Implicit Propagation of Directly Addressed Grids in Lattice Boltzmann Methods

Adrian Kummerländer[a,b,c], Márcio Dorn[d], Martin Frank[a,e], Mathias J. Krause[a,b,c]

[a]*Institute of Applied and Numerical Mathematics, Karlsruhe Institute of Technology, Karlsruhe, Germany*
[b]*Institute for Mechanical Process Engineering and Mechanics, Karlsruhe Institute of Technology, Karlsruhe, Germany*
[c]*Lattice Boltzmann Research Group, Karlsruhe Institute of Technology, Karlsruhe, Germany*
[d]*Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil*
[e]*Steinbuch Center for Computing, Karlsruhe Institute of Technology, Eggenstein-Leopoldshafen, Germany*

Abstract

Lattice Boltzmann methods (LBM) are well suited to highly parallel computational fluid dynamics (CFD) simulations due to their separability into a perfectly parallel collision step and a propagation step that only communicates within a local neighborhood. The implementation of the propagation step provides constraints for the maximum possible bandwidth-limited performance, memory layout and usage of vector instructions. This paper continues the work on implicit propagation on directly addressed grids started by A-A [1] and its SSS formulation [2] by reconsidering them as transformations of the underlying space filling curve. In this work, a new Periodic Shift (PS) pattern is proposed that imposes minimal restrictions on the implementation of collision operators and utilizes virtual memory mapping to provide consistent performance accross a range of targets. Various implementation approaches as well as time dependency and performance anisotropy are discussed. Benchmark results for SSS and PS on SIMD CPUs including Intel Xeon Phi as well as Nvidia GPUs are provided. Finally, the application of PS as the propagation pattern of the open source LBM framework OpenLB [3] is summarized.

Keywords: lattice boltzmann methods, lattice boltzmann streaming, benchmark results, SIMD, GPU, HPC

1. Introduction

Current high performance computation (HPC) setups combine different levels of parallelization and acceleration capabilities into one heterogeneous system. Simulations of transport phenomena using LBM are a major application for HPC in e.g. process engineering [4, 5].

The lattice Boltzmann (LB) algorithm is commonly separated into a local and thus perfectly parallel *collision step* and a non-local *streaming step* — both applied to cells on a regular lattice. These properties render LBM into a simulation method especially suited to extensively parallel execution. The close dependency between data layout, propagation pattern and resulting time to solution resp. performance of LBM-based simulation codes is well established in literature [6, 7]. On a per-computation-node basis, the realization of non-local streaming is an essential aspect of implementations of the LB algorithm [6]. Thus approaches to realizing this part of the algorithm are of particular interest.

While the SWAP pattern [8] yields good performance on CPUs, its inherent sequentiality and non-optimal bandwidth demands render it unsuited for perfectly parallel streaming on e.g. GPUs as well as vectorization on CPUs.

One of the main benefits of the Shift-Swap-Streaming (SSS) pattern [2] is its amenability to automatic vectorization while being perfectly parallel and convenient to implement compared to its A-A formulation [1]. However, utilizing this in the LBM framework OpenLB [3] is impeded by constraints w.r.t. implicitly reverting population locations during write-back. This led to a reconsideration of approaches to implementing the LBM streaming step with minimal demands on the collision implementation, yielding the results documented by the present work.

We aim to provide a consistent framework for formalizing implicit propagation patterns on directly addressed grids, encompassing both existing patterns and leading to a new *Periodic Shift* pattern. This framework is introduced in Section 3, following an overview of general LBM performance considerations and established propagation patterns in Section 2. The new pattern is documented by Section 4, including the discussion of implementation approaches in Section 4.1 and performance characteristics compared to SSS in Section 4.2.

Detailed bandwidth-related performance benchmarks for both SSS and PS using the established lid driven cavity case on CPU and GPU targets are provided in Section 5. An application of virtual memory PS for vectorizing the collision loop, speeding up single-core execution by a factor of up to 3.93, as well as the pattern choice in OpenLB is discussed in Section 5.1.1.

*Corresponding author
Email address: `adrian.kummerlaender@kit.edu` (Adrian Kummerländer)

2. Lattice Boltzmann Methods

Following the separation into local and non-local steps, the streaming step propagates information largely independent of the modeled physics while the specific transport phenomena is captured by the choice of collision operator and equilibrium distribution.

Definition 1 (Collision step) *The relaxation of population values f_i towards a local equilibrium distribution f_i^{eq} according to a given collision operator Ω*

$$f_i^{post}(x,t) = \Omega\left(f_i(x,t), f_i^{eq}(x,t)\right)$$

is referred to as the collision *step. This computation maps* pre- *to* post-collision *populations f_i^{post}.*

A common choice for Ω is the BGK operator with single relaxation time $\tau > 0.5$

$$\Omega = f_i(x,t) - \frac{1}{\tau}(f_i(x,t) - f_i^{\mathrm{eq}}(x,t))$$

that in combination with a formulation of the Maxwell-Boltzmann equilibrium f_i^{eq} can be shown to converge to solutions of the Navier-Stokes equations. Other choices for collision operators and equilibrium distributions are possible, providing models for a wide range of transport phenomena.

The discretization of the collision operator depends on the choice of of discrete velocities. A common set for three-dimensional Navier Stokes as a target equation is *D3Q19*.

Definition 2 (D3Q19 velocity set)

$$\{\xi_i\}_{i=0}^{18} = \left\{\xi \in \{-1,0,1\}^3 \middle| \exists j \in \{0,1,2\} : \xi_j = 0\right\}$$

Definition 3 (Streaming step) *Communication of post-collision populations to the neighboring cells corresponding to a discrete velocity set*

$$f_i(x+\xi_i, t+1) = f_i^{post}(x,t)$$

is called the streaming *or* propagation *step. Algorithms for implementing this non-local operation are referred to as* propagation patterns*.*

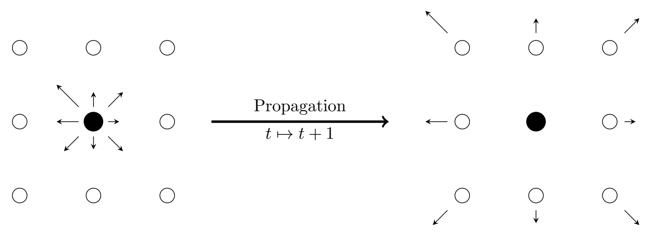


Figure 1: Propagation of post-collision populations to neighbor cells

All patterns that are discussed in this work apply equally to the streaming of post-collision populations resulting of any collision operator based on discrete velocity stencils.

Efficient LB implementations commonly employ *fused* collide and stream loops that to some degree recombine both steps again. This naturally leads to the notion of *implicit propagation* taking place while colliding.

2.1. Performance Considerations

Matching the LB algorithm's separation, the resulting throughput of cells delivered by a specific implementation of the algorithm is dependent on the realization of both the collision and the streaming steps. For the purposes of this work we consider the collision as the abstract computation transforming a set of pre-collision population values into a set of post-collision populations. The propagation step determines the surrounding framework of where these populations are stored in memory, how they are accessed by the collision step and in which way they are propagated to neighboring cells.

While this places most performance considerations on the propagation step, the realization of the collision step into actual arithmetic instructions is also of importance. Specifically, the number of arithmetic instructions resulting from writing out the same abstract collision step into actual executable code may vary greatly between implementations. While the theoretical floating point performance of modern hardware typically exceeds the number of datums that can be delivered to the arithmetic units, there are obviously still upper limits for the computations per time span and additional implementation constraints that make it desirable to minimize the number of floating point operations as far as possible. One approach to this is to utilize common subexpression elimination (CSE) at some stage of compilation. This can be a manual step [3] but lends itself very well to automatic generation from symbolic expressions in a CAS framework as is commonly applied in literature [9, 10, 11]. We follow the automatic approach for the benchmark codes used for this work. Specifically, SymPy [12] is used to generate C++ templates against a generic cell concept than can be instantiated both for GPU kernel functions and using lightweight wrappers around SIMD intrinsics on CPUs.

2.2. Propagation Patterns

Propagation patterns may be coarsely grouped into directly or indirectly addressed single- or dual-grid approaches using either a *pulling* or *pushing* scheme w.r.t the collision step.

Name	Grid		Addressing		Parallel	Access
	One	Two	Direct	Indirect		
Unfused	✓	✓	✓	✓	✓	$4Q$
A-B [1, 6]	×	✓	✓	✓	✓	$2Q$
SWAP [8]	✓	×	✓	✓	×	$3Q-1$
A-A [1]	✓	×	✓	✓	✓	$2Q$
EsoTwist [13]	✓	×	✓	✓	✓	$2Q$
SSS [2]	✓	×	✓	×	✓	$2Q$

Table 1: Overview of existing Propagation Patterns

Directly addressed patterns utilize a bijection between spatial and in-memory lattice coordinates. Contrasting this with indirectly addressed patterns, direct addressing allocates memory for all cells in the simulation domain and enables direct access to any cell while indirect addressing maintains a sparse list of cells and their neighborhood relations. In this context, *enabling direct access* means that access to any cell given its spatially embedded location is possible without querying such intermediary data structures. This property is dropped by indirect addressing in exchange for potentially significantly reduced memory usage in sparse simulation domains.

Dual grid approaches maintain two instances of the simulation lattice in memory which enables a set of straight forward A-B propagation patterns [1, 6]. Such a pattern alternates between two lattices for every timestep, reading the populations from grid A and writing the post-collision values to grid B. *Pulling* or *pushing* refers to whether the propagation is performed by pulling the values from the neighbor cells during collision or by pushing the new post-collision values to them. Notably the collide-and-stream step of an A-B pattern is trivially perfectly parallel without any write conflicts and requiring only $2Q$ memory accesses per cell. However its cache utilization is not ideal as the write and read populations take up separate cache entries.

Propagation is more involved when only a single grid is to be used. Correspondingly, most relevant publications [1, 2, 8, 13] are concerning such single-grid patterns.

The SWAP pattern [8] achieves in-place streaming by introducing sequential ordering of the collision steps in combination with eponymous swapping of the previously processed neighborhood subset. Its memory access pattern is cache-friendly and the number of memory accesses per cell at $3Q-1$ is lower than a unfused algorithm's $4Q$ operations in exchange for giving up the parallel nature of the LBM algorithm. This renders the SWAP pattern fundamentally unsuited to both the utilization of vector instructions on CPUs and GPU execution in general.

The shortcomings of the SWAP pattern w.r.t. memory bandwidth and more importantly parallelizability are addressed by the A-A pattern [1]. Instead of requiring a sequential ordering of collision operations, the accessed memory locations are alternated between odd and even timesteps by using two separate versions of each operator. This way both collision and streaming steps are perfectly parallel which is essential when targeting GPUs.

While A-A can in principle be used on both indirectly and directly addressed grids it is more suited to the latter due to the need for accessing all neighboring nodes during every second timestep. This issue is addressed by Esoteric Twist [13] which uses a non-isotropic population access pattern that distinguishes between pushing and pulling depending on the streaming direction. While not requiring it [13] also introduces the usage of a pointer-based control structure to replace the separate collision operators required by A-A. This approach is used by the Shift-Swap-Streaming [2] (SSS) pattern which reformulates A-A into a both auto-vectorization-friendly and perfectly parallel pattern for directly addressed grids.

