# Moment Representation of Regularized Lattice Boltzmann Methods on NVIDIA and AMD GPUs

Pedro Valero-Lara, Jeffrey S. Vetter, John Gounley
Oak Ridge National Laboratory
Oak Ridge, TN, USA
{valerolarap},{vetter},{gounleyjp}@ornl.gov

Amanda Randles
Duke University
Durham, NC, USA
amanda.randles@duke.edu

## Abstract

The lattice Boltzmann method is a highly scalable Navier-Stokes solver that has been applied to flow problems in a wide array of domains. However, the method is bandwidth-bound on modern GPU accelerators and has a large memory footprint. In this paper, we present new 2D and 3D GPU implementations of two different regularized lattice Boltzmann methods, which are not only able to achieve an acceleration of ~1.4× w.r.t. reference lattice Boltzmann implementations but also reduce the memory requirements by up to 35% and 47% in 2D and 3D simulations respectively. These new approaches are evaluated on NVIDIA and AMD GPU architectures.

***CCS Concepts:*** • **Computing methodologies → Parallel algorithms**; *Modeling and simulation.*

***Keywords:*** Lattice Boltzmann Method, Computational Fluid Dynamics, GPU, CUDA, HIP, NVIDIA, AMD

**ACM Reference Format:**
Pedro Valero-Lara, Jeffrey S. Vetter, John Gounley and Amanda Randles. 2018. Moment Representation of Regularized Lattice Boltzmann Methods on NVIDIA and AMD GPUs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Over the past decade, graphics processing unit (GPU) accelerators have become a standard platform for researchers conducting computational fluid dynamics (CFD) simulations with the lattice Boltzmann method (LBM). The inherently data-parallel nature of the lattice Boltzmann algorithm has led to strong performance on GPUs and the factors related to this performance have been well-studied. In this paper, we evaluate whether a recently developed alternative LBM algorithm that takes advantage of GPU shared memory and regularization can outperform the standard LBM algorithm [3]. For this comparison, we extend this new algorithm to 3D and to recursive regularization for the first time and evaluate performance on NVIDIA and AMD GPUs.

Advanced strategies for an efficient implementation of computationally intensive lattice Boltzmann methods on GPUs have long been a focus of research. The potential for GPUs to improve the performance of LBM motivated very early porting efforts [21]. Since then, methods have been developed to make efficient use of global memory capacity [15], overlapping communication and computation [9], exploit asynchronous communication [11], identify optimal data access patterns [4, 14], mesh refinement [17–19], solid-fluid interaction, among many others.

One line of research for GPU-focused lattice Boltzmann algorithms has focused on exploiting shared memory. Shared memory allows for communication between threads within the same thread block. Early LBM implementations made significant use of shared memory to ensure data stayed in the cache between reads from and writes to global memory [13, 15]. However, the practice fell out of fashion as increases in GPU cache sizes and more efficient LBM algorithms were able to minimize problematic cache evictions. Subsequently, a number of implementations were successful in obtaining near-peak performance with respect to this bound [16].

This work extends the recently published work [3], a 2D implementation of the moment representation of the regularized lattice Boltzmann method used GPU shared memory to store data during mappings between LBM's moment and distribution representations. By ensuring this data remained in the cache and could be communicated between threads within a block, this implementation was able to store the simulation state in global memory using a lossless compressed moment representation. By significantly reducing the amount of data being written to and from global memory, this proof-of-concept 2D implementation was able to outperform a standard LBM implementation on an NVIDIA GPU. Other recent studies [2, 12] have focused on applications of the moment representation to simulations of soft matter and turbulent flows. The contributions of this study that significantly expand on previous studies are as follows:


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1. The implementation of a new regularized approach based on recursive regulations instead of projective regularization for lattice Boltzmann method.
2. The implementation of both approaches (recursive and projective regularization) for 2D and 3D lattice Boltzmann simulations using NVIDIA and AMD GPUs.
3. A detailed performance analysis on two different GPU architectures.

The rest of the paper is organized as follows: Section 2 presents the formulation of the lattice Boltzmann methods and the GPU algorithms are described in Section 3. Performance analysis is carried out in Section 4 for 2D and 3D LBM lattices on NVIDIA and AMD GPUs. Finally, the conclusions and future directions are summarized in Section 5.

## 2 Lattice Boltzmann method

The LBM is an explicit Navier-Stokes solver for weakly compressible flows with lattice-symmetry characteristics which respect the conservation of the macroscopic moments [10]. LBM does this by modeling the fluid as a distribution function of microscopic particles. These dual microscopic and macroscopic aspects are key features of the mesoscopic method and have significant implications for LBM regularization schemes.

