de nós

Os valores se ajustaram muito bem à curva teórica, o que mostra que a ordem de erro de truncamento para o código próprio é a mesma esperada pela teoria [1].

4.2 Análise de desempenho

Durante o desenvolvimento, foi observado que a variação de alguns parâmetros da simulação e de configuração da GPU afetam o desempenho do código. Os principais são: o número de *lattices*, a frequência de transferência de memória entre GPU e CPU, o número de *threads* por bloco e a configuração do cache.

Foi então feito um teste de referência e depois testes com alteração em um dos parâmetros para análise do impacto dessa no número de MLUPS. Foi utilizada uma placa de vídeo Tesla K20x com ECC desligado.

O caso utilizado foi o de placas paralelas, sendo que foram feitos 5000 passos no tempo para cada simulação. A tabela 6 apresenta o parâmetro dos testes feitos, o valor de MLUPS e o ganho ou perda desse com relação ao teste de referência.

N	Transf. mem.	Nº *Threads*	Cache	MLUPS	Relação
1024	1000 passos	64	`preferL1`	1165.9	-
64	1000 passos	64	`preferL1`	284.1	-75.6%
128	1000 passos	64	`preferL1`	675.9	-42.0%
256	1000 passos	64	`preferL1`	1009.3	-13.4%
512	1000 passos	64	`preferL1`	1134.2	-2.7%
2048	1000 passos	64	`preferL1`	1193.4	2.4%
4096	1000 passos	64	`preferL1`	1190.6	2.1%
1024	100 passos	64	`preferL1`	1010.9	-13.3%
1024	500 passos	64	`preferL1`	1147.3	-1.6%
1024	2000 passos	64	`preferL1`	1181.4	1.3%
1024	5000 passos	64	`preferL1`	1187.0	1.8%
1024	1000 passos	16	`preferL1`	685.1	-41.2%
1024	1000 passos	32	`preferL1`	1122.8	-3.7%
1024	1000 passos	128	`preferL1`	1148.9	-1.5%
1024	1000 passos	256	`preferL1`	1142.6	-2.0%
1024	1000 passos	512	`preferL1`	1126.9	-3.3%
1024	1000 passos	1024	`preferL1`	1107.2	-5.0%
1024	1000 passos	64	`preferShared`	1005.3	-13.8%

Tabela 6: Simulações para análise de desempenho

É possível perceber que para malhas pequenas o código apresenta um desempenho muito menor que para malhas maiores, sendo que a partir de 512 o número de MLUPS se estabiliza, com pequenas variações apenas. Já a transferências de memória é influente apenas se for muito frequente, como no caso que ocorre a cada 100 passos, não apresentando grande relevância no desempenho a partir de certo valor.

O número de *threads* por bloco possui grande impacto apenas para 16 (valor não múltiplo de 32). A partir de 32 o desempenho mostra pequenas variações apenas, sendo que o melhor resultado obtido foi com o valor da referência, 64. A opção de cache se mostrou de influencia considerável no número de MLUPS, apresentando grande importância para o desempenho.

A partir dessas observações é possível decidir quais escolhas fazer para aumentar o desempenho do código. Porém é importante dizer que alguns fatores possuem diferentes graus de relevância conforme o tamanho da malha e as condições de contorno. Por exemplo, a frequência de transferência de memória é mais impactante em malhas pequenas, pois nessas o tempo de computação é pequeno, assim o tempo gasto na transferência tem maior relevância para o número de MLUPS. Logo é pre-

ciso manter isso em mente ao comparar o número de MLUPS de uma simulação para outra.

5 Conclusão

A implementação do LBM em GPU atingiu um nível satisfatório de melhora de desempenho com relação à versão sequencial, com um número de MLUPS cerca de 50 vezes maior. A validação dos dados pode ser considerada um sucesso, apresentando os resultados esperados teoricamente para ambos escoamentos.

Referências

[1] Timm Kruger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The Lattice Boltzmann method: Principles and practice*. Springer, The address, 1 edition, 2017.

[2] Wang Xian and Aoki Takayuki. Multi-gpu performance of incompressible flow computation by lattice boltzmann method on gpu cluster. *Parallel Computing*, 37(9):521 – 535, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[3] NVIDIA. *CUDA Toolkit Documentation*.

[4] NVIDIA. *CUDA C Programming Guide*.

[5] Mark J. Mawson and Alistair J. Revell. Memory transfer optimization for a lattice boltzmann solver on kepler architecture nvidia gpus. *Computer Physics Communications*, 185(10):2566 – 2574, 2014.

[6] M. Januszewski and M. Kostur. Sailfish: A flexible multi-gpu implementation of the lattice boltzmann method. *Computer Physics Communications*, 185(9):2350 – 2368, 2014.

[7] O. Botella and R. Peyret. Benchmark spectral results on the lid-driven cavity flow. *Computers Fluids*, 27(4):421 – 433, 1998.