CUDA Acceleration Optimization Based on Lattice Boltzmann Method: Postprint

Authors: Zhang Qianyi, Wei Huajian, He Yinan, Li Huabing, Li Huabing

Date: 2022-10-19T00:00:00+00:00

Abstract

To improve computational efficiency of fluid flow simulations while ensuring result accuracy, acceleration of Poiseuille flow simulations based on the Lattice Boltzmann Method (LBM) was implemented by leveraging the CUDA programming platform and the powerful floating-point computing capabilities of GPUs. Two distinct addressing schemes—linear addressing and index addressing—were designed and applied to various stages of the LBM program, including lattice collision, streaming, and macroscopic quantity computation, to investigate their respective impacts on computational efficiency. Additionally, unified memory management was employed in the program, enabling variables allocated in this manner to be accessed simultaneously from both host and device sides, thereby simplifying code complexity and reducing overhead associated with frequent memory allocation for variables. Using an Intel(R) Xeon(R) E-52620 v4 CPU and an Nvidia Quadro GP100 GPU for computations, speedups of 71× and 25× over CPU serial code were achieved for the linear addressing and index addressing methods, respectively.

Full Text

GPU Accelerated Optimization of Poiseuille Flow Simulation Based on Lattice Boltzmann Method

ZHANG Qianyi, HU Jian, LI Yinan, LI Huabing
School of Material Science and Engineering, Guilin University of Electronic Technology, Guilin 541004, China

Abstract: To improve computational efficiency while ensuring accuracy in fluid simulations, this study leverages the CUDA parallel programming platform and the powerful floating-point computing capabilities of GPUs to accelerate Poiseuille flow simulation based on the Lattice Boltzmann Method. Two distinct addressing approaches—linear addressing and subscript addressing—were

designed and applied to the collision, streaming, and macroscopic quantity calculation steps of the LBM program. The impact of these addressing methods on computational performance was systematically investigated. Unified memory management was employed, enabling variables allocated in this memory space to be accessed seamlessly by both host and device, thereby simplifying code complexity and reducing overhead from frequent memory allocation. Performance evaluations on Intel(R) Xeon(R) E-52620 CPU and Nvidia Quadro GP100 GPU platforms demonstrate significant acceleration.

Keywords: CUDA; lattice Boltzmann method; plane Poiseuille flow; linear addressing; subscript addressing

1. Introduction

With processor clock frequencies approaching their physical limits and the rapid advancement of artificial intelligence and deep learning, single-core computing solutions have become obsolete. GPU-based architectures featuring massive multi-core parallelism have emerged as powerful alternatives, offering accessible programming interfaces that have been successfully applied to computational fluid dynamics. Since NVIDIA introduced CUDA (Compute Unified Device Architecture) in 2007, the barrier to entry for parallel programming has been substantially lowered, catalyzing rapid development across numerous domains.

CUDA' s exceptional computational capabilities and optimization techniques have garnered widespread attention in scientific computing. The Lattice Boltzmann Method (LBM) represents a mesoscopic-scale modeling approach that has found extensive application in fluid mechanics simulations. LBM has been successfully employed to model diverse flow phenomena including porous media flow, blood flow, multiphase flow, and lymphatic flow. Previous studies by Tölke et al. demonstrated TeraFLOP-scale computing for CFD applications using LBM, while other researchers have implemented parallel computations for cavity flow models. The method' s clear physical background, straightforward boundary condition treatment, and inherent parallelism make it exceptionally well-suited for large-scale GPU acceleration.

2. GPU Architecture and CUDA Programming Model

2.1 GPU Architecture Fundamentals

GPUs differ fundamentally from CPUs in their architectural design philosophy. While CPUs prioritize versatility for handling diverse data types, complex logic operations, and branch prediction—requiring sophisticated control units and substantial cache memory—GPUs are optimized for data-parallel tasks with relatively simple control flow. GPUs employ thousands of streamlined cores and arithmetic logic units (ALUs) paired with simplified control logic and memory

systems, making them ideal for computationally intensive, massively parallel workloads.