### 2.1 Standard lattice Boltzmann method

The standard lattice Boltzmann method focuses on the evolution of the particle distribution function $f$ on a fixed Cartesian lattice with a set of $Q$ discrete velocities $\mathbf{c}_i$. The first three moments of the distribution $f$ correspond to density $\rho$, momentum $\rho\mathbf{u}$, and a second order tensor $\mathbf{\Pi}$ related to momentum flux. These moments are evaluated using the discrete velocities $\mathbf{c}_i$ or, equivalently, the Hermite polynomials $\mathscr{H}$

$$\rho = \sum_{i=1}^{Q} f_i = \sum_{i=1}^{Q} \mathscr{H}^{(0)} f_i \tag{1}$$

$$\rho u_\alpha = \sum_{i=1}^{Q} c_{i\alpha} f_i = \sum_{i=1}^{Q} \mathscr{H}^{(1)}_\alpha f_i \tag{2}$$

$$\Pi_{\alpha\beta} = \sum_{i=1}^{Q} \left(c_{i\alpha}c_{i\beta} - c_s^2\delta_{\alpha\beta}\right) f_i = \sum_{i=1}^{Q} \mathscr{H}^{(2)}_{\alpha\beta} f_i. \tag{3}$$

From the first two (hydrodynamic) moments, the Maxwell-Boltzmann equilibrium distribution $f^{eq}$ can be approximated as

$$f_i^{eq} = \omega_i \rho \left( \mathscr{H}^{(0)} + \frac{1}{c_s^2}\mathscr{H}^{(1)}_\alpha u_\alpha + \frac{1}{2c_s^4}\mathscr{H}^{(2)}_{\alpha\beta} u_\alpha u_\beta \right), \tag{4}$$

for lattice weights $\omega_i$ and lattice speed of sound $c_s^2$ [10]. The corresponding non-equilibrium distribution $f^{neq}$ is trivially computed as $f^{neq} = f - f^{eq}$.

The evolution of particle distribution $f$ is formulated in terms of these equilibrium and non-equilibrium terms by the lattice Boltzmann equation

$$f_i(\mathbf{x} + \mathbf{c}_i, t+1) = f_i^{eq}(\mathbf{x}, t) + \left(1 - \frac{1}{\tau}\right) f_i^{neq}(\mathbf{x}, t), \tag{5}$$

for relaxation time $\tau$, lattice site $\mathbf{x}$, and timestep $t$. The lattice Boltzmann equation is best understood as the combination of two steps: collision and streaming. In collision, local inter-particle interaction leads to a relaxation toward the equilibrium distribution, which is computed as

$$f_i^*(\mathbf{x}, t) = f_i^{eq}(\mathbf{x}, t) + \left(1 - \frac{1}{\tau}\right) f_i^{neq}(\mathbf{x}, t), \tag{6}$$

for post-collision distribution $f^*$. During streaming, conversely, post-collision distribution components advance on along the lattice according to their discrete velocities:

$$f_i(\mathbf{x} + \mathbf{c}_i, t+1) = f_i^*(\mathbf{x}, t) \tag{7}$$

### 2.2 Projective regularization

The fundamental idea of projective regularization in LBM is to reduce the number of degrees of freedom by performing the collision operation in the space of the Hermite polynomials [5]. In a second-order approximation of the athermal Navier-Stokes equations, this space corresponds to the set of the first three sets of moments $\mathscr{M} = \{\rho, \rho\mathbf{u}, \mathbf{\Pi}\}$. For a $D$ dimensional simulation with $Q$ distribution components, these moments account for a total of $M = 1 + D + \frac{(D)(D+1)}{2}$ degrees of freedom in this moment space.

In projective regularization, the BGK algorithm is modified by projecting the non-equilibrium distribution $f^{neq}$ into moment space. As hydrodynamic moments are conserved, this results only in the non-equilibrium second order moment $\Pi^{neq}$

$$\Pi^{neq}_{\alpha\beta} = \sum_{i=1}^{Q} \mathscr{H}^{(2)}_{\alpha\beta} f_i^{neq}. \tag{8}$$

This second order moment is mapped back to distribution space and replaces $f_i^{neq}$ in equation 6, resulting in the following projective regularization collision operation,

$$f_i^* = f_i^{eq} + \left(1 - \frac{1}{\tau}\right) \frac{\omega_i}{2c_s^4} \mathscr{H}^{(2)}_{\alpha\beta} \Pi^{neq}_{\alpha\beta}. \tag{9}$$

We observe that the projective regularization collision operation can be equivalently formulated in moment space,

$$\Pi^*_{\alpha\beta} = \Pi^{eq}_{\alpha\beta} + \left(1 - \frac{1}{\tau}\right) \Pi^{neq}_{\alpha\beta}, \tag{10}$$

for equilibrium second order moment $\Pi^{eq}_{\alpha\beta} = \rho u_\alpha u_\beta$. The post-collision distribution $f_i^*$ is then recovered by the equation:

$$f_i^* = \omega_i \left( \mathscr{H}^{(0)}\rho + \frac{1}{c_s^2}\mathscr{H}^{(1)}_\alpha \rho u_\alpha + \frac{1}{2c_s^4}\mathscr{H}^{(2)}_{\alpha\beta}\Pi^*_{\alpha\beta} \right). \tag{11}$$

In this latter formulation, equations 8, 10, and 11 replace the collision kernel in equation 6, mapping back to distribution space in order to perform the subsequent streaming operation. We observe that projective regularization adds a

modest amount of additional floating point operations but does not substantially alter the computational profile.

### 2.3 Recursive regularization

Recursive regularization builds on its projective counterpart by approximating higher-order moments from the first three moments $\mathscr{M} = \{\rho, \rho\mathbf{u}, \mathbf{\Pi}\}$. By recursively deriving non-equilibrium components of the higher-order moments in this way, a complete Hermite polynomial basis with $Q$ moments is obtained in the moment space [8]. The resulting algorithm differs from projective regularization in several important respects, improving numerical stability but adding computational complexity [7].