3. Implicit Propagation

The use of regular grids to discretize the simulation domain is a central aspect of LBM performance. In addition to collision operators being uniform for all cells, both local propagation and communication between distributed domains is straight forward. Minimization of the surface between individual subdomains in a packing problem and implementation convenience suggest the use of cuboids as the basic geometry of LBM lattices. In this context a *cuboid* denotes the finite subset

$$C := \times_{i=1}^{d}\{0,\dots,c_i-1\} \subset \mathbb{Z}_{\geq 0}^{d}$$

of the positive orthant of d-dimensional Euclidean space. The vector $c \in \mathbb{Z}_{\geq 0}^{d}$ describes the *extent* i.e. the number of cells along each dimension for cuboid C. Note that this describes the lattice used for non-dimensionalized simulations with $\Delta x = 1$. Storing distinct values for each $x \in C$ in memory requires a bijection with the set

$$M := \{0,\dots,|C|-1\} \subset \mathbb{Z}$$

of one-dimensional locations. Bijections $m_c : C \to M$ are referred to as a discrete space filling curves (SFC).

Definition 4 (Location invariance of neighborhood distances) *Let $x, y \in C$ be a pair of locations in cuboid C. The one-dimensional distance w.r.t. SFC m_c is given by*

$$\delta : C \times C \to \mathbb{Z},\ (x,y) \mapsto m_c(x) - m_c(y).$$

This distance is called location invariant *iff*

$$\forall \xi \in \mathbb{Z}^d\ \forall x,y \in \{x \in C | x+\xi \in C\} : \delta(x,x+\xi) = \delta(y,y+\xi).$$

This invariance of the *in-memory* distance between all well-defined spatial neighbor locations is the essential property that can be employed for implicit propagation on directly addressed grids. Any curves that satisfy this property enable streaming of populations along their discrete velocity directions by translation of the starting point.

Definition 5 (Implicit Propagation) *Let C be a cuboid with memory bijection m_c fulfilling definition 4, $\xi \in \mathbb{Z}^d$ a discrete velocity and t the current time. The memory access function*

$$p_t : \mathbb{Z} \to \mathbb{R}$$

returns the current population values for all $x \in m_c(C)$ at time t and dummy values for $x \in \mathbb{Z} \setminus m_c(C)$. Propagation from $x \in C$ at time t to $x+\xi \in C$ at time $t+1$ is equivalent to

$$p_{t+1}(m_c(x+\xi)) = p_t(m_c(x)).$$

Due to invariance of the neighborhood distance the memory access function p_{t+1} can be defined as

$$p_{t+1} : x \mapsto p_t(\tilde{m}_c(x)) \text{ where } \tilde{m}_c : x \mapsto m_c(x) + \delta(x, x+\xi)$$

while being equivalent to propagation along ξ for all $x \in \{x \in C | x+\xi \in C\}$. Note that p_{t+1} is simply a shifted *view of the original memory function p_t. The propagation is thus performed implicitly.*

As each of the lattice's q populations is identified by a distinct discrete velocity $\xi_i \in \mathbb{Z}^d$, implicit propagation can only be performed if the population data is stored in separate memory arrays. This is commonly referred to as a *Structure of Arrays (SoA)* memory layout.

The essential difference between propagation patterns that are expressible in terms of this framework, is the specific way by which the indexing shift is performed. An overview of the established A-B and A-A patterns as well as Shift-Swap-Streaming (SSS) and the novel Periodic Shift (PS) pattern is provided in Table 2.

Using definition 4 a discrete space filling curve m_c for arbitrary cuboids C can be constructed. Starting with $d = 1$ the definition of the distance function δ can be transformed into a definition of $m(i)$ for arbitrary $i \in Z_{\geq 0}$.

$$\begin{aligned}
\delta(i,0) &= m(i) - m(0) \\
&= m(i) - m(i-1) + m(i-1) - m(0) \\
&= \delta(i,i-1) + \delta(i-1,0) \\
&= \cdots = \sum_{j=1}^{i} \delta(j,j-1) \\
\iff m_c(i) &= \sum_{j=1}^{i} \delta(j,j-1) + m(0) \\
\implies m(i) &= i\delta + m(0) \qquad\qquad \delta \equiv \delta(j,j-1)
\end{aligned}$$

Repeating this construction in the d-dimensional case for each component of $x \in C \subset \mathbb{Z}^d_{\geq 0}$ yields a family of valid space filling curves

$$\delta(x,0) = m_c(x) - m(0) = \sum_{i=1}^{d} x_i\delta_i$$

$$\iff m_{c,\delta}(x) = \sum_{i=1}^{d} x_i\delta_i + m_{c,\delta}(0).$$

Choosing $m_c(0) = 0$ and using the ordering

$$\begin{aligned}
x \geq y &\iff \delta(x,y) \geq 0 \\
x \leq y &\iff \delta(x,y) \leq 0
\end{aligned}$$

which yields

$$\forall\, i\, \exists!\, j : \delta(i,j) = \delta$$

values for δ_i can be fixed to select a δ-minimal function m_c from family $m_{c,\delta}$:

$$\delta_i = 1 \wedge \delta_{i-1} = c_i\delta_i.$$

Using $i = d$ to close the recursion without loss of generality this produces

$$\delta_i := \prod_{j=i+1}^{d} c_j.$$

The bijection resulting from these constants is the Sweep SFC [14] which is widely used to represent d-dimensional arrays is memory.

Definition 6 (Sweep space filling curve) *Let C be a cuboid of $|C| = \prod_{i=1}^{d} c_i$ cells. The* Sweep *space filling curve*

$$m_c : C \to M,\, x \mapsto \sum_{i=1}^{d} x_i \prod_{j=i+1}^{d} c_j$$

provides a bijection between d-dimensional cuboid C and 1-dimensional memory indices M. Note that the mapping between spatial dimensions and components x_i may be permuted without loss of generality.

Larger values for δ_i are possible and enable interleaving of several separate cuboid embeddings in memory when starting with $\delta_d = m > 1$. This may provide locality benefits for the coupling of multiple independent lattices.

	Approach	SFC
A-B	Read from grid A using p_{t+1}, write to grid B using p_t	any
A-A	Alternate operators using either p_t or p_{t+1}, writes performed in opposite directions	any
SSS	Shift in control structure of standard arrays, writes performed in opposite directions	Sweep
PS	Rotate cyclic arrays	Sweep

Table 2: Overview of different implicit propagation patterns

This discrete SFC provides the foundation for implicit propagation on any directly addressed grids. While Shift-Swap-Streaming and Periodic Shift explicitly depend on Sweep, the connection is more subtle for the two-grid A-B and one-grid A-A patterns. Both of these can in theory use other SFCs that don't fulfill Definition 4 for their memory bijection such as e.g. a Hilbert curve. However the non-trivial and location-dependent neighborhood distances in such a curve would increase the cost of propagation and render them closer to indirectly addressed patterns.

The A-B pattern can be formulated in terms of implicit propagation by using the Sweep SFC for both grids s.t. the memory bijection p_{t+1} is used for reads and p_t is used for writes. This is equivalent to a pull-style A-B pattern.

Correspondingly, the A-A pattern can be formulated by using either p_t or p_{t+1} for the alternating operators performing reverted writes.

If a control-structure is used to encode p, we get the SSS pattern. In this case, as $\tilde{m}_c(C) \not\subseteq m_c(C)$ for $\xi \neq 0$ one has to ensure that the underlying memory buffer is valid for indices outside of $m_c(C)$. Some values for locations not in C° will thus be located in a so called *padding* area. It can be observed that this padding area grows with every time step. SSS addresses this by writing the post-collision values in reversed order. This way the direction of the shift reverses every timestep and the padding area is bounded.

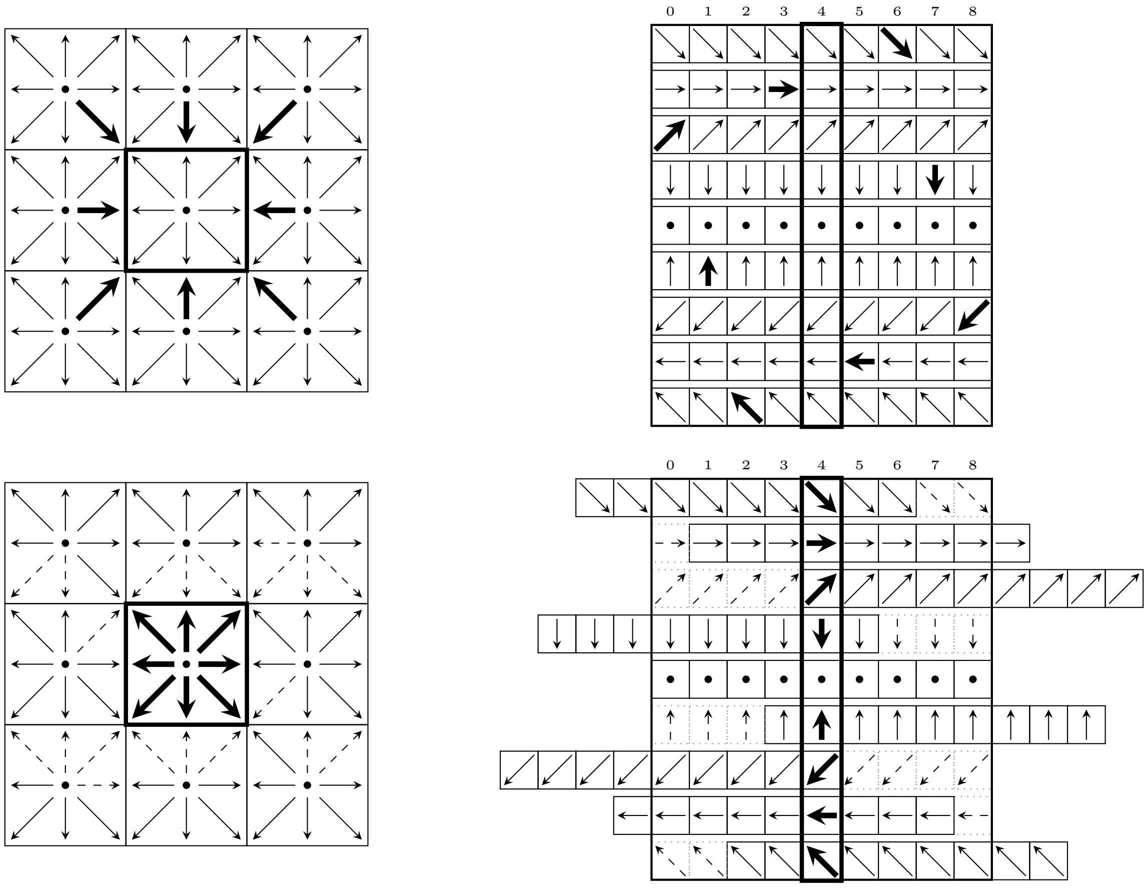


Figure 2: Propagation without data transfer by population array rotation in PS

4. Periodic Shift Pattern

The reversed post-collision storage approach taken by SSS is one possibility for bounding the indexing shift. Another option is to wrap the memory bijection m_c so that $\tilde{m}_c(C) = m_c(C)$. This is realized by implementing the q population buffers as cyclic or *rotatable* arrays with shiftable start positions.

