# Otimizando a Execução de uma Aplicação de Dinâmica dos Fluidos para GPUs *

**Matheus da Silva Serpa, Claudio Schepke**

[1] LEA: Laboratório de Estudos Avançados em Computação
Universidade Federal do Pampa - Campus Alegrete
Av. Tiarajú, 810 - Bairro: Ibirapuitã - Alegrete - RS - CEP: 97546-550

matheusserpa@alunos.unipampa.edu.br, claudioschepke@unipampa.edu.br

***Resumo.*** *A computação em GPUs surge como alternativa para acelerar o desempenho de uma variedade de algoritmos. Nessa arquitetura a dimensão dos blocos de threads impacta diretamente no ganho de desempenho. Nossos experimentos mostraram que blocos entre 64 e 96 obtêm até o dobro de desempenho em relação a blocos maiores que otimizam a ocupação da GPU. Os melhores resultados alcançaram um speedup de 250x em relação a versão sequencial.*

## 1. Introdução

A Computação de Alto Desempenho (HPC) surge como um importante tema de pesquisa na área de Computação, pois com o desenvolvimento e execução de algoritmos paralelos em arquiteturas *multi-core* e *many-core* é possível reduzir o tempo de execução, permitindo solucionar problemas maiores em tempo aceitável [Navarro et al. 2014].

Devido a questões de consumo de energia e dissipação de calor, duas abordagens de sistemas paralelos tem sido adotadas ao invés de *single-core* [Diaz et al. 2012]. Os sistemas *multi-core* costumam ter de duas a doze unidades de processamento (*cores*). Esta arquitetura pode executar programas sequenciais ou paralelos desenvolvidos utilizando interfaces de programação paralela como OpenMP e MPI [Pacheco 2011].

A segunda abordagem é conhecida por *many-core*, pois incorpora de dezenas a centenas de *cores*. Esta abordagem é exemplificada pelos coprocessadores como o Intel Xeon Phi [Jeffers and Reinders 2013] e pelas Unidades de Processamento Gráfico (GPUs) que podem ser programadas com os modelos de programação OpenCL e CUDA [Sanders and Kandrot 2010].

O desenvolvimento de *software* foi afetado por essa mudança de paradigma e diversas aplicações sofreram reengenharias para tornar possível a execução paralela [Sutter and Larus 2005]. Na visão do *framework* CUDA, os fluxos de execução (*threads*) de uma GPU são organizados em uma hierarquia de dois níveis. Um bloco é um conjunto de *threads* e uma grade é um grupo de blocos com tamanho similar.

Neste trabalho investigamos como escolher a dimensão do bloco para otimizar o desempenho de GPUs para uma aplicação desenvolvida em CUDA. Nós consideramos como estado da arte para avaliação o Método de Lattice Boltzmann (MLB), usado para simulação de dinâmica dos fluidos [Chung 2010].

*Trabalho desenvolvido recebendo fomento do Edital N° 05/2014 PROBIC - FAPERGS / UNIPAMPA.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados com o MLB. A Seção 3 apresenta detalhes da nossa versão paralela do MLB. Finalmente, a Seção 4 e a Seção 5 respectivamente, apresentam a discussão e conclusão deste trabalho.

## 2. Trabalhos Relacionados

O MLB é uma das aplicações reais que tem sido referência para avaliações de arquiteturas paralelas pois demanda alta capacidade de processamento.

Habich et al. [Habich et al. 2013] desenvolveram versões otimizadas do MLB D3Q19 para diversas GPUs. Ao avaliar a relação do desempenho com a ocupação os autores mostraram que com a otimização da ocupação o desempenho foi baixo e o pico foi alcançado com ocupação próxima de 50%.

