LBM 3D stencil memory bound improvement with many-core processors asynchronous transfers

Minh Quan HO * ‡, Bernard TOURANCHEAU *, Christian OBRECHT †,
Benoît DUPONT DE DINECHIN ‡ and Julien HASCOET ‡
*LIG UMR 5217 - Grenoble Alps University - Grenoble, France
†CETHIL UMR 5008 - INSA-Lyon - Villeurbanne, France
‡Kalray S.A. - Montbonnot, France

***Abstract*—State-of-the-art academic and industrial many-core processors present an alternative to mainstream CPU and GPU processors. In particular, the 93-Petaflops Sunway supercomputer, built from NoC-based many-core processors, has opened a new era for high performance computing that does not rely on GPU acceleration. However, memory bandwidth remains the main challenge for these architectures. This motivates our approach for optimizing 3D Lattice Boltzmann Method (LBM) applications, one of the most data-intensive kinds of stencil computations, on many-core processors by using local memory and asynchronous software-prefetching. A representative 3D LBM solver is taken as an example. We achieve 33% performance gain on the Kalray MPPA-256 processor, by actively prefetching data compared to a "passive" programming model (OpenCL). We also introduce *two-wall*, a new LBM propagation algorithm which performs in-place lattice updates. This method cuts down memory requirement by half, reduces bandwidth losses in copying halo cells and offers another 5% improvement, delivering an overall 38% performance gain.**

I. Introduction

Since the last decade, Lattice Boltzmann Method (LBM) has become widely used in computational fluid dynamics for incompressible and weakly compressible flows [1]. A LBM computation is characterized by its stencil type, denoted D*d*Q*q*, where *d* is the space dimension number (one, two or three) and *q* is the number of *particle distribution functions* (PDFs) [2]. Physically, an LBM timestep on a lattice node consists of a *collision* and a *propagation* (aka. *streaming*) step. The collision applies a pre-defined physical model on the lattice's distribution vector of *q* values. The propagation then updates these new distribution information to the node itself and its $q-1$ neighboring nodes. Most well-known stencil types are D2Q5, D2Q9 and D3Q19 (see Fig. 1), or D3Q27.

From the programming point of view, LBM kernels are easy to implement and well suited for parallelization on recent multi-/many-core platforms. Lattice Boltzmann Methods are also known for their computational intensity and particularly high memory bandwidth requirements. Taking the example of a basic D3Q19 LBM, depending on collision operator, between 200 to 400 floating-point operations are performed on a lattice node per timestep. Most LBM implementations require storing all the 19 distribution values for each lattice node. A lattice domain $L \times L \times L$ contains $19 \times L^3$ floating numbers. Updating this lattice grid in a single timestep requires $2 \times 19 \times L^3$ memory operations (load and store) for less than $400 \times L^3$ arithmetic operations. Thus, simulating the whole lattice domain through T timesteps will generate a huge amount of data movement of $T \times 2 \times 19 \times L^3 \times sizeof(float)$ (bytes, single-precision) for $T \times 400 \times L^3$ floating-point operations. While recent architectures gain performance by multiplying the number of cores, memory systems can not keep pace with this trend. Previous studies [3] [4] show that LBM implementations are memory-bound and hardly obtain good performance on CPU or MIC processors. GPU-based accelerators, thanks to their graphic-dedicated high bandwidth memory, appear to be the most suitable platforms for LBM today. However, their low capacity of local memory inhibits optimization techniques for data prefetching (to reduce transfer time) and data sharing between cores (for stencil neighbor dependency).

Although other NoC-based many-core processors have much less global memory bandwidth, they embed significant amount of fast local memory [5] [6] and more predictability in data transfer. This enables explicit and efficient user buffer management for sophisticated optimizations, like software prefetching and streaming. This motivates our approach in developing a pipelined LBM algorithm on many-core based on local memory and asynchronous communication. Our algorithm is described in every detail and can be used on similar many-core architectures. This can also be generalized for other kinds of stencil computations which are less data-intensive and less repetitive such as image processing or CNN.

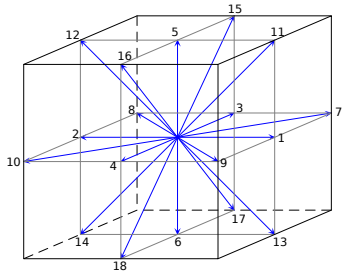


Fig. 1: LBM D3Q19 stencil

Contributions. Our key contributions are as follows:

1) Introduction of a new parallel algorithm for decomposing 3D stencil domains, by using local memory as user-buffers and asynchronous software-prefetching to build a *pipelined 3D stencil* kernel. The proposed approach is implemented

based on the LBM kernel from OPAL [4] and delivers 33% performance gain compared to the original OpenCL code on the Kalray MPPA-256 Bostan (2nd generation) processor.

2) Detailed description of using generic equations for dynamically calculating decomposition indices, sub-cube dimension and halo size, usable with or without *ghost layer* [7] and also applicable to 2D stencil.

3) Presentation of *two-wall*, a new LBM propagation method which performs in-place lattice update (*one-step one-lattice*). This approach reduces the memory requirement almost by half, while being computationally equivalent to the two-lattice method. Applying the two-wall method permits enlarging local cube size and reducing bandwidth lost in copying halo cells, thus yielding another 5% performance gain (MLUPS). Both DDR limitation and halo bandwidth are identified as the main performance factors on the MPPA processor.