The *Periodic Shift* propagation pattern resulting of this approach places minimal demands on the implementation of collision operations. Post-collision populations can be written back to their original locations without reverting and the population buffers do not need to take into account any padding areas given suitable cyclic array realizations. While the former aspect can be an advantage when adapting an existing LBM code, this approach also results in a different memory access pattern that in turn leads to different performance characteristics.

Propagation by shifting the starting point of cyclic arrays is fully transparent and *defuses* the collide-and-stream algorithm into its canonical local collide and non-local stream step while preserving the bandwidth advantage. These features are traded for by the comparably larger difficulty of implementing high-performance cyclic arrays as will be expanded on in Section 4.1.

```
SHIFT($D, \xi$)
1  return $\xi_1 + \sum_{i=2}^{d} \xi_i \prod_{j=1}^{i-1} D.extent[j]$

PERIODICSHIFT($D, f_{\text{old}}$)
1  // Rotate along distance given by discrete velocities
2  for $i \in \{1, \ldots, D.q\}$
3      $f_{\text{new}}[i] = \text{ROTATE}\,(f_{\text{old}}[i], \text{SHIFT}(D.c[i]))$
4  return $f_{\text{new}}$
```

Listing 1: Formulation of PS in terms of a Rotate function

Figure 2 illustrates how propagation is performed using only rotation of cyclic arrays. Outgoing populations take on the locations of undefined incoming populations after propagation. The lattice populations are well-defined as a whole only after collision and prior to streaming. Only the interior is well-defined after streaming. This is not a problem as these values need to be reconstructed by boundary conditions anyway independently of the specific pattern. It should be noted that the incoming populations at the outer boundary are *not* equivalent to a periodic boundary condition for the underlying sweep space filling curve.

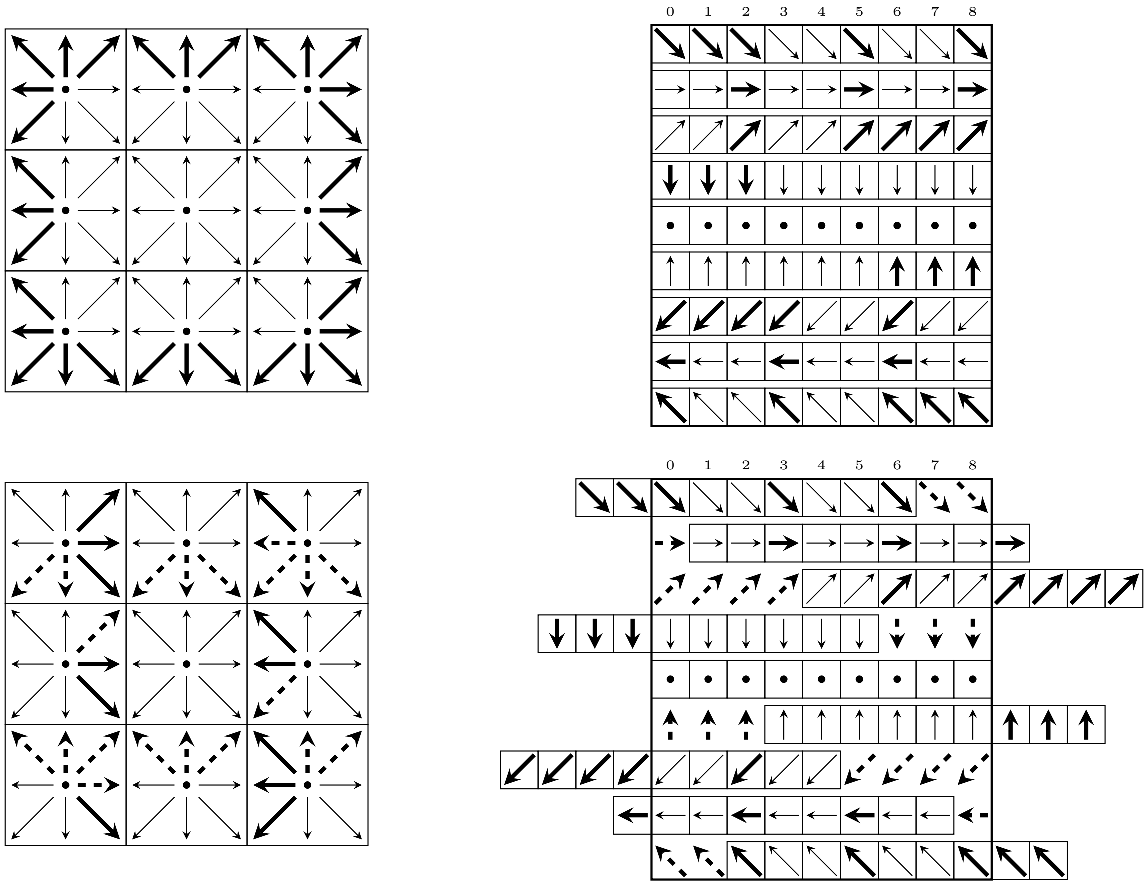


Figure 3: Post-propagation locations of outgoing populations in PS

An example of where the outgoing populations are propagated to is provided in Figure 3, all incoming populations are undefined and need to be reconstructed by e.g. boundary conditions or overlap communication with neighboring blocks in any case.

4.1. Implementation

In order to realize PS as an implicit propagation step, the ROTATE function must be expressed as some kind of pointer transformation. Explicit propagation would result of actually rotating the population arrays but this would defeat the goal of providing *zero memory transfer cost* streaming. This section explores various approaches to ROTATE, an overview is provided in Table 3.

Cyclic array	Padding-less grids	SIMD[*]
Modulo	Any	×
Branching	Any	×
Base-2 bit modulo	$\exists x \in \mathbb{N} : volume = 2^x$	×
Virtual address	$volume \cdot \text{SIZEOF}(\text{FPT}) \mod pagesize = 0$	✓

[*] w.r.t. straight forward SIMD on consecutive data. More complex approaches (e.g. using `gather` intrinsics) are possible in all cases.

Table 3: Overview of approaches to implementing PS

Reflecting the implicit nature, ROTATE is provided as a function GETPOPULATION :

$$SHIFT \times \{0, \ldots, volume - 1\} \to \{0, \ldots, volume - 1\}$$

that maps a given shift and fixed cell index set to physical memory indices. The essential implementation question is how to do this as efficiently as possible.

4.1.1. Rotation via Explicit Index Computation

The simplest way of implementing an implicit rotation is to use the cell indices modulo the shift remainder for translating between cell index and memory location. This memory location

$$f_{\text{start}} + (shift + i) \mod n_{\text{cells}}$$

is then evaluated w.r.t. a shifted pointer f_{start}. A downside of this approach is that consecutive cell indices do not map to consecutive memory locations in the general case. This prevents the application of most SIMD instructions and forces implementers to either invest more effort than would be necessary in e.g. the SSS pattern or to refrain from vectorizing the collision step altogether.

In the same vein, modulo operations can be replaced by branching between two start pointers depending on whether the requested cell index crosses the rotation fold.

This tends to slightly reduce the access overhead on CPUs compared to the direct usage of modulo instructions. Other variants of this branch-based approach might also be of interest depending on the target platform. These include e.g. replacing the branch by *comparison-indexed* accesses or maintaining just one start pointer per array and applying the offset using a remainder-dependent mask.

```
GetPopulation($f_{\text{base}}, n_{\text{cells}}, shift, i$)
 // Pre-computing start pointer options
 if $shift \geq 0$
     $remainder = n_{\text{cells}} - shift - 1$
     $f^0_{\text{start}} = f_{\text{base}} + shift$
     $f^1_{\text{start}} = f_{\text{base}} - (n_{\text{cells}} - shift)$
 else
     $remainder = -shift - 1$
     $f^0_{\text{start}} = f_{\text{base}} + (n_{\text{cells}} + shift)$
     $f^1_{\text{start}} = f_{\text{base}} + shift$

 // Single branch and addition to resolve access
 if $i > remainder$
     return $f^1_{\text{start}} + i$
 else
     return $f^0_{\text{start}} + i$
```

Listing 2: GetPopulation implemented using branching

For completeness, another way of reducing the cost of the modulo operation is to restrict the supported lattice volumes to powers of two. In this case modulo can be replaced by a cheap bit level AND:

$$\text{base}_2(x \bmod y) \equiv \text{base}_2(x) \text{ AND } \text{base}_2(y-1) \text{ for } y = 2^z.$$

The obvious downside is that only a very small subset of padding-less cuboid sizes is supported.

4.1.2. Rotation via Virtual Memory Address Translation

An efficient way of implementing cyclic arrays is by modifying the page table to translate two adjacent virtual address buffers to a single shared physical address buffer. This offers an approach to direct applicability of all SIMD instructions and eliminating the access complexity inherent in explicit index computations. Rotation wrapping is resolved in hardware at address translation time without any additional runtime cost as these translations must be performed in any case. However, memory management at this level is highly OS specific.

The in-memory size of cyclic arrays implemented in this fashion needs to be a multiple of the system-specific page size. Note that the set of valid grid dimensions is a *reasonably dense* subset of all possible grid dimensions. Adapting existing block decomposition schemes to fit cuboids to the closest valid extend is straight forward. Alternatively the size of the population arrays may also be simply padded to the next multiple of the page size to transparently handle any lattice sizes.

Using the virtual memory approach, propagation is handled only by the memory setup and population pointer shift. All collision operators and boundary conditions are provided a contiguous view of the population arrays.

```
ConstructPopulationBuffer($f_{\text{base}}, n_{\text{cells}}$)
 // shm_open in Unix, cuMemCreate in CUDA
 $f_{\text{physical}}$ = AllocatePhysicalBuffer($n_{\text{cells}}$)
 // mmap in Unix, cuMemAddressReserve in CUDA
 $f_{\text{base}}$ = AllocateVirtualBuffer($2 * n_{\text{cells}}$)
 // mmap in Unix, cuMemMap in CUDA
 MapVirtualBuffer($f_{\text{base}}$            , $n_{\text{cells}}, f_{\text{physical}}$)
 MapVirtualBuffer($f_{\text{base}} + n_{\text{cells}}, n_{\text{cells}}, f_{\text{physical}}$)

Rotate($f_{\text{base}}, f_{\text{start}}, n_{\text{cells}}, shift$)
 $f_{\text{start}} = f_{\text{start}} + shift$
 // Ensure consecutive access to all cell locations
 if $f_{\text{start}} < f_{\text{base}}$
     $f_{\text{start}} = f_{\text{start}} + n_{\text{cells}}$
 else if $f_{\text{start}} + n_{\text{cells}} > f_{\text{base}} + 2 * n_{\text{cells}}$
     $f_{\text{start}} = f_{\text{start}} - n_{\text{cells}}$

GetPopulation($f_{\text{start}}, i$)
 // $f_{\text{start}}$ is rotated $f_{\text{base}}$ for all $i \in [0, n_{\text{cells}})$
 return $f_{\text{start}} + i$
```

Listing 3: Setup and access of virtual memory PS

When implementing this virtual memory mapping in Unix environments it is important to use shared memory objects allocated via `shm_open` as the physical buffer instead of temporary files — the latter can seem workable but leads to unwanted disk flushing. Directly accessing these shared areas for inter-process propagation on shared memory systems may provide an additional possibility for optimization compared to serialization into MPI messages.