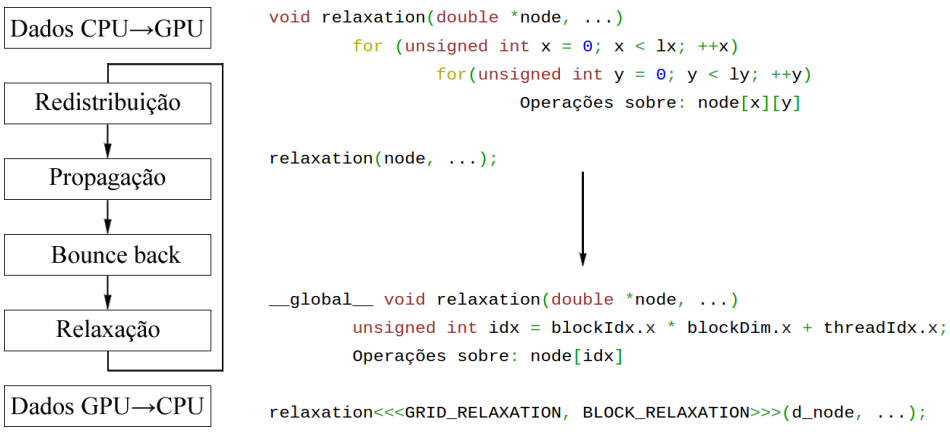
Toelke [Tölke 2010] apresenta diversas otimizações para um MLB D2Q9. Seus experimentos avaliaram as dimensões de bloco de 32 a 256. Os resultados mostram que para uma malha de $1024 \times 1024$ o ideal é blocos de tamanho 64 pois o número de pontos atualizados é maior e consequentemente o tempo de execução é menor.

Volkov [Volkov 2010] mostra técnicas para otimizar implementações e ter melhor desempenho com ocupação baixa. O trabalho cita bibliotecas como CUBLAS e CUFFT que reduziram a dimensão do bloco em até 8x e aumentaram o desempenho em 2x em relação a dimensões que otimizavam a ocupação.

Diferente dos trabalhos relacionados, utilizamos a função da API CUDA que dimensiona automaticamente o tamanho do bloco de forma a otimizar a ocupação. Portanto, investigamos se essa abordagem otimiza o desempenho de GPUs.

## 3. Implementação do Algoritmo

A versão bidimensional do MLB (D2Q9) foi implementada em linguagem C com intuito de resolver problemas de simulação e avaliar arquiteturas paralelas. O laço principal do algoritmo e um exemplo das principais modificações da versão sequencial para versão CUDA são apresentadas na Figura 1.




(a) Laço Principal (b) Modificações gerais da versão CUDA

**Figura 1. Implementação Paralela do Algoritmo**

## 4. Avaliação de Desempenho

A fim de verificar o impacto da dimensão do bloco na execução do MLB, foram feitos experimentos em uma GeForce GT 740M. A GPU possui 384 *CUDA Cores* e 2 GB de memória global. Para os experimentos, foi considerada precisão dupla, memória ECC desligada e quatro tamanhos de malha: $128 \times 128$, $256 \times 256$, $512 \times 512$ e $1024 \times 1024$.

Na Figura 2 mostramos o *speedup* em relação a versão sequencial utilizando um *core* do processador Intel Xeon E5. Para todos os casos, o tempo de execução foi obtido a partir da média de 30 execuções e o desvio padrão foi próximo de 0s.

Nossos experimentos utilizaram três abordagens para escolher a dimensão do bloco. A primeira, utilizando o *framework* CUDA 6.5 que a partir do *kernel* e da quantidade de *threads*, calcula o tamanho do bloco otimizando a ocupação da GPU. Outra abordagem, baseou-se em blocos lineares em relação ao tamanho da malha, 256 para uma malha 256 x 256. A terceira é representada pelas colunas melhor e pior na Figura 2 que são, o melhor e o pior resultado a partir da variação da dimensão do bloco de 32 em 32.

Em relação a função de relaxação, o melhor caso (64 *threads*) foi até 1,41x melhor que a API CUDA (640 *threads*). De acordo com alguns desenvolvedores, maximizar a ocupação da GPU é o melhor meio para otimizar o desempenho na execução. Outros autores, como Volkov [Volkov 2010] mostram que abordagens como aumentar o *Instruction-level parallelism* (ILP) e diminuir a dimensão do bloco funcionam melhor.

Nossos experimentos mostram que os melhores casos são os que tem dimensões menores (64 e 96) e permitem mais blocos ativos (16 e 13). A distribuição linear não escala após 256 pois para blocos de dimensões maiores (512 e 1024) o número de blocos ativos por multiprocessador é menor (2 e 1) e isso afeta a execução da função de relaxação.