Like projective regularization, recursive regularization begins with computing $\Pi^{neq}_{\alpha\beta}$ using equation 8. Subsequently, the non-equilibrium components of the third and fourth order moments $\mathbf{a}_{(3)}$ and $\mathbf{a}_{(4)}$ are derived from a set of recursion relations, respectively. As these relations and their derivation are too lengthy to recapitulate here, the reader is referred to [8]. For the purpose of this study, it is sufficient to note that $\mathbf{a}^{neq}_{(3)}$ is a function of $\rho$, $\mathbf{u}$, and $\mathbf{\Pi}^{neq}$, while $\mathbf{a}^{neq}_{(4)}$ is a function of the aforementioned moments and $\mathbf{a}^{neq}_{(3)}$. When combined with the equilibrium moments $\mathbf{a}^{eq}_{(3)} = \rho\mathbf{uuu}$ and $\mathbf{a}^{eq}_{(4)} = \rho\mathbf{uuuu}$, the third and fourth order moments $\mathbf{a}_{(3)}$ and $\mathbf{a}_{(4)}$ are fully approximated.

Analogous to the moment space formulation of projective regularization, collision is performed for non-conserved moments, for $\mathbf{\Pi}$ using equation 10 and for the higher order moments with the following operations:

$$\mathbf{a}^{*}_{(3)} = \mathbf{a}^{eq}_{(3)} + \left(1 - \frac{1}{\tau}\right)\mathbf{a}^{neq}_{(3)} \tag{12}$$

$$\mathbf{a}^{*}_{(4)} = \mathbf{a}^{eq}_{(4)} + \left(1 - \frac{1}{\tau}\right)\mathbf{a}^{neq}_{(4)}. \tag{13}$$

Subsequently, the post-collision distribution $f^{*}_{i}$ is obtained by extending equation 11 to higher order moments:

$$f^{*}_{i} = \omega_i\Big(\mathscr{H}^{(0)}\rho + \frac{1}{c_s^2}\mathscr{H}^{(1)}_{\alpha}\rho u_\alpha + \frac{1}{2c_s^4}\mathscr{H}^{(2)}_{\alpha\beta}\Pi^{*}_{\alpha\beta} + \frac{1}{2c_s^6}\mathscr{H}^{(3)}_{\alpha\beta\gamma}a^{*}_{(3)\alpha\beta\gamma} + \frac{1}{4c_s^8}\mathscr{H}^{(4)}_{\alpha\beta\gamma\delta}a^{*}_{(4)\alpha\beta\gamma\delta}\Big). \tag{14}$$

We observe that, in contrast to projective regularization, the computational complexity of recursive regularization is somewhat higher, owing to the derivation of equlibrium and non-equilibrium components of the higher order moments.

## 3 GPU implementation of LBM

### 3.1 Implementation of the 2 lattice distribution representation

LBM is amenable to fine granularity (one thread per lattice node) being the computation of every *lattice* point is independent with respect to the others. In the standard distribution representation, we need two different lattices ($f_1$ and $f_2$ in Algorithm 1) to compute the streaming stage in parallel.

**Algorithm 1** Standard distribution representation LBM in 3D

```
for ind = 1 → Nx · Ny · Nz do
    lattice_speeds = 19 or 27
    Streaming
    for i = 1 → lattice_speeds do
        x_stream = x − c_x[i]
        y_stream = y − c_y[i]
        z_stream = z − c_z[i]
        ind_stream = z_stream · Nx · Ny + y_stream · Nx + x_stream
        f[i] = f_1[i][ind_stream]
    end for
    for i = 1 → lattice_speeds do
        ρ+ = f[i]
        u_x+ = c_x[i] · f[i]
        u_y+ = c_y[i] · f[i]
        u_z+ = c_z[i] · f[i]
    end for
    u_x = u_x/ρ
    u_y = u_y/ρ
    u_z = u_z/ρ
    Collision
    for i = 1 → lattice_speeds do
        cu = c_x[i] · u_x + c_y[i] · u_y + c_z[i] · u_z
        tmp = (u_x)^2 + (u_y)^2 + (u_z)^2
        f_eq = ω[i] · ρ · (1 + 3 · cu + cu^2 − 1.5 · tmp)
        f_2[i][ind] = f[i] · (1 − 1/τ) + f_eq · 1/τ
    end for
end for
```

Depending on the ordering of the LBM steps (collision and streaming), two different strategies arise – 'push' and 'pull' – which have important consequences in terms of performance [16]. In the push configuration, collision is performed before streaming, while the pull configuration performs these in the opposite order. Introduced by [22], the pull configuration is considered the fastest GPU implementation of the standard distribution representation of LBM [16]. As presented in Algorithm 1, the pull approach minimizes the pressure on memory versus the push configuration, as the macroscopic level can be completely computed on top regions of the memory hierarchy. For the pull configuration, we use a 1D grid of 1D block of threads, in which each GPU thread performs a complete LBM update on a single lattice node.

Memory management is of vital importance in GPU implementations of LBM, given the high latency of the GPU memory. To achieve a high memory bandwidth, it is necessary that the memory accesses are carried out in a coalesced pattern, with consecutive threads accessing adjacent memory locations. To achieve this, we use the structure of array (SoA) addressing scheme for the LBM distribution array, which has proven to be very efficient in multicore and GPU architectures [16].