The CUDA programming model operates with two distinct memory spaces: host memory (CPU) and device memory (GPU), each with independent address spaces. A typical CUDA device comprises multiple Streaming Multiprocessors (SMs), with each SM containing numerous Streaming Processors (SPs) and associated hardware resources. During kernel execution, threads from a thread block are dispatched to SMs in groups called warps, which serve as the fundamental scheduling unit.

2.2 CUDA Memory Hierarchy

CUDA implements a sophisticated memory hierarchy with multiple cache levels to optimize data access efficiency. The primary memory types include registers, shared memory, global memory, constant memory, texture memory, and local memory. Each SM features independent first-level cache while sharing a unified second-level cache across the device. Registers provide the fastest access and are allocated per-thread. When register usage exceeds available capacity, the compiler spills data to local memory in DRAM. Shared memory represents a controllable first-level cache visible only to threads within the same block, enabling explicit programmer optimization for data reuse and inter-thread communication.

3. Lattice Boltzmann Method Theory

The Lattice Boltzmann Method evolved from lattice gas cellular automata, where Boolean variables were replaced by continuous single-particle distribution functions $f_i(x,t)$. The fundamental evolution equation is:

$$f_i(x + e_i\delta t, t + \delta t) - f_i(x,t) = \Omega_i$$

where Ω_i represents the collision operator. Using the Bhatnagar-Gross-Krook (BGK) approximation with a single relaxation time τ, the collision term simplifies to:

$$f_i(x + e_i\delta t, t + \delta t) - f_i(x,t) = -\frac{f_i - f_i^{(eq)}}{\tau}$$

Here, $f_i^{(eq)}$ is the local equilibrium distribution function and τ is the relaxation time controlling the viscosity. The D2Q9 model employs nine discrete velocity directions ($i = 0, 1, \ldots, 8$) with weights ω_i and lattice velocities e_i. The equilibrium distribution is:

$$f_i^{(eq)} = \rho\omega_i \left[1 + \frac{3(e_i \cdot u)}{c^2} + \frac{9(e_i \cdot u)^2}{2c^4} - \frac{3u^2}{2c^2}\right]$$

where ρ is macroscopic density and u is fluid velocity. The weights are $\omega_0 = 4/9$, $\omega_{1-4} = 1/9$, and $\omega_{5-8} = 1/36$.

4. Algorithm Implementation and Optimization

4.1 Computational Framework

A typical LBM simulation involves: (1) variable declaration and distribution function initialization, (2) collision and streaming steps with boundary treatment, (3) macroscopic quantity calculation, and (4) convergence checking. The iterative collision-streaming cycle dominates computational cost, making it the primary target for GPU parallelization.

This study models lymphatic flow as Poiseuille flow in a circular pipe with constant cross-section. Three implementation schemes were developed to evaluate addressing strategies:

Scheme 1 (Serial CPU): Uses three-dimensional arrays `df[i][j][k]` for distribution functions, where `i` and `j` are spatial indices and `k` is the velocity direction. This serial implementation serves as the performance baseline.

Scheme 2 (GPU with Linear Addressing): Employs unified memory via `cudaMallocManaged()` to allocate a single linear memory block for the distribution function: `double *df` with size `width $\times$ height $\times$ 9`. Four kernel functions handle parallel execution: `addKernelCollide()`, `addKernelCopy()`, `addKernelStream()`, and `addKernelCalculate()`. Each thread computes a global linear index:

$$\text{id} = k + (j + i \times \text{height}) \times 9$$

enabling coalesced memory access patterns.

Scheme 3 (GPU with Subscript Addressing): Constructs multi-dimensional array semantics on top of linear memory using pointer arrays. First, a linear block `df0` is allocated, then `df1[i*height+j]` pointers are set to `&df0[(i*height+j)*9]`, and finally `df[i]` points to `&df1[i*height]`. This allows natural `df[i][j][k]` syntax while maintaining unified memory benefits. The same kernel functions as Scheme 2 are used, but with subscript-based access.