Nvidia GPUs offer low-level access to virtual memory starting CUDA 10.2 [15]. Thus all described approaches to implementing the PS pattern can also be applied there. It should be noted that the GPU page size is commonly larger than on CPUs which further reduces the set of padding-less lattice dimensions. GPU LBM codes commonly spawning one thread per cell [9, 10] in combination with bandwidth-limited collision leads to the expectation that branching approaches can also be implemented efficiently.

4.2. Performance Characteristics

The specific implementation of the memory bijection underlying implicit propagation as well as the SFC are an essential determinant of the performance characteristics. In this context *isotropy* describes the relationship between performance and spatial dimension ratio while *time dependency* considers performance differences between time steps.

4.2.1. Time dependency

By principle, the memory access functions between two consecutive LB algorithm steps can not be the same when an implicit pattern is used for streaming on a single grid. This means that the performance of implicit propagation patterns is necessarily dependent on the specific time step as different access functions lead to different memory access patterns (e.g. different alignment) that influence the data access speed.

Such a dependency can be observed for an indirectly addressed A-A pattern [16, Fig. 7] where each even step achieves up to twice the performance of the odd step. As the memory access pattern is the same between A-A and SSS this is also consistently observable in the latter. However due to direct addressing it is much less pronounced, alternating between approximately 0.99 and 1.01 of the mean performance as depicted in Figure 4.

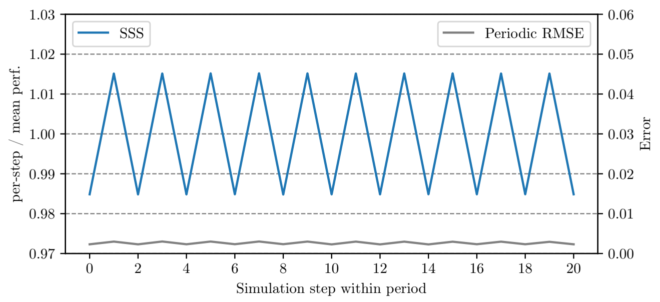


Figure 4: Per-step performance for a 128^3 lattice using SSS on ZEN

More complex per-step performance characteristics are observed for the Periodic Shift pattern. Figure 5 summarizes averaged per-step performance results relative to the achieved mean performance. Compared to the alternating states in SSS, the PS results follow a varied distribution with outliers down to only 75% of the mean performance.

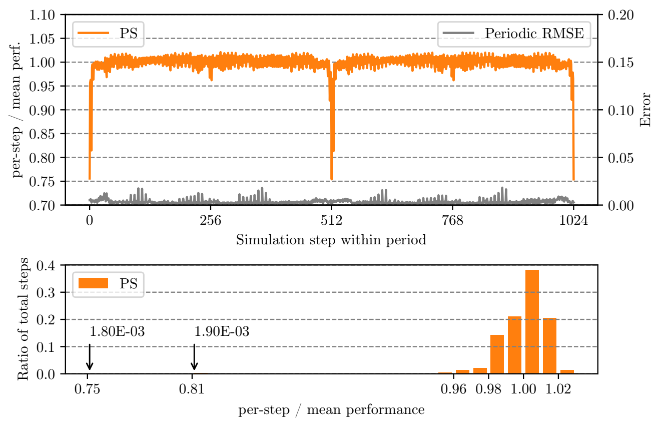


Figure 5: Per-step performance for a 128^3 lattice using PS on ZEN

Consistently repeating 512 step periods can be observed independently of the cuboid size on any double-precision lattice. The distinct performance regressions visible in the histogram are localized every 512 steps exactly. Not by coincidence $512 \times \text{SIZEOF}(\textit{double}) = 4096$ is the page size of the CPU used for these measurements. Due to this, all shifts in double-precision arrays starting at page-aligned addresses will again be aligned every 512 steps irrespectively of the individual shift distances per step. Aligned accesses lead to increased frequency of conflict misses in the cache hierarchy which explains the observed recurring performance degradation.

Most current CPUs utilize a hierarchy of set-associative caches [17] storing entries in lines of some fixed size. Alignment of multiple accesses to the page boundary reduces the number of cache sets available to satisfy these accesses. This in turn increases the number of cache lines that are prematurely evicted, increasing the number of cache misses which decreases performance. Conflict misses may also occur in the translation lookaside buffer (TLB) that is used to cache the results of virtual memory address translation.

While the performance regression caused by such conflict misses doesn't necessarily impact the observed mean performance, a straight forward workaround is to pre-shift the starting positions of all population arrays by some suitable distance. Choosing such a distance depends on knowledge of the specific cache setup used by the targeted processor.

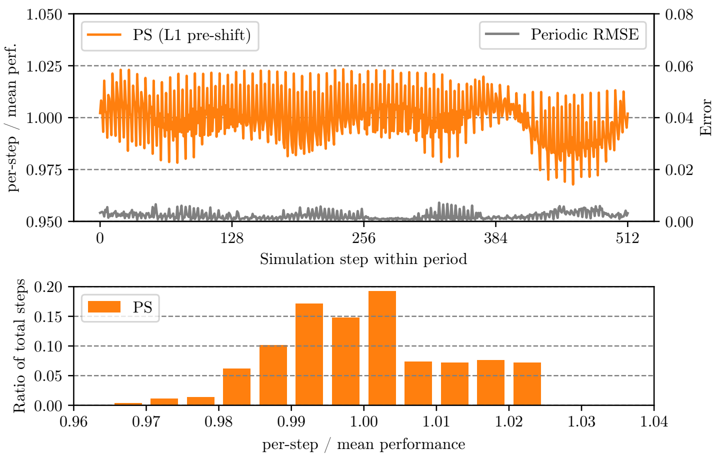


Figure 6: Per-step performance for a 128^3 lattice using PS on ZEN with L1-aware pre-shift

On all tested CPUs pre-shifting the population arrays to by one cache line each such that address bits 6 to 12 are unique eliminated the performance minima. In order to investigate potential issues with TLB conflicts, an older Intel Haswell CPU providing only 4-way TLB associativity was also tested. There, smaller periodic regressions are still observable for L1-only pre-shifting. Applying additional TLB pre-shifting in address bits 12 to 18 removes these artifacts. In comparison no significant differences between pre-shifting only L1 or both L1 and TLB are observed for ZEN. This matches the full associativity of ZEN's TLB setup.

4.2.2. Isotropy

In order to investigate the isotropy of both implicit propagation patterns, additional properties of the chosen space filling curve m_c need to be introduced.

Definition 7 (Preferred dimension) *Dimension i is called the* preferred dimension *of a given SFC m_c if any well-defined spatial neighbors along x_i are also neighbors with respect to the memory index. i.e.*

$$\forall x \in C^\circ\, \forall y \in \{x | x_i \pm 1\} : |\delta(x, y)| = 1 \iff i \textit{ is preferred.}$$

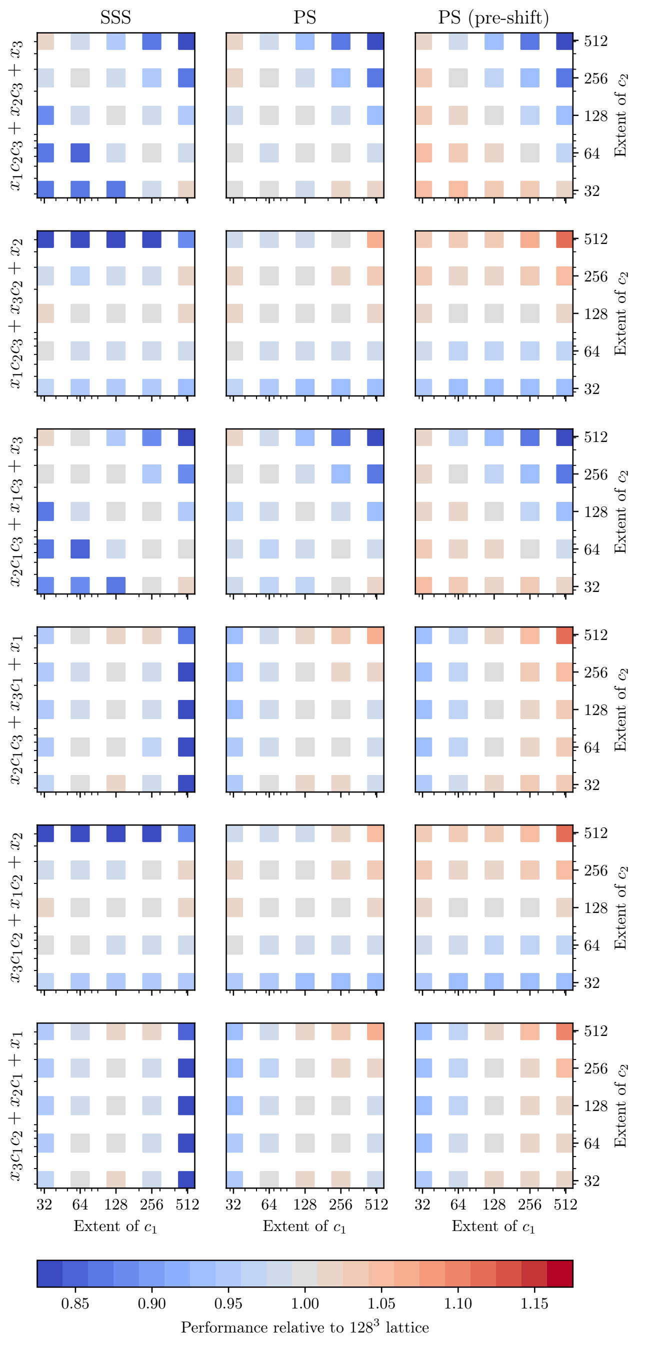
Definition 8 (Invariant dimension) *Dimension i s.t. $m_c \equiv m_{\tilde{c}}$ for $c_i \neq \tilde{c}_i$ is called an invariant dimension of m_c. For the sweep SFCs that are considered there is exactly one such dimension.*

The preferred dimension of the Sweep SFC as given in Definition 6 is x_d while the invariant dimension is x_1. Ignoring interleaved parameterization there are six different versions of the Sweep SFC usable on a directly addressed 3D lattice. Figure 7 plots the performance of various configurations of 128^3 cells for the SSS and PS pattern relative to the performance for the equilateral reference $C = \times_{i=1}^{3}\{0, \dots, 127\}$. The measurements were obtained on over 100 non-parallelized steps on a minimal simulation domain using a single mask to select a BGK collision step in the interior and bounce back at the frontier.

In the general case any spatial neighbors along a non-preferred dimension are not neighbors in memory. This lack in isotropy can impact performance in two ways: By changing the locality properties that are important for non-contiguous access patterns and by modifying access alignment per propagation step. This closely relates to the time dependency analysis in the previous section.