**Algorithm 2** Moment representation of LBM in 3D

**Define** $colSize_x$
**Define** $colSize_y$
**Let** $numCols = (N_x \cdot N_y)/(colSize_x \cdot colSize_y)$
**Let** $numLayers = N_z$
$\#Moments = 10$
$lattice\_speeds = 19$ *or* $27$
**Global memory array**
$Moment_{GPU}[N_x \cdot N_y \cdot \#Moment]$
**Shared memory array**
$F_{shared}[lattice\_speeds]$
**Local memory array**
$Moment_{local}[\#Moment]$
**for** $column = 1 \rightarrow numCols$ **do**
  **for** $layer = 1 \rightarrow numLayers$ **do**
    **Read moments from global memory (Fig. 1)**
    $\rho = Moment_{GPU}[ind_{GPU}(column, layer, ..., 0)]$
    $u_x = Moment_{GPU}[ind_{GPU}(column, layer, ..., 1)]$
    $u_y = Moment_{GPU}[ind_{GPU}(column, layer, ..., 2)]$
    $u_z = Moment_{GPU}[ind_{GPU}(column, layer, ..., 3)]$
    **for** $m = 4 \rightarrow \#Moments$ **do**
      $ind = ind_{GPU}(col, layer, ..., m)$
      $Moment_{local}[m] = Moment_{GPU}[ind]$
    **end for**
    **Perform collision (Eq.10)**
    **for** $m = 4 \rightarrow \#Moments$ **do**
      $tmp = (1 - \frac{1}{\tau})\ Moment_{local}[m]$
      $Moment_{local}[m] = Moment_{local}[m] + tmp$
    **end for**
    **Map moments to f (Eq.11)**
    **for** $l = 1 \rightarrow lattice\_speeds$ **do**
      $ind = shared\_ind(l, col, layer, ...)$
      $F_{shared}[ind] = Stream(l, Moment_{local}, \rho, u_x, ...)$
    **end for**
    **Synchronization**
    **Map** $f$ **to moments in GPU memory** $(Eq.8)$
    **for** $m = 1 \rightarrow \#Moments$ **do**
      **for** $l = 1 \rightarrow lattice\_speeds$ **do**
        $ind = shared\_ind(l, ...)$
        $Moment_{local}[m] += Moment(F_{shared}[ind])$
      **end for**
    **end for**
    **Write back moments to Global GPU memory**
    **for** $m = 1 \rightarrow \#Moments$ **do**
      $ind = GPU\_ind(column, layer, ..., m)$
      $Moment_{GPU}[ind] = Moment_{local}[m]$
    **end for**
  **end for**
**end for**

### 3.2 Implementation of the 1 lattice moment representation

While the projective and recursive regularization schemes in sections 2.2 and 2.3 can be implemented in the distribution representation from the previous section, their simulation states being completely represented by the first three moments $\mathscr{M}$ enables a more efficient approach based on moments. The fundamental idea of the moment representation (MR) implementation is to only read and write to and from global memory at the moment space [20]. However, to retain LBM's property of exact streaming, the streaming operation must be performed in the distribution space, necessitating mapping between distribution and moment spaces before and after streaming. This basic form has several significant implications for implementation design, both in terms of the algorithm itself and how the lattice sites are associated with thread blocks on the GPU.

The major steps of the MR implementation for projective regularization are illustrated in Algorithm 2 with the *push* configuration of collision first and streaming second. In contrast, the GPU implementation of MR differs significantly from both the CPU version and from the standard LBM implementation discussed in Section 3.1. The most important differences between the implementation of the distribution and moment representations fall into (1) thread distribution, (2) memory access pattern, and (3) shared memory usage.

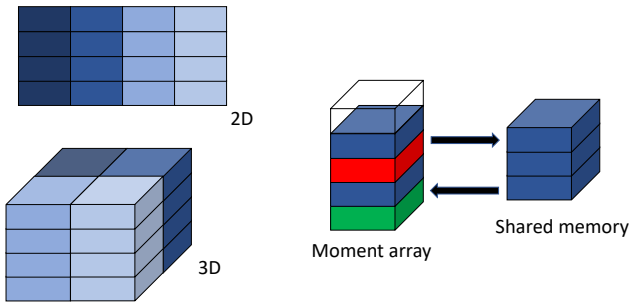


**Figure 1.** Blocks of threads distributions of the *lattice* nodes for the moment-based representation of the LBM. During a timestep, moments from a given layer (red layer) are read from memory, collision is performed and the post-collision distribution are streamed to the corresponding locations in shared memory, and the moments corresponding to two layers earlier are computed and written back (green layer).

Unlike the thread distribution used in the standard representation, the fluid domain is decomposed into columns, as illustrated in 2D and 3D in figure 1, and each column is associated with a specific block of threads. Each column is composed of a number of tiles, each one or more lattice points high. The number of threads in the thread block is equal to the number of lattice points in a tile, along with a one lattice point wide halo in the non-axial dimensions of the column. This halo is necessary to allow distribution components to stream into the column from adjacent columns. In 2D, for a tile with dimensions $x_t \times y_t$ in a column parallel to the $y$-axis, the number of threads in the thread block is $(x_t + 2) \times y_t$. Similarly, in 3D, for a tile with dimensions

$x_t \times y_t \times z_t$ in a column parallel to the $z$-axis, the number of threads in the thread block is $(x_t + 2) \times (y_t + 2) \times z_t$.