A Figura 2(b) apresenta o *speedup* em relação a todo MLB. O *speedup* é menor em relação a Figura 2(a) pois algumas funções fazem pouca computação e mantê-las na GPU se justifica por evitar a cópia dos dados CPU-GPU em cada iteração, o que tornaria o tempo de execução pior que o sequencial. A melhor dimensão foi até 1,45 melhor que a linear e até 1,63x melhor que a API CUDA.

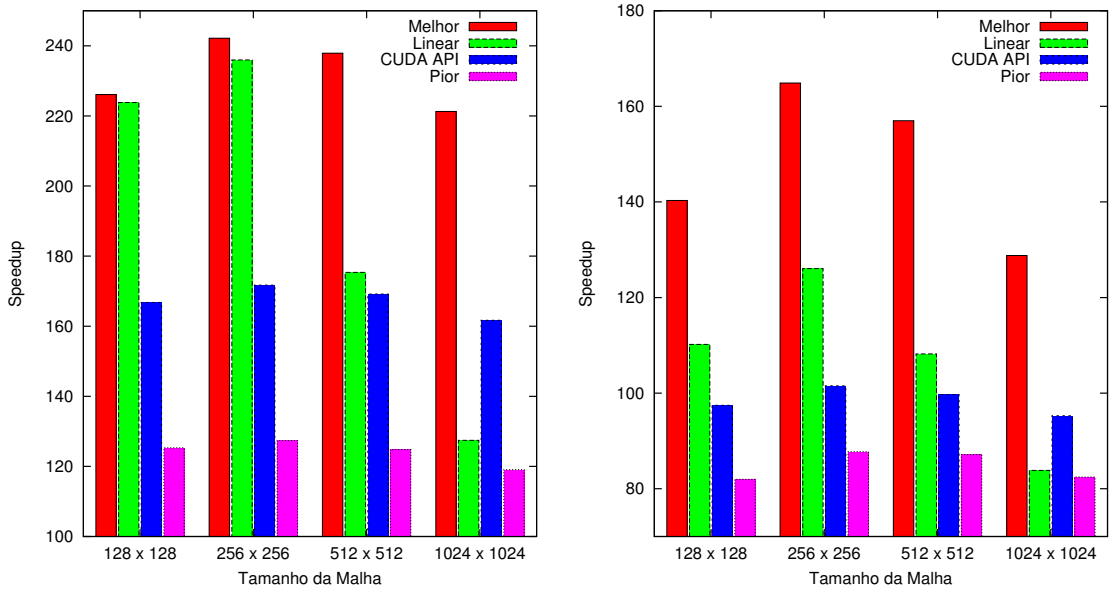
O desempenho menor da dimensão linear em relação a melhor dimensão ocorre, pois, o bloco linear é igual para todas funções e a melhor dimensão varia os blocos otimizando o desempenho função à função. Os pontos onde a API CUDA e a dimensão linear se aproximam são pontos onde as dimensões são iguais ou próximas tal como 512 e 640.

## 5. Conclusão e Trabalhos Futuros

Neste trabalho investigamos o impacto da dimensão do bloco em relação ao tempo de execução de uma versão do MLB para GPUs. Nossos experimentos mostraram que blocos com dimensões entre 32 e 96 otimizam o desempenho da GPU mantendo mais blocos ativos por multiprocessador.

O método da API CUDA que maximiza a ocupação da GPU pode ser utilizado no inicio do desenvolvimento quando um desempenho médio é aceitável, mas para otimizar a execução é necessário testar diversos blocos ou buscar uma heurística melhor.

Trabalhos futuros incluem o desenvolvimento de aplicações em arquiteturas híbridas, compostas por *multi* e *many-core* cooperando para execução simultaneamente.




(a) Função de Relaxação

(b) Método de Lattice Boltzmann

**Figura 2. Ganho de desempenho em relação a versão sequencial**

## Referências


Chung, T. J. (2010). *Computational Fluid Dynamics*. Cambridge University Press.

Diaz, J., Munoz-Caro, C., and Nino, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *Parallel and Distributed Systems, IEEE Transactions on*.

Habich, J., Feichtinger, C., Kostler, H., Hager, G., and Wellein, G. (2013). Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. *Computers & Fluids*, 80(0):276 – 282.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*, volume 1. Morgan Kaufmann.

Navarro, C. A., Hitschfeld-Kahler, N., and Mateu, L. (2014). A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*.

Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue - ACM*.

Tölke, J. (2010). Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39.

Volkov, V. (2010). Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10.