By definition the extent $\delta(x, y)$ of all individual array shifts is determined by m_c. Specifically, the shifts are a function of cuboid size components c_i along all dimensions i that m_c is not invariant in. The preferred non-invariant dimension was found to determine the overall anisotropic *shape* of the relative performance for both SSS and PS. This can be seen clearly in Figure 7. While the equilateral reference results in approximately equal performance (see Section 5.1), the specific anisotropy varies along patterns and preferred dimensions by $\pm 15\%$. Notably, SSS trends to mostly regressed performance while PS provides both negative and positive peaks at larger extents along the preferred dimension. This suggest that pre-shifting could also be applied to tune SSS performance and that the SFC choice should be modified for non-equilateral lattices w.r.t. the ratio of extents and pattern.

From a cache utilization perspective, the ideal case is for accesses to happen in sweeps over all cells following their in-memory sequence. However, using this approach for changes to small lattice subsets would constitute a large overhead of unnecessary cache loads. In this case one could use a ordered list of the specific cell indices to be modified. In this case, the problem with anisotropic SFCs becomes apparent e.g. when processing the boundary conditions for a lid driven cavity or more generally when copying overlap populations to communication buffers. In this case the preferred dimension controls what fraction of accesses is contiguous and which are isolated. Correspondingly, we observed the usage of multiple masks in a single sweep of the entire lattice to be more efficient than list-based approaches.



Single-threaded relative performance of simulations on lattices of extent $c = (c_1, c_2, 128^3/(c_1c_2))$ for SSS and (pre-shifted) PS on ZEN. Qualitatively similar results were also observed for other cell counts.

Figure 7: Configurations of 128^3 cells for different SFCs and patterns

Name		ZEN	SKL	KNL
Manufacturer		AMD	Intel	Intel
Processor		Threadripper 2990WX	Xeon Platinum 8151[0]	Xeon Phi 7250
Architecture		ZEN+	Skylake SP	Knights Landing
Cores		32	12	68
SIMD ISA		AVX2	AVX2, AVX-512	AVX2, AVX-512
L1	[KiB]	32×96	12×64	68×64
L2	[MiB]	32×0.5	12×1	34×1
L3	[MiB]	8×8	24.75	16 GiB[1]
Compiler		GCC 10.2[2]	GCC 9.3.0[2]	ICC 19.1.0.166[3]
Accessed via		Local Workstation	Amazon EC2	DUG McCloud
Memory bandwidth				
`update`[4]	[GB/s]	60.2	101.4	365.1
`update_19`	[GB/s]	58.1	99.7	316.9
LBM bandwidth[5]				
`update`	[MLUPs]	198	334	1201
`update_19`	[MLUPs]	191	328	1042

[0] Custom Xeon Skylake SP of an Amazon EC2 `z1d.metal` instance
[1] Xeon Phi offers 16 GiB of high-bandwidth on-die memory usable in a L3-like fashion
[2] Using flags `"-O3 -march=native -mtune=native -mavx2 -mavx512f -mavx512dq"`
[3] Using flags `"-O3 -w -ipo -axMIC-AVX512,CORE-AVX2"`
[4] AVX2 / AVX-512 vector update microbenchmark provided by `likwid_bench`
[5] Number of MLUPs when using double precision FPT i.e. bandwidth/$(2 * 19 * 8 * 1e6)$

Table 4: Specifications of the CPUs used for the pattern benchmarks

5. Benchmark Results

We compare Periodic Shift (PS) to both Shift-Swap-Streaming (SSS) [2] and SWAP [8]. All benchmarks were performed using the established lid driven cavity (LDC) [18, 19]. This is used frequently as a performance benchmark and validation case [3, 20, 21]. It is also specifically suited to comparisons of propagation patterns due to the minimal usage of boundary conditions. All tests were performed on a D3Q19 lattice using either single or double precision floating point values.

The bulk fluid is simulated using a plain BGK collision while the walls are modelled using the fullway bounce-back boundary condition with post-collision values

$$f_\xi^{\text{post}} := f_{-\xi}^{\text{pre}}.$$

The tangentially moving lid is also modelled using fullway bounce-back extended by a moving wall correction [22]

$$f_i^{\text{post}} := f_j^{\text{pre}} - 2\rho_0 w_j \frac{\xi_j \cdot v_w}{c_s^2} \text{ where } \xi_j = -\xi_i$$

for given wall velocity $v_w \in \mathbb{R}^3$ and $\rho_0 = 1$. The individual cells of a given simulation cuboid C are assigned these collision steps as

$$\begin{aligned} C_{\text{bulk}} &:= \times_{i=0}^{2} \{1, \dots, c_i - 2\} \\ C_{\text{lid}} &:= \{x \in C | x_2 = c_2 - 1\} \\ C_{\text{wall}} &:= C \setminus (C_{\text{bulk}} \cup C_{\text{lid}}). \end{aligned}$$

We primarily focused on comparing PS and SSS for equilateral lid driven cavities w.r.t. to their bandwidth saturation using custom benchmark codes[1]. Further tests were performed comparing the performance of PS, SWAP and explicitly reverted SSS in the context of OpenLB [3].

CPU reference memory bandwidth was measured with `likwid-bench` [23] which offers a large set of specialized microbenchmarks. Conversions between the number of millions of lattice updates per second (MLUPs) and memory bandwidth values were confirmed by `likwid-perfctr`.

In addition a `update_19` microbenchmark following [24] was used to determine performance limits with respect to the LBM specific access pattern. This benchmark uses the same SoA layout of a D3Q19 lattice as is used for the simulations but doesn't perform any streaming and only performs a triad-like computation for each cell.

5.1. CPU Performance

While CPU-based LBM performance is commonly not competitive compared even to desktop-grade GPUs, they are still a central component of any HPC system and an important execution target of many LBM codes [3, 25, 26].

Table 4 provides an overview of the different CPUs that were used for the benchmarks. In order to cover a representative selection of hardware setups, each CPU targets a different use case: The ZEN system belongs to the class of higher-end desktop CPUs, SKL is a member of Intel's widespread HPC processor series and KNL is a non-conventional bandwidth-focused CPU. Notably both Intel CPUs support SIMD-widths up to 512 bits while ZEN is limited to 256 bit vectors.

[1]SweepLB `https://code.kummerlaender.eu/SweepLB` for CPU and LiterateLB `https://literatelb.org` for GPU benchmarks

Description		ZEN		SKL		KNL	
		SSS	PS	SSS	PS	SSS	PS
Min	[MLUPs]	512	534	655	757	130	140
Max	[MLUPs]	832	831	912	945	971	1017
Avg	[MLUPs]	720	720	813	871	798	852
Bandwidth saturation w.r.t. `update_avx(512)`							
Min		0.09	0.09	0.13	0.15	0.04	0.04
Max		0.24	0.22	0.55	0.58	0.82	0.87
Avg		0.12	0.12	0.30	0.32	0.65	0.70
Bandwidth saturation w.r.t. `update_19`							
Min		0.32	0.33	0.35	0.38	0.66	0.73
Max		0.50	0.51	0.61	0.64	1.00	1.00
Avg		0.37	0.37	0.44	0.48	0.88	0.94

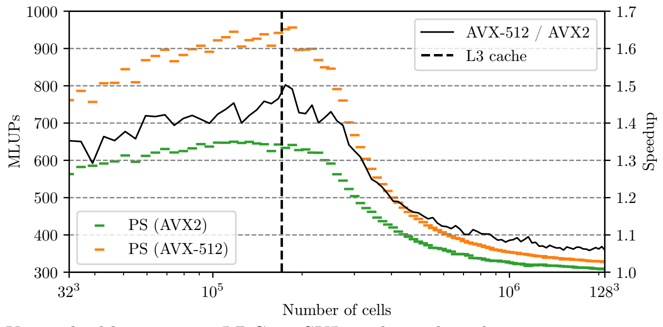
w.r.t. lattice sizes between 32^3 and `#bytes`$(L3)/(19*8)$. On KNL up to the maximum sampled size of 264^3 due to using HBM in transparent cache mode.

Table 5: Overview of cache-fitting double-precision CPU results

Highlighting the level 3 cache sizes as one important difference between the test systems, Table 5 summarizes both the total and bandwidth-related performance for LDC sizes between 32^3 and the maximum tested cache fitting value. Interestingly, SKL yields the best total mean performance while having both the lowest number of cores and the smallest cache size. While this must be qualified by the correspondingly smaller selection of samples, SKL also saturates the cache bandwidth better than ZEN and provides significantly higher worst-case results than KNL.

While ZEN provides good bandwidth saturation for larger problems, the comparably low value thereof restricts its competitiveness to smaller problems. This should not be taken as a general issue but rather as an common downside of desktop-grade CPUs. ZEN's total performance on cache-fitting problems is close to both other test CPUs and performance on the server-grade AMD EPYC version of ZEN can be expected to be significantly better.

The cache bandwidth utilization of only ~ 0.12 on ZEN is mitigated to some degree by its large L3 cache of 64 MiB leading it to provide the best cell throughput for problem sizes in the vicinity of the performance intersection of SKL and KNL. The likely explanation for the comparably lower utilization is the more developed SIMD support on Intel.



Using double-precision LDC on SKL with 12 threads.

Figure 8: Comparison of AVX2 and AVX-512 using PS on SKL

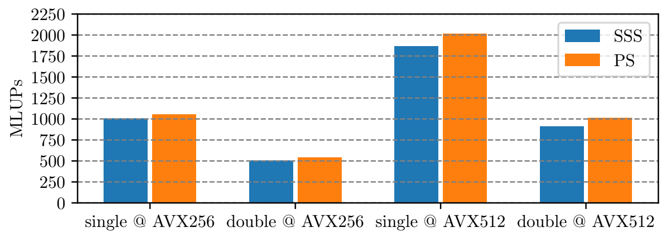
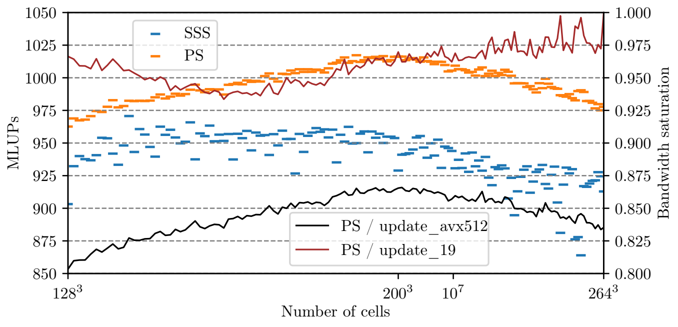


Figure 9: Performance on KNL for a 256^3 cavity using 136 threads

The choice between AVX2 and AVX-512 improves the performance for small cache-fitting lattice sizes on SKL by a factor of ~ 1.5 as depicted for PS in Figure 15. However no two fold speedup as can be achieved on KNL is observed there and the effect diminishes for larger lattice sizes. This is only to some degree observed on KNL due to its 16 GiB of on-die high-bandwidth memory acting as a L3 analogue. This suggests that AVX-512 provides primarily a computational and not a memory access advantage. Furthermore, the benchmarked LB implementation is only bandwidth-bound for problem sizes beyond the cache capacity.