Each block within the column is updated via a sliding window algorithm that uses circular array time shifting. The purpose of this approach is to maximize data reuse and to avoid race conditions while columns are updated simultaneously. At each iteration, the thread block starts by reading from global memory the moments $\mathscr{M}$ from all $N$ lattice points in the bottom tile and its halo. For each lattice point, collision is performed in moment space using equation 10, and the post-collision moments are mapped to distribution space using equation 11. To perform streaming, the post-collision distribution components are written to an array in shared memory, into array indices associated with the lattice site to which the distribution component is streaming. We note that streaming is performed only for distribution components whose streaming destination is within the column – those streaming into other columns are handled by the halos of those other columns. Moreover, to account for distribution components that stream up or down, out of the tile but within the column, the shared memory distribution array must include two additional layers beyond the size of the tile. Consequently, the shared memory size is $x_t \times (y_t + 2) \times Q$ double precision values for a 2D fluid domain and $x_t \times y_t \times (z_t + 2) \times Q$ for a 3D fluid domain. This process is repeated for subsequent tiles, moving from the bottom to the top of the fluid domain.

After a given tile has streamed into shared memory and synchronization has been performed, all distribution components have streamed into the shared memory locations associated with the tile directly below it. Once this has occurred, the moments $\mathscr{M}$ on this tile are recomputed using equations 1 – 3 and written back to global memory. To avoid a race condition with reads being performed by adjacent columns, circular array shifting is employed, as illustrated on the right side of figure 1 [1].

We observe that there are several implementation details to keep in mind. First, optimal performance is achieved with two or more thread blocks per SM, so the targeted tile size and shared memory usage per column must be adjusted to account for this. Second, for 3D fluid domains, tiles that are more than one lattice point high – and, consequently, have a 3D block of threads - consistently underperform those that are a single lattice point high, due to poorer memory access patterns.

## 4 Performance Evaluation

To understand the performance gain of the LBM moment representation (MR) on GPUs, we evaluated the performance reached against a reference implementation of the standard distribution representation (ST). Versions of the moment representation approach using the projective and recursive regularization collision kernels are denoted MR-P and MR-R, respectively.

| GPU Arch. | NVIDIA V100 | AMD MI100 |
|---|---|---|
| Frequency | 1,455 MHz | 1,502 MHz |
| CUDA/HIP Cores | 5,120 | 7,680 |
| SM/CU counts | 80 | 120 |
| Shared Mem. | up to 96 KB per SM | 64KB per CU |
| L1 | up to 96 KB per SM | 16 KB per CU |
| L2 (unified) | 6,144 KB | 8,192 KB |
| Memory | HBM2 16 GB | HBM2 32 GB |
| Bandwidth | 900 GB/s | 1228.86 GB/s |
| Compiler | nvcc v11.0.221 | hipcc 4.2 |

**Table 1.** Summary of the main features of the NVIDIA V100 and AMD MI100 GPUs

All three approaches – ST, MR-P, and MR-R – are implemented in CUDA and HIP proxy applications. They simulate flow in a rectangular 2D or 3D channel, using bounceback boundary conditions at the channel walls and finite difference boundary conditions at the inlet and outlet [6]. Two common single-speed LBM lattices are considered: D2Q9 and D3Q19.

### 4.1 Performance modeling

We measure the performance on two relatively comparable GPU architectures, the NVIDIA V100 GPU and the AMD MI100. Metrics for both devices are listed in Table 1. Performance of the methods is measured using MFLUPS, a standard metric for LBM throughput which stands for the number of million lattice updates per second. These measurements are compared with predictions from a theoretical roofline model. Additionally, we use measurements from GPU profilers to confirm the theoretical roofline analysis. Precise performance measurements were obtained using the NVIDIA profilers, nvvp and nsight, for the V100 and the AMD profiler rocprof for the MI100.

The roofline performance model is used to estimate the ideal performance in MFLUPS for each propagation pattern. As LBM methods are typically bandwidth bound, the roofline performance model simplifies to a function of the device global memory bandwidth $B_{BW}$ and the number of bytes per fluid lattice update $B/F$ of the lattice Boltzmann propagation pattern:

$$MFLUPS_{max} = \frac{B_{BW}}{10^6 \times B/F}. \tag{15}$$

The number of bytes per fluid lattice update $B/F$ for each method and lattice are shown in Table 2. We observe that, because the differences between the projective MR-P and recursive MR-R methods are limited to in-cache behavior, their $B/F$ requirements are identical and, for brevity, are collectively represented as MR. For example, the required memory by the ST models to simulate 15 million fluid points

| Pattern | Bytes/FLUP (B/F) | D2Q9 | D3Q19 |
|---|---|---|---|
| ST | 2Q*double | 144 | 304 |
| MR | 2M*double | 96 | 160 |

**Table 2.** Bytes per fluid lattice update for each propagation pattern and lattice.

| GPU | NVIDIA (Volta) V100 | | AMD MI100 | |
|---|---|---|---|---|
| Model | D2Q9 | D3Q19 | D2Q9 | D3Q19 |
| ST | 6,250 | 2,960 | 8,533 | 4,042 |
| MR | 9,375 | 5,625 | 12,800 | 7,680 |

**Table 3.** Estimated optimal *MFLUPS* from roofline performance model for each propagation pattern for NVIDIA V100 GPU and AMD MI100 GPU using direct addressing and equation 15.