The remainder of this paper is structured as follows. Section II lists some related works that we feel relevant for our contributions. In order to give a default bandwidth baseline of different architectures, Section III presents results of the GPU-STREAM [8] benchmark on parallel architectures of interest: (1) NUMA-CPU, (2) Xeon-Phi co-processor, (3) GPU and (4) a many-core processor (MPPA-256). The low-level asynchronous transfer primitives required for building *pipelined 3D stencils* are described in Section IV. Section V presents the overview and technical details of the new OPAL streaming algorithm using these asynchronous transfers. In Section VI, we describe the *two-wall* method for performing *local in-place lattice update*. Experimental results are presented in Section VII, and we conclude in Section VIII.

II. Related work

The straightforward method for implementing LBM is to use two instances of the lattice grid (*two-lattice*). Collision is carried out on data from the first grid and new distribution values (propagation) are written to the second grid. The two lattice grids are then swapped for the next timestep. *One-step two-lattice* with fused collision and propagation was first introduced by Massaioli and Amati [9]. In the fused kernel, propagation can be done either before (pull scheme) or after collision (push scheme). In spite of its implementation simplicity, the two-lattice method results in unaffordable memory allocation on large domains. Pohl et al. [10] proposed a *compressed-grid* (aka. *shift*) approach to reduce double-memory requirement of the two-lattice method. In the same goal, Mattila et al. [11] developed a *swap* algorithm that requires almost half of memory allocation compared to the two-lattice method. Comparisons of these algorithms with varying lattice-indexing and data layouts were carried out in [7] [12]. These studies show advantage of equivalent high computational capacity of *compressed-grid* and *swap* method compared to the two-lattice, while being low memory consumption. However, both these two approaches require careful iteration order and complex index calculation for shifting the two lattice grids (*shift*) or swapping distribution values between neighbors (*swap*). These dependencies make it impossible to implement *shift* and *swap* algorithms on GPUs and accelerators in offloading models (CUDA, OpenCL). Today, these approaches are only implemented by sequential CPU code inside a domain and are parallelized over CPU clusters by using MPI for inter-domain halo exchanging (enabling weak-scaling but not strong-scaling).

Most existing LBM implementations on GPU employ the fused two-lattice approach as the easiest and most high-computational method. In particular, OpenCL Processor Array LBM (OPAL) [4] implements a one-step two-lattice solver of a 3D *lid-driven cavity* fluid [13], based on the D3Q19 stencil. OPAL is designed to be simple and portable on GPUs, accelerators and other OpenCL-enabled devices. Multiple-relaxation-time (MRT) collision [14] is fused with propagation in a single OpenCL kernel in *pull* scheme and *bounce-back* boundary conditions. The collision kernel performs about 300 instructions (most of them are floating-point arithmetic operations) per lattice update. Switching between SOA (Structure-of-Array), AOS (Array-of-Structure) and TA (Tiled-Array) layouts is possible for selecting the best architecture-dependent memory acces patterns.

In the related work on porting a 3D seismic wave propagation on the MPPA processor, Castro et al. [15] developed a *2D-prefetching* algorithm for anticipating data transfers between global memory and local memory. Their application is a D3Q13 stencil, touching four neighboring nodes from the center node $(-2, -1, +1, +2)$ in each dimension. The 3D domain is decomposed in small 2D planes which are copied as many as possible to the local memory (up to 8 planes). They observed important waiting time for data arrival without identifying clearly the DDR bandwidth limitation of the MPPA. The *must-lost* halo bandwidth effect was not studied either.

McIntosh-Smith et al. implemented an OpenCL D3Q19-BGK LBM solver [3] which is very similar to OPAL. OPAL results obtained on CPU Xeon, co-processor Xeon Phi and GPU concur with those of their implementation. They studied the impact of OpenCL work-group size on LBM performance, which is known to be the key tuning parameter on nowadays OpenCL implementations and is very platform-dependent.

Januszewski et al. [16] presented Sailfish-CFD, a feature-rich python-based LBM solver for 2D and 3D problems. Real-time visualization is supported for online monitoring. Kernel codes are generated through flexible *Mako* templates and launched on device using either *pycuda* or *pyopencl*. We tried out the 128^3 3D lid-driven cavity example from Sailfish and obtained similar MLUPS performance to OPAL on the same Tesla C2070 GPU. Using and extending Sailfish to generate hardware-specific kernels would be interesting. However, the paper focus is on a new algorithm whose kernel is kept as simple as possible.

Polybench [17] was particularly developed for benchmarking polyhedral compilers (PPCG [18], Pluto [19] and Pochoir [20]). It contains examples from linear algebra, image processing and 2D/3D stencils. The main advantage of polyhedral compilers is optimizing naive code by tiling on CPU and

parallelisation through OpenMP. PPCG also targets GPU and other accelerators by generating OpenCL code. The performance gain of using PPCG on GPU depends on memory pattern of applications and their legitimacy to polyhedral analysis. Recent studies [18] [21] on comparing GPU using PPCG vs. sequential CPU code show up to 1000x speedup on a general matrix-multiply (`GEMM`), but only 10x on a 2D stencil (`jacobi-2d`). Any above-mentioned GPU LBM solver would be at least 50x faster than on a single CPU.

III. GPU-STREAM Benchmark

In this section, we briefly present the sustained memory bandwidth of different multi-/many-core platforms on the GPU-STREAM [8] benchmark. These scores give a performance baseline, enabling a better understanding of further comparisons and discussions in this paper. The GPU-STREAM benchmark contains four different kernels Copy, Mul, Add, Triad and is run "out-of-the-box" on four architectures: (1) GPU NVIDA Tesla C2070, (2) Co-processor Intel Xeon Phi 3100, (3) NUMA 2x 8-core Intel Xeon CPU E5-2667v3 and (4) Many-core Kalray MPPA-256 Bostan. The returned vector of each run is tested for correctness against a host vector computed through the same computational steps. For the sake of briefness, only Triad