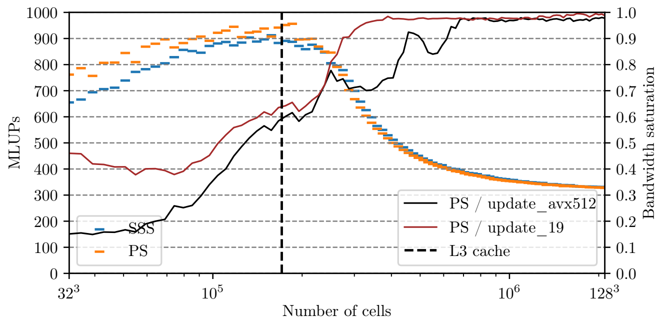
Using double-precision LDC on KNL with 136 threads and AVX-512.

Figure 10: Performance for SSS and PS on KNL

Description		ZEN		SKL		KNL	
		SSS	PS	SSS	PS	SSS	PS
Min	[MLUPs]	164	166	307	308	864	963
Max	[MLUPs]	168	169	329	327	971	1017
Avg	[MLUPs]	166	167	314	314	940	997
Bandwidth saturation w.r.t. `update_avx(512)`							
Min		0.89	0.90	0.97	0.97	0.74	0.80
Max		0.91	0.93	0.99	0.99	0.82	0.87
Avg		0.90	0.91	0.98	0.98	0.80	0.85

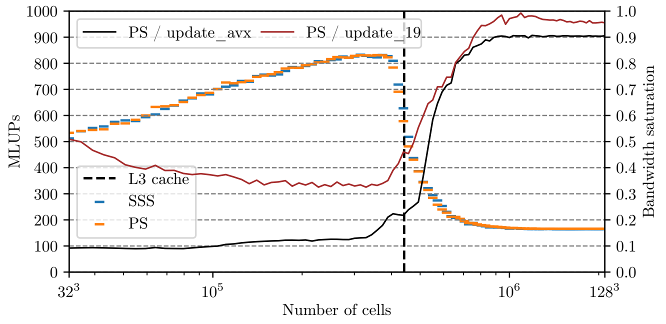
Table 6: Double-precision CPU results for problems larger than 128^3

While providing the best mean cache saturation w.r.t. both reference measurements, KNL underperforms for sizes fitting in the L3 cache on SKL and ZEN. For these sizes, KNL yields the lowest measured total cell throughput and bandwidth utilization relative to `update_avx512`. This relationship inverts for the saturation relative to `update_19` suggesting that independent of the propagation pattern the data layout and parallelization scheme is not an ideal fit to KNL and needs to be adapted to fully utilize the hardware. This is supported by the observation that different from ZEN and SKL, reference bandwidths obtained using the ISA-specific `update_avx` microbenchmarks on KNL are significantly higher than the ones obtained using the lattice-like access pattern in `update_19`. The small per-core problem size due to the high number of threads is also a possible factor for KNL's underperformance alongside the generally lower peak bandwidth.



Using double-precision LDC on SKL with 12 threads and AVX-512.

Figure 11: Performance for SSS and PS on SKL



Using double-precision LDC on ZEN with 32 threads and AVX2.

Figure 12: Performance for SSS and PS on ZEN

Focusing now on a wider spectrum of problem sizes, Tables 6 and 7 provide an overview of the results for the subset of sample sizes larger than 128^3 resp. all samples. There, benefits of KNL's high bandwidth are observable and result in the highest measured number of MLUPs and a three fold improvement of the mean performance for problems beyond cache effects on SKL and ZEN.

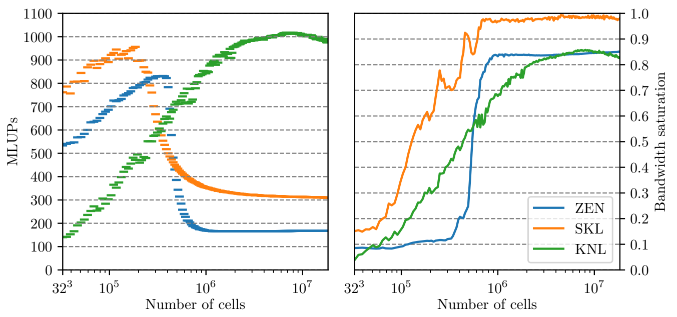
Description		ZEN		SKL		KNL	
		SSS	PS	SSS	PS	SSS	PS
Min	[MLUPs]	164	166	307	302	130	140
Max	[MLUPs]	833	831	912	956	971	1017
Avg	[MLUPs]	258	259	394	398	798	852
Bandwidth saturation w.r.t. `update_avx(512)`							
Min		0.09	0.09	0.13	0.15	0.04	0.04
Max		0.91	0.93	0.99	0.99	0.82	0.87
Avg		0.77	0.77	0.90	0.90	0.65	0.70

Table 7: Overview of all double-precision CPU results

Despite the different memory access space covered by both tested patterns, the achieved total mean performance is approximately the same on both ZEN and SKL and reasonably close on KNL. The average performance for cache-fitting problems on SKL is larger for PS by around 58 MLUPs but this does not extend to larger problems and even inverts slightly, yielding nearly equal mean values when considering all tested problem sizes.

On ZEN, the SSS pattern can be observed to provide better performance at the cache-size boundary preceded by approximately equal results and followed by a slight but consistently reproduced advantage of around 2 MLUPs for PS. It should be repeated at this point that access alignment was observed to be an important factor on ZEN as an initial version of the benchmark code produced only half the performance when not explicitly aligning the SSS population buffer. This is implicitly guaranteed for PS when using the virtual memory approach.

Different from SKL and ZEN results, a consistent advantage for PS across most sample sizes of ~ 50 MLUPs was observed on KNL. This translates into a 5% increase of the mean bandwidth saturation and the only observed CPU result exceeding 1000 MLUPs.



Using vectorized double-precision LDC parallelized via OpenMP. Bandwidth-relative performance w.r.t. Likwid `update_avx(512)`. ZEN: AVX2 @ 32 threads, SKL: AVX-512 @ 12 threads, KNL: AVX-512 @ 136 threads.

Figure 13: CPU performance using PS for ZEN, SKL and KNL

Concluding this section, our performance evaluation doesn't provide a foundation for prefering one pattern over the other in practical applications. Both SSS and PS provide good and mostly bandwidth-limited performance across all test CPUs. Further discussion of the pattern choice will be provided in Section 5.1.1. The similar results despite different access patterns can be argued as support for the concept of utilizing implicit propagation based on the SFC neighborhood characteristics. On all tested CPUs the virtual memory system is *performance transparent* to the degree that wrapping accesses into a cyclic array in PS provides performance parity to the reverted stores in SSS. As the per-collision view of the lattice w.r.t. to SIMD instructions is identical between both patterns, arguments for considering SSS as auto-vectorization friendly [2] apply equally to PS.

5.1.1. Application to OpenLB

The open source LBM framework OpenLB [3] used the SWAP pattern [8] up until version 1.3r1 [27]. Starting with version 1.4 [28] it relies on Periodic Shift with a branching control structure. This section documents this important step in OpenLB's development and explores an approach to enabling vectorization of the collision loop. This is in turn led to significantly improved CPU-based performance in addition to providing groundwork for supporting GPU targets in OpenLB's parallelization concept.

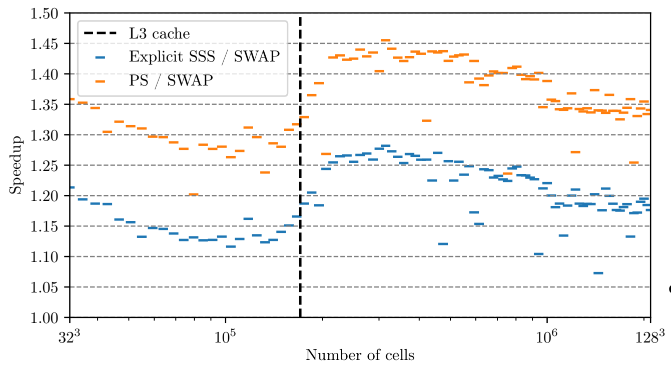


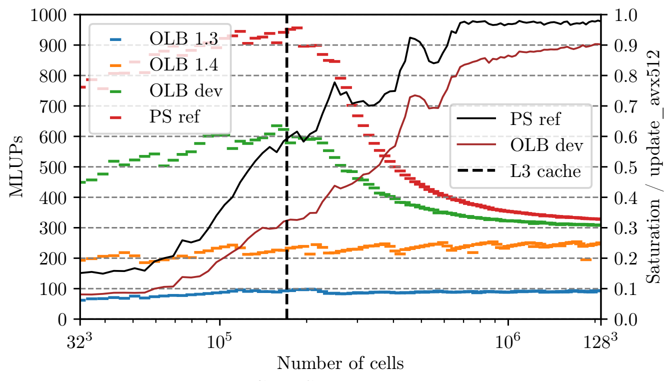
Figure 14: Single thread speedup relative to SWAP on SKL

As the branching PS pattern in OpenLB 1.4 is based on a comprehensive revamp of the framework's core data structures, it is straight forward to modify this latest release to use an adapted SSS pattern with explicit revert. Implementing the canonical version [2] — where the revert is performed implicitly while committing the post-collision values — poses a challenge due to how collisions are modeled in OpenLB. Specifically there is no straight forward way of distinguishing between cell writes and reads which prevents the implicit revert during write back. This obstacle motivated the initial exploration of an alternative that is formalized as the PS pattern by the present work.

Figure 14 plots the single thread speedup of non-SIMD branching PS and *explicit* SSS in OpenLB 1.4 [28] relative to the SWAP pattern in OpenLB 1.3 [27]. As cell accesses in both versions are performed via an interface class the difference in performance w.r.t. the previous release is given largely due to the change in data structure from AoS to SoA and the newly optimal per-cell bandwidth of $2Q$ instead of $3Q - 1$.

Despite each cell's populations being explicitly reverted immediately after applying the collision step — at a time where the data is likely to reside in the cache — the memory operations produced by calling `Cell::revert` reduce the mean speedup from PS's 36% to 20% for explicit SSS.

In order to utilize vectorization, the virtual memory approach to PS was implemented. The existing CSE-optimized operators were adapted to accept a generic cell concept and instantiated with intrinsic wrappers. Finally, the collision loop was adapted into a masked sweep similar to e.g. Walberla [25].



Using double-precision LDC on SKL with 12 threads.
OpenLB 1.3: SWAP, 1.4: PS (branching), dev: PS (vmem, AVX-512).

Figure 15: Comparison of OpenLB versions on SKL

Table 8 provides an overview of the speedups obtained for the choices of floating point precision and SIMD ISA as well as compared to OpenLB 1.3, i.e. the last version prior to starting the optimization efforts. One particular aspect worth highlighting here is that as a side effect of vectorization the switch from double to single precision values yields a speedup of ∼ 1.71 whereas no previous release was able to obtain a advantage from this significant reduction of bandwidth requirements. However the unclear underlying issue is not resolved completely as is evident in the OpenMP results where the minimum observed speedup due to choosing single precision in the prototype is below 1 and the mean speedup still falls below the reference's mean speedup of 2.35.