| GPU | NVIDIA (Volta) V100 | |
|---|---|---|
| Bandwidth | 900GB/s | |
| Model | D2Q9 | D3Q19 |
| ST | 790 GB/s | 765 GB/s |
| MR | 664 GB/s | 650 GB/s |
| GPU | AMD MI100 | |
| Bandwidth | 1,228.86 GB/s | |
| Model | D2Q9 | D3Q19 |
| ST | 665 GB/s | 655 GB/s |
| MR | 614 GB/s | 664 GB/s |

**Table 4.** Estimated optimal *MFLUPS* from roofline performance model for each propagation pattern for NVIDIA V100 GPU and AMD MI100 GPU using direct addressing and equation 15.

is about 2GB for D2Q9 simulations and 4.2GB for D3Q19 simulations, against the 1.3GB and 2.23GB required by the MR models for the same kind of simulations, reducing the memory requirements in about a 35% and 47% respectively.

By combining the device bandwidths from Table 1 and the bytes per fluid lattice update from Table 2, we can use equation 15 to produce the roofline performance estimates in Table 4.

### 4.2 Performance analysis on 2D simulations

Figure 2 graphically illustrates the performance for the ST, MR-P, and MR-R GPU implementations over a range of problem sizes and in comparison with the roofline predictions.

The average performance of the ST approach using the NVIDIA V100 is about 5,300 MFLUPS, which is approximately 85% of the theoretical peak estimated from the roofline model. Using the AMD MI100, the ST approach is able to reach a performance of up to 6,200 MFLUPS. Although this is about a 15% higher performance when compared with performance on the V100, this represents 72% of the ST roofline. Conversely, for the MR-P projection pattern, we achieve significantly higher performance than for ST: about 7,000 MFLUPS and 8,600 MFLUPS on the V100 and MI100, respectively. However, the percentage of theoretical peak performance attained by the MR-P is somewhat lower, with approximately 75% for the V100 and 67% for the MI100.

Versus MR-P, the recursive MR-R propagation pattern does add computational complexity in terms of extra floating point operations and temporary variables. The arithmetic intensity of MR-R is almost 60% higher than MR-P for the NVIDIA V100. However, the impact on performance on MFLUPS for the D2Q9 lattice is not significant. Indeed, we observe that the MR-R scheme is only marginally slower than MR-P on the V100 and the two schemes have virtually identical performance on the MI100.

To better understand the performance of the ST and MR schemes with respect to the roofline, it is necessary to consider bandwidth utilization. We have measured the bandwidth reached by each of the implementations evaluated in this study. On the V100, the reference ST propagation pattern reaches about 790 GB/s, close to the 90% of the peak. The percentage of peak was considerably lower for the MR propagation patterns, achieving only 664 GB/s, for about 73% of the peak bandwidth on the V100. On the MI100, we see similar behavior: the ST method still outperforms the MR method, with a bandwidth of 665 GB/s versus 614 GB/s. However, the difference in the percentages of peak on the MI100 is somewhat wider – about 74% for ST and only 50% for MR – and both percentages are significantly worse than was achieved on the V100.

On both GPUs, we observe similar conclusions, the lower bandwidth reported by the MR approach is mainly because of higher complexity in the implementation, including a more complex memory pattern, shared memory usage, halos, and restrictions on thread block size. Although the ST implementation obtains a higher bandwidth (about 17% and 24% higher on V100 and MI100, respectively), the MR implementation is able to achieve an overall better performance in MFLUPS (about 32% and 38% better performance on V100 and MI100, respectively). This improvement is due to the lower number of memory accesses with MR, about 30% fewer for the V100 and 23% fewer for the MI100.

### 4.3 Performance analysis on 3D simulations

For the 3D performance analysis, we focus on the popular D3Q19 lattice. Figure 3 shows the performance of the ST, MR-P, and MR-R propagation patterns with respect to roofline predictions over a range of problem sizes.

Using the ST propagation pattern with D3Q19, we observe similar performance on the NVIDIA and AMD GPUs, almost 2,600 MFLUPS for the V100 and about 2,800 on the