4.2 Kernel Function Design

The collision kernel `addKernelCollide<<<grid,block>>>(df,iSol,⋯,dev_p)` launches a 2D thread grid where each thread computes:

$$\text{id} = \text{adr}(i, j, k) = k + (j + i \times \text{height}) \times 9$$

and updates the distribution function:

$$df[\text{id}] = df[\text{id}] - \frac{df[\text{id}] - f^{(eq)}(k, \rho, u_x, u_y)}{\tau}$$

The streaming kernel uses shared memory to load nine distribution values per thread block, performs the propagation step, and writes results back to global memory. The macroscopic quantity kernel similarly leverages shared memory to compute density and velocity components through efficient parallel reduction.

5. Results and Analysis

Performance evaluations were conducted on a server equipped with an Intel(R) Xeon(R) E-52620 CPU and an Nvidia Quadro GP100 GPU (CUDA 10.0, theoretical bandwidth 720 GB/s, double-precision performance 5.3 TFlop/s). The test case was plane Poiseuille flow, with all simulations running for a fixed number of iterations to ensure result consistency.

All three schemes produced identical simulation results, validating the correctness of the parallel implementations. Scheme 2 (GPU linear addressing) achieved approximately [FIGURE:N]× speedup over the serial CPU baseline. Scheme 3 (GPU subscript addressing) reduced computation time by nearly two-thirds compared to Scheme 1, demonstrating that subscript addressing is particularly advantageous for GPU iterative computations. In contrast, CPU-based subscript addressing showed negligible performance improvement, indicating this optimization is GPU-specific.

The superior performance of subscript addressing on GPU stems from improved memory access patterns and compiler optimization opportunities when using multi-dimensional array semantics. The unified memory management significantly reduced development complexity while maintaining high performance.

6. Conclusion

This study successfully accelerated Lattice Boltzmann Method simulations of Poiseuille flow using CUDA-enabled GPUs. Two addressing methods were implemented and evaluated: linear addressing and subscript addressing. The results demonstrate that GPU parallelization yields substantial performance improvements, with subscript addressing providing the best efficiency for iterative LBM computations. The work highlights the tremendous potential of GPU-CUDA architectures for large-scale scientific computing applications.

References

[1] NVIDIA Corporation. *NVIDIA CUDA Programming Guide 10.1* [EB/OL]. (2019-11-28) [2020-11-10]. https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html.

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

[3] Chen H, Chen S. Lattice Boltzmann model for simulation of magnetohydrodynamics[J]. *Physical Review Letters*, 1991, 67(27): 3776-3779.

[4] Shan X, Chen H. Lattice Boltzmann model for simulating flows with multiple phases and components[J]. *Physical Review E*, 1993, 47(3): 1815-1819.

[5] Munn LL, et al. The effects of valve leaflet mechanics on lymphatic pumping assessed using numerical simulations[J]. *The FASEB Journal*, 2020, 34(1): 7043.

[6] McNamara GR, Zanetti G. Use of the Boltzmann equation to simulate lattice-gas automata[J]. *Physical Review Letters*, 1988, 61(20): 2332.

[7] Qian YH, d'Humières D, Lallemand P. Lattice BGK models for Navier-Stokes equation[J]. *Europhysics Letters*, 1992, 17(6): 479-484.

[8] Chen S, Matthaeus WH. Recovery of the Navier-Stokes equations using a lattice-gas Boltzmann method[J]. *Physical Review A*, 1992, 45(8): 5339-5342.

[9] Abdulmajeed A. *Lattice Boltzmann Method: Theory and Engineering Applications*[M]. Beijing: Electronic Industry Press, 2015: 20-22.

[10] Zheng Y, et al. Lattice Boltzmann method for blood flow simulation[J]. *Science Technology and Engineering*, 2010, 10(7): 56(28): 2434-2444.

[11] Li H. *Lattice Boltzmann Method for Flow Fields: Algorithm Design and Program Optimization*[M]. Dalian: Dalian University of Technology Press, 2013: 47-53.

[12] Zhang Q, et al. GPU-CUDA based lattice Boltzmann method for porous media flow[J]. *Journal of Guilin University of Electronic Technology*, 2015(1): 20-26.

Editor: Zhang Suobin

Note: Figure translations are in progress. See original paper for figures.