Description		1.4	dev	PS ref
Single core speedup w.r.t. 1.3 (SWAP)				
Min		1.20	3.11	-
Max		1.46	3.98	-
Avg		1.36	3.58	-
Single core speedup float / double				
Min		0.87	1.36	1.76
Max		1.04	1.92	2.34
Avg		0.99	1.71	2.06
Single core speedup AVX-512 / AVX2				
Min		-	1.11	1.26
Max		-	1.39	1.62
Avg		-	1.24	1.40
Speedup float / double (OpenMP)				
Min		0.86	0.97	1.34
Max		1.10	2.30	3.73
Avg		0.99	1.84	2.35
Double precision performance (OpenMP)				
Min	[MLUPs]	185	302	327
Max	[MLUPs]	254	517	956
Avg	[MLUPs]	232	384	566
128^3	[MLUPs]	250	302	327

Table 8: Comparison of OpenLB performance aspects on SKL

Name		RTX	V100	A100
GPU		GeForce RTX 2070	Tesla V100	A100
Architecture		Turing	Volta	Ampere
Cores		2304	5120	6912
Memory	[GiB]	8	16	40
CUDA		11.2	11.2	11.2
Accessed via		Local Workstation	OVHcloud	HoreKa
Memory bandwidth (max. for problem sizes between 128^3 and 320^3)				
`triad`[0]	[GB/s]	400.3	826.8	1332.6
`update_19`	[GB/s]	386.2	799.2	1292.2
LBM bandwidth[1]				
`triad`	[MLUPs]	2634	5439	8767
`update_19`	[MLUPs]	2541	5258	8501

[0] Using the CUDA `triad` implementation provided by `BabelStream` [29]
[1] Number of MLUPs when using single precision FPT i.e. bandwidth/$(2 * 19 * 4 * 1e6)$

Table 9: Specifications of the GPUs used for the pattern benchmarks

5.2. GPU Performance

Table 9 summarizes the basic characteristics for the three target GPUs. As FP performance on GPUs has historically been optimized for single-precision values and bandwidth-saturating double-precision capabilities are still mostly restricted to HPC-focused GPGPUs, all benchmarks are performed for both precisions.

CUDA [30] was chosen as the environment instead of a portable option such as OpenCL for its single source approach and broad C++ support enabling direct sharing of most templatized LB specifics with existing CPU codes.

Table 10 provides an overview of obtained performance metrics using both single and double precision values. All three tested patterns produce good mean bandwidth saturations around 0.9 when using single-precision and peaking at full saturation for some samples. As no satisfactory equivalent to Likwid [23] for detailed reference bandwidth measurements could be found for GPU targets, per-size bandwidths were only measured using the same `update_19` case also used for CPU benchmarks (again similar to the approach in [24]).

As expected, double-precision benchmarks only yielded satisfactory bandwidth-relative performance on V100 and A100 whereas only a mean utilization of 0.51 was obtained on RTX. Notably, the throughput for small single-precision problems on RTX and A100 exceeded the results achieved by larger problem sizes due to cache effects. While a comparable peak for small problems was also observed on V100 it did not exceed the other results there.

No single pattern was observed to provide a consistent advantage over all samples when considering the approximately equal mean bandwidth utilizations and very similar total mean performance of SSS and the two PS variants. As a special case when considering only small cache-sensitive problem sizes on V100, SSS consistently provided the best performance by a margin up to 600 MLUPs. A similar difference was observed on A100 but not on RTX.

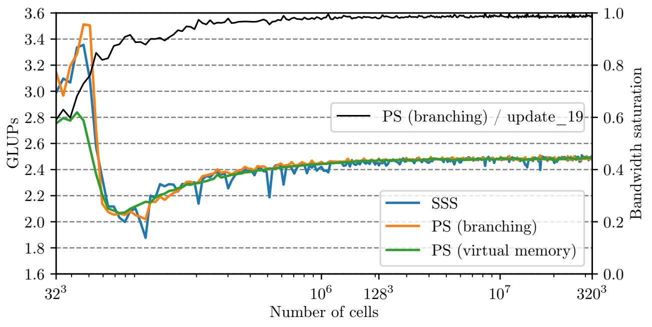


Figure 16: Performance for single-precision lattices on RTX

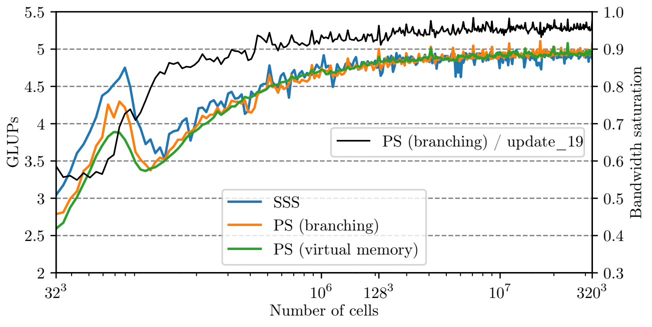


Figure 17: Performance for single-precision lattices on V100

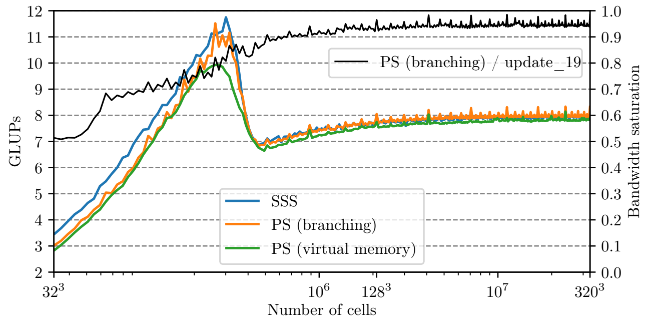


Figure 18: Performance for single-precision lattices on A100

Description		RTX			V100			A100		
		SSS	PS		SSS	PS		SSS	PS	
			branch	vmem		branch	vmem		branch	vmem
Single-precision performance										
Min	[MLUPs]	1877	2021	2066	3043	2788	2594	3443	3016	2824
Max	[MLUPs]	3356	3518	2881	4987	5108	5081	11749	11519	9926
Avg	[MLUPs]	2453	2466	2450	4726	4703	4691	7812	7786	7574
Single-precision bandwidth saturation w.r.t. `update_19`										
Min		0.56	0.59	0.52	0.62	0.55	0.53	0.58	0.51	0.48
Max		0.99	1.00	1.00	0.97	0.98	0.98	0.95	0.99	0.96
Avg		0.97	0.97	0.97	0.93	0.92	0.92	0.91	0.91	0.88
Double-precision performance										
Min	[MLUPs]	545	550	537	1862	1783	1737	2900	2667	2530
Max	[MLUPs]	641	643	642	2587	2612	2618	6638	6175	5493
Avg	[MLUPs]	631	632	631	2492	2493	2482	4258	4179	4152
Double-precision bandwidth saturation w.r.t. `update_19`										
Min		0.44	0.44	0.43	0.52	0.50	0.49	0.59	0.54	0.52
Max		0.56	0.56	0.55	1.00	1.00	1.00	0.99	0.97	0.98
Avg		0.51	0.51	0.51	0.94	0.94	0.94	0.94	0.93	0.93

Table 10: Overview of GPU performance results

SSS was observed to produce a larger spread of results for larger sample sizes with some significant lower outliers on
710 RTX and V100, likely caused by alignment problems that are masked by the larger access pattern space of PS.

Description		A100		
		SSS	PS	
			branch	vmem
Single-precision performance				
Min	[MLUPs]	7589	7698	7536
Max	[MLUPs]	7982	8332	8143
Avg	[MLUPs]	7873	7959	7768
Single-precision bandwidth saturation				
Min		0.91	0.92	0.90
Max		0.95	0.99	0.96
Avg		0.94	0.95	0.92
Double-precision performance				
Min	[MLUPs]	4156	4123	4131
Max	[MLUPs]	4270	4276	4282
Avg	[MLUPs]	4238	4199	4205
Double-precision bandwidth saturation				
Min		0.95	0.95	0.95
Max		0.97	0.97	0.98
Avg		0.97	0.96	0.96

Table 11: GPU results for problems larger than 128^3 on A100

Masking the SSS advantage for smaller lattice sizes by considering only larger non-cache impacted problems on A100 in Table 11, we observe very similar performance ac-
715 cross all patterns and samples — exceeding 0.9 saturation and peaking at 0.99 for branching PS. Qualitatively similar characteristics are also observed for RTX and V100.

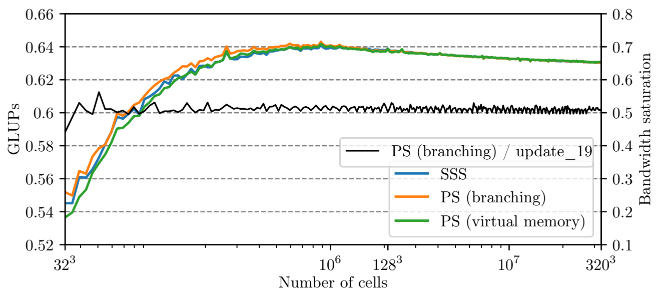


Figure 19: Performance for double-precision lattices on RTX

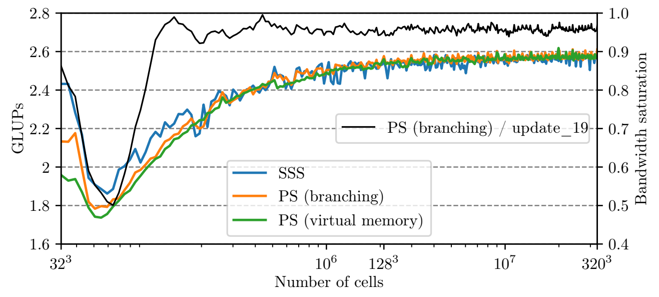


Figure 20: Performance for double-precision lattices on V100

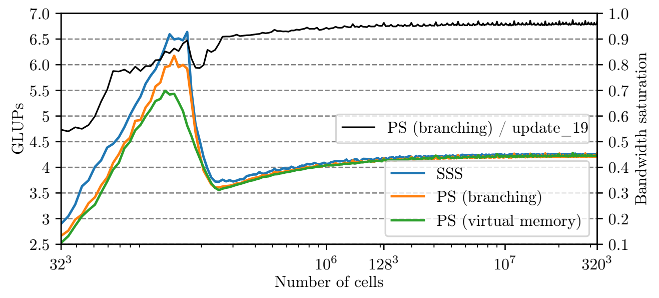


Figure 21: Performance for double-precision lattices on A100

Summarizing all GPU benchmarks, the best results are provided by SSS resp. branching PS for both floating point precisions. A clear SSS advantage for small problem sizes can be observed. Virtual memory PS is only competitive for larger problems but provides a lower predisposition to size-dependent fluctuations. Such fluctuations where performance changes significantly between adjacent resolution samples are most frequent on SSS.

As low-level access to virtual memory on GPUs is a rather new feature and the set of mapping functions is not yet on par with the functionality on CPUs (e.g. a separate physical buffer needs to be allocated for each population array instead of using offset mapping into a single shared buffer as on CPUs), improvements of virtual memory PS performance can be expected for the future. Furthermore, as the subpar results for small problem sizes are of limited relevance for practical applications, virtual memory PS's minimal demands on collision implementation as well as consistent performance between problem sizes render it into a viable pattern choice on GPUs.

Finally — similarly to CPUs — the pattern choice for practically relevant problem sizes depends primarily on the specific implementation context rather than performance considerations. All considered patterns were found to yield satisfactory bandwidth-relative performance close to and exceeding 0.9. GPU-specific fine tuning of e.g. block sizes and pre-shifting can be expected to enable further improvements.