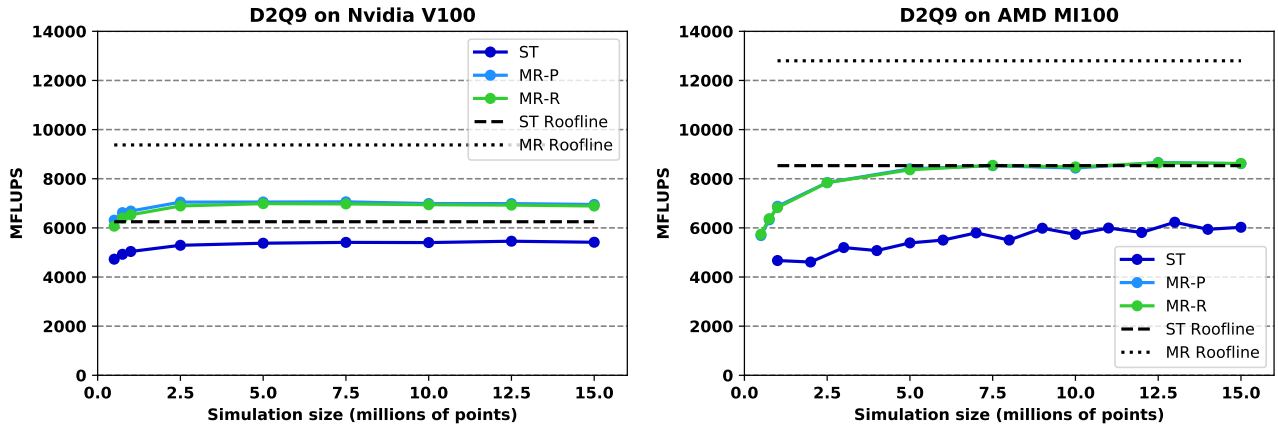


**Figure 2.** Performance analysis on D2Q9 LBM simulations for NVIDIA V100 (left) and AMD MI100 (right).

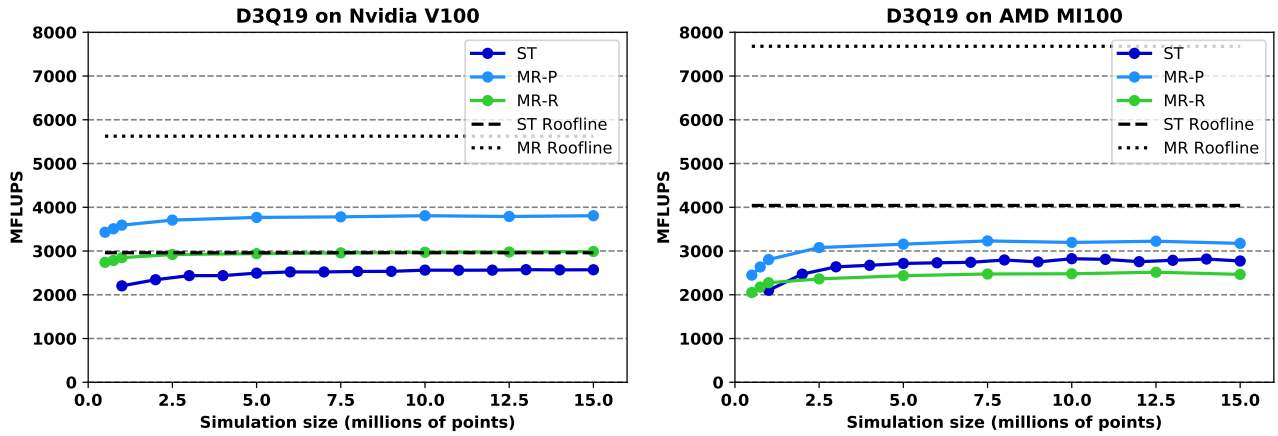


**Figure 3.** Performance analysis on D3Q19 LBM simulations for NVIDIA V100 (left) and AMD MI100 (right).

MI100. However, while this result is approximately 88% of the roofline prediction for the V100, it is only 69% of the predicted performance for the MI100. The MR-P propagation pattern is able to achieve outperform ST on both devices, recording sustained performance of 3,800 MFLUPS on the V100 and 3,200 on the MI100. However, the MR-P results are substantially lower than their rooflines, at only than 68% and 42% of expected performance on the V100 and MI100, respectively. These numbers are consistent with bandwidth, as MR-P propagation pattern reaches 650 GB/s (70% of peak bandwidth) on the V100 and about 664 GB/s (54% of peak bandwidth) on the MI100. Unlike the 2D results for all propagation patterns and the ST results for D3Q19, the V100 significantly outperforms the MI100 in terms of MFLUPS for the MR-P propagation pattern with D3Q19, despite the device's lower bandwidth.

In contrast to D2Q9, the performance differences between the MR-P and MR-R propagation patterns are quite clear with the D3Q19 lattice. Compared with the MR-P baseline, MFLUPS drop by about 800 for the V100 and 700 for the MI100.

## 5 Conclusions and future work

In this paper, we demonstrated the ability of the GPU implementation based on a moment representation propagation pattern to reduce the execution time of LBM calculations. These speedups are the result of leveraging regularization, already being used in lattice Boltzmann simulations to improve stability, to losslessly compress simulation data, and to reduce data motion to and from global memory. The projection-based moment representation MR-P achieves speedups of up to 1.32× and 1.38× for the D2Q9 lattice on the NVIDIA V100 and MI100 GPUs, respectively, as well as speedups of 1.46× and 1.14× for the D3Q19 lattice.

Overall, we found a very compelling case for using the moment representation propagation pattern on the NVIDIA V100 GPU. Results for the AMD MI100 are more mixed: while very strong performance was obtained with D2Q9, the

moment representation only modestly outperformed D3Q19 versus the distribution representation.

There are several directions for future work with the moment representation. First, further research with the moment representation should focus on lattices with a large number of components, such as the single-speed D3Q27, and multi-speed lattices such as D3Q39, because their increased runtime is often cited as a reason for not using them. Second, emerging GPU architectures feature significantly larger cache sizes, which may facilitate more efficient versions of the moment representation on these devices.

## Acknowledgments

This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory at the Oak Ridge National Laboratory, which is supported by DOE's Office of Science under Contract No. DE-AC05-00OR22725. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE's Office of Science and the National Nuclear Security Administration. This manuscript has been authored by UT-Battelle LLC under Contract No. DE-AC05-00OR22725 with the DOE. The publisher, by accepting the article for publication, acknowledges that the US Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of the manuscript or allow others to do so, for US Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

## References

[1] Gérard Dethier, Pierre-Arnoul de Marneffe, and Pierre Marchot. 2011. Lattice Boltzmann Simulation Code Optimization Based on Constant-time Circular Array Shifting. *Procedia Computer Science* 4 (2011), 1004–1013.