6. Conclusion

Implicit propagation of directly addressed grids in LBM was considered in terms of SFC transformations utilizing the spatial invariance of neighborhood distances. Detailed descriptions of SSS and PS within this framework were formulated. A range of approaches to the efficient cyclic array rotations in PS were explored. Cyclic buffers relying on in-hardware virtual address translation were identified as a promising foundation for implementing PS on both SIMD CPUs and GPUs.

The performance of both patterns was evaluated w.r.t. to spatial isotropy and per-step time dependency. Relative performance anisotropy was found to be less pronounced for PS. A relationship between cuboid extent ratio, preferred dimension of the SFC and resulting performance was established. Per-step evaluation reproduced the expected alternating pattern for SSS due to it sharing the memory access pattern of A-A on directly addressed grids. A more complex pattern of time dependent performance was observed for PS and tied to the specific cache architecture of the targeted CPU. Cache aware pre-shifting was successfully explored as a possible mitigation.

Both patterns were implemented in lid driven cavity benchmark cases — utilizing adapted versions of existing SIMD CPU and CUDA GPU codes — for evaluation on a range of CPU and GPU targets. Both the total and bandwidth-related performance on Intel and AMD CPUs as well as different Nvidia GPUs was measured for a wide range of cavity sizes. Bandwidth-related results between 0.8 and full saturation were observed and the advantage of higher cache bandwidths was utilized.

While underperforming for small problems that benefit from cache-effects on ZEN and SKL, KNL's high bandwidth memory was translated into mean and maximum cell throughput values of 852 resp. 1017 MLUPs for PS utilizing AVX-512.

Both patterns consistently delivered mean bandwidth saturations exceeding 0.9 when using single precision on RTX, V100 and A100. For double-precision this was only realized on V100 and A100 due to computation constraints.

While SSS produced significantly better performance for small problem sizes on V100 / A100 and a consistent performance advantage for PS was observed on Xeon Phi, no single pattern was identified as superior in the general case. Both patterns were found to be bandwidth bound in most cases.

6.1. Application to OpenLB

The branching version of the PS pattern that is used by OpenLB 1.4 [28] was documented. The choice between SSS and PS in the specific implementation context of OpenLB was discussed and evaluated w.r.t. the previously utilized SWAP pattern.

Single core speedups up to 3.93 with a mean of 3.58 compared to SWAP in OpenLB 1.3 [27] were obtained by adding vectorized PS to OpenLB 1.4 [28]. Previously unavailable performance potential due to the larger cell throughput ceiling w.r.t. single-precision populations was utilized, yielding average single core speedups of 1.71. The same applied to cache-fitting problem sizes, yielding twofold improvements compared to the main memory limited performance.

Performance for problems beyond the scope of cache effects was found to approach saturation at 0.9 of Likwid reference measurements, approaching the throughput of the optimized reference implementation.

6.2. Acknowledgements

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior CAPES PROBRAL [88881.198766/2018- 01], Finance Code 001 - Brazil, and was financed in part by Alexander von Humboldt-Stiftung (AvH) [*BRA 1190826 HFST* CAPES-P] - Germany.

A100 benchmark simulations were performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

We thank Curtin University and DUG for supporting this work by providing access to an Intel Xeon Phi system.

References

[1] P. Bailey, J. Myre, S. Walsh, D. Lilja, M. Saar, Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors, in: 2009 International Conference on Parallel Processing, IEEE, Vienna, 2009, pp. 550–557. doi:10.1109/ICPP.2009.38.

[2] M. Mohrhard, G. Thäter, J. Bludau, B. Horvat, M. J. Krause, Auto-vectorization friendly parallel lattice Boltzmann streaming scheme for direct addressing, Computers & Fluids 181 (2019) 1–7. doi:10.1016/j.compfluid.2019.01.001.

[3] M. J. Krause, A. Kummerländer, S. J. Avis, H. Kusumaatmaja, D. Dapelo, F. Klemens, M. Gaedtke, N. Hafen, A. Mink, R. Trunk, J. E. Marquardt, M.-L. Maier, M. Haussmann, S. Simonis, Openlb—open source lattice boltzmann code, Computers & Mathematics with Applications 81 (2021) 258–288. doi:10.1016/j.camwa.2020.04.033.

[4] M. Haussmann, F. Ries, J. B. Jeppener-Haltenhoff, Y. Li, M. Schmidt, C. Welch, L. Illmann, B. Böhm, H. Nirschl, M. J. Krause, A. Sadiki, Evaluation of a Near-Wall-Modeled Large Eddy Lattice Boltzmann Method for the Analysis of Complex Flows Relevant to IC Engines, Computation 8 (2) (2020) 43. doi:10.3390/computation8020043.

[5] R. Trunk, C. Bretl, G. Thäter, H. Nirschl, M. Dorn, M. J. Krause, A Study on Shape-Dependent Settling of Single Particles with Equal Volume Using Surface Resolved Simulations, Computation 9 (4) (2021) 40. doi:10.3390/computation9040040.

[6] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice Boltzmann kernels, Computers & Fluids 35 (8-9) (2006) 910–919. doi:10.1016/j.compfluid.2005.02.008.

[7] M. Wittmann, T. Zeiser, G. Hager, G. Wellein, Comparison of different propagation steps for lattice Boltzmann methods, Computers & Mathematics with Applications 65 (6) (2013) 924–935. doi:10.1016/j.camwa.2012.05.002.

[8] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, J. Westerholm, An efficient swap algorithm for the lattice Boltzmann method, Computer Physics Communications 176 (3) (2007) 200–210. doi:10.1016/j.cpc.2006.09.005.

[9] M. Januszewski, M. Kostur, Sailfish: A flexible multi-GPU implementation of the lattice Boltzmann method, Computer Physics Communications 185 (9) (2014) 2350–2368. doi:10.1016/j.cpc.2014.04.018.

[10] Ł. Łaniewski-Wołłk, J. Rokicki, Adjoint lattice boltzmann for topology optimization on multi-gpu architecture, Computers & Mathematics with Applications 71 (3) (2016) 833–848. doi:10.1016/j.camwa.2015.12.043.

[11] M. Bauer, H. Köstler, U. Rüde, Lbmpy: Automatic code generation for efficient parallel lattice Boltzmann methods, arXiv:2001.11806 [cs] (apr 2020). arXiv:2001.11806.

[12] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, A. Scopatz, Sympy: symbolic computing in python, PeerJ Computer Science 3 (Jan. 2017). doi:10.7717/peerj-cs.103.

[13] M. Geier, M. Schönherr, Esoteric Twist: An Efficient in-Place Streaming Algorithmus for the Lattice Boltzmann Method on Massively Parallel Hardware, Computation 5 (4) (2017) 19. doi:10.3390/computation5020019.

[14] M. F. Mokbel, W. G. Aref, I. Kamel, Analysis of multi-dimensional space-filling curves, GeoInformatica 7 (3) (2003) 179–209. doi:10.1023/A:1025196714293.

[15] NVIDIA, Cuda release 10.2 (2020).
URL https://developer.nvidia.com/cuda-toolkit

[16] F. Robertsén, K. Mattila, J. Westerholm, High-performance SIMD implementation of the lattice-Boltzmann method on the Xeon Phi processor, Concurrency and Computation: Practice and Experience 31 (13) (2019) e5072. doi:10.1002/cpe.5072.

[17] J. L. Hennessy, D. A. Patterson, A. C. Arpaci-Dusseau, Computer Architecture : A Quantitative Approach, Morgan Kaufmann, San Francisco, CA, USA, 2017.

[18] S. Hou, Q. Zou, S. Chen, G. D. Doolen, A. C. Cogley, Simulation of Cavity Flow by the Lattice Boltzmann Method, Journal of Computational Physics 118 (2) (1995) 329–347. doi:10.1006/jcph.1995.1103.

[19] L.-S. Luo, W. Liao, X. Chen, Y. Peng, W. Zhang, Numerics of the lattice Boltzmann method: Effects of collision models on the lattice Boltzmann simulations, Physical Review E 83 (5) (May 2011). doi:10.1103/PhysRevE.83.056710.

[20] F. Kuznik, C. Obrecht, G. Rusaouen, J.-J. Roux, LBM based flow simulation using GPU computing processor, Computers & Mathematics with Applications 59 (7) (2010) 2380–2392. doi:10.1016/j.camwa.2009.08.052.

[21] C. Obrecht, F. Kuznik, B. Tourancheau, J.-J. Roux, A new approach to the lattice boltzmann method for graphics processing units, Computers & Mathematics with Applications 61 (12) (2011) 3628–3638. doi:10.1016/j.camwa.2010.01.054.

[22] N.-Q. Nguyen, A. J. C. Ladd, Lubrication corrections for lattice-Boltzmann simulations of particle suspensions, Physical Review E 66 (4) (Oct. 2002). doi:10.1103/PhysRevE.66.046708.

[23] J. Treibig, G. Hager, G. Wellein, Likwid: A lightweight performance-oriented tool suite for x86 multicore environments, in: Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, 2010.

[24] M. Wittmann, V. Haag, T. Zeiser, H. Köstler, G. Wellein, Lattice Boltzmann benchmark kernels as a testbed for performance analysis, Computers & Fluids 172 (2018) 582–592. doi:10.1016/j.compfluid.2018.03.030.

[25] M. Bauer, S. Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, U. Rüde, waLBerla: A block-structured high-performance framework for multiphysics simulations, Computers & Mathematics with Applications 81 (2021) 478–501. doi:10.1016/j.camwa.2020.01.007.

[26] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, F. Marson, J. Lemus, C. Kotsalos, R. Conradin, C. Coreixas, R. Petkantchin, F. Raynaud, J. Beny, B. Chopard, Palabos: Parallel Lattice Boltzmann Solver, Computers & Mathematics with Applications 81 (2021) 334–350. doi:10.1016/j.camwa.2020.03.022.

[27] M. Krause, S. Avis, D. Dapelo, N. Hafen, M. Haußmann, M. Gaedtke, F. Klemens, A. Kummerländer, M.-L. Maier, A. Mink, J. Ross-Jones, S. Simonis, R. Trunk, OpenLB Release 1.3: Open Source Lattice Boltzmann Code (May 2019). doi:10.5281/zenodo.3625967.

[28] M. Krause, S. Avis, H. Kusumaatmaja, D. Dapelo, M. Gaedtke, N. Hafen, M. Haußmann, J. Jeppener-Haltenhoff, L. Kronberg, A. Kummerländer, J. Marquardt, T. Pertzel, S. Simonis, R. Trunk, M. Wu, A. Zarth, OpenLB Release 1.4: Open Source Lattice Boltzmann Code (Nov. 2020). doi:10.5281/zenodo.4279263.

[29] T. Deakin, J. Price, M. Martineau, S. McIntosh-Smith, GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models, in: M. Taufer, B. Mohr, J. M. Kunkel (Eds.), High Performance Computing, Vol. 9945, Springer International Publishing, Cham, 2016, pp. 489–507. doi:10.1007/978-3-319-46079-6_34.

[30] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda, Queue 6 (2) (2008) 40–53. doi:10.1145/1365490.1365500.