[2] Marco A Ferrari, Waine B de Oliveira Jr, Alan Lugarini, Admilson T Franco, and Luiz A Hegele Jr. 2023. A graphic processing unit implementation for the moment representation of the lattice Boltzmann method. *International Journal for Numerical Methods in Fluids* (2023).

[3] John Gounley, Madhurima Vardhan, Erik W. Draeger, Pedro Valero-Lara, Shirley V. Moore, and Amanda Randles. 2022. Propagation Pattern for Moment Representation of the Lattice Boltzmann Method. *IEEE Trans. Parallel Distributed Syst.* 33, 3 (2022), 642–653. https://doi.org/10.1109/TPDS.2021.3098456

[4] Gregory Herschlag, Seyong Lee, Jeffrey S Vetter, and Amanda Randles. 2021. Analysis of GPU Data Access Patterns on Complex Geometries for the D3Q19 Lattice Boltzmann Algorithm. *IEEE Transactions on Parallel and Distributed Systems* 32, 10 (2021), 2400–2414.

[5] Jonas Latt and Bastien Chopard. 2006. Lattice Boltzmann method with regularized pre-collision distribution functions. *Math. Comput. Simul.* 72, 2-6 (2006), 165–168.

[6] Jonas Latt, Bastien Chopard, Orestis Malaspinas, Michel Deville, and Andreas Michler. 2008. Straight velocity boundaries in the lattice Boltzmann method. *Phys. Rev. E* 77, 5 (2008), 056703.

[7] Jonas Latt, Christophe Coreixas, and Joël Beny. 2021. Cross-platform programming model for many-core lattice Boltzmann simulations. *PLOS One* 16, 4 (2021), e0250306.

[8] Orestis Malaspinas. 2015. Increasing stability and accuracy of the lattice Boltzmann scheme: recursivity and regularization. *arXiv preprint arXiv:1505.06900* (2015).

[9] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. 2013. Multi-GPU implementation of the lattice Boltzmann method. *Computers & Mathematics with Applications* 65, 2 (2013), 252–261.

[10] Yue-Hong Qian, Dominique d'Humières, and Pierre Lallemand. 1992. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)* 17, 6 (1992), 479.

[11] Fredrik Robertsén, Jan Westerholm, and Keijo Mattila. 2017. Designing a graphics processing unit accelerated petaflop capable lattice Boltzmann solver: Read aligned data layouts and asynchronous communication. *The International Journal of High Performance Computing Applications* 31, 3 (2017), 246–255.

[12] Adriano Tiribocchi, Andrea Montessori, Giorgio Amati, Massimo Bernaschi, Fabio Bonaccorso, Sergio Orlandini, Sauro Succi, and Marco Lauricella. 2023. Lightweight lattice Boltzmann. *The Journal of Chemical Physics* 158, 10 (2023).

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

[14] Pedro Valero-Lara. 2016. Leveraging the Performance of LBM-HPC for Large Sizes on GPUs Using Ghost Cells. In *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10048)*. Springer, 417–430. https://doi.org/10.1007/978-3-319-49583-5_31

[15] Pedro Valero-Lara. 2017. Reducing memory requirements for large size LBM simulations on GPUs. *Concurrency and Computation: Practice and Experience* 29, 24 (2017).

[16] Pedro Valero-Lara, Francisco D Igual, Manuel Prieto-Matías, Alfredo Pinelli, and Julien Favier. 2015. Accelerating fluid–solid simulations (Lattice-Boltzmann & Immersed-Boundary) on heterogeneous architectures. *Journal of Computational Science* 10 (2015), 249–261.

[17] Pedro Valero-Lara and Johan Jansson. 2015. Multi-domain Grid Refinement for Lattice-Boltzmann Simulations on Heterogeneous Platforms. In *18th IEEE International Conference on Computational Science and Engineering, CSE 2015, Porto, Portugal, October 21-23, 2015*. IEEE Computer Society, 1–8. https://doi.org/10.1109/CSE.2015.9

[18] Pedro Valero-Lara and Johan Jansson. 2015. A Non-uniform Staggered Cartesian Grid Approach for Lattice-boltzmann Method. In *Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015, 2014 (Procedia Computer Science, Vol. 51)*. Elsevier, 296–305. https://doi.org/10.1016/j.procs.2015.05.245

[19] Pedro Valero-Lara and Johan Jansson. 2017. Heterogeneous CPU+GPU approaches for mesh refinement over Lattice-Boltzmann simulations. *Concurr. Comput. Pract. Exp.* 29, 7 (2017). https://doi.org/10.1002/cpe.3919

[20] Madhurima Vardhan, John Gounley, Luiz A. Hegele, Erik W. Draeger, and Amanda Randles. 2019. Moment representation in the lattice Boltzmann method on massively parallel hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1.

[21] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E Kaufman. 2004. The Lattice-Boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics* 10, 2 (2004), 164–176.

[22] Gerhard Wellein, Thomas Zeiser, Georg Hager, and Stefan Donath. 2006. On the single processor performance of simple lattice Boltzmann kernels. *Comput. Fluids* 35, 8-9 (2006), 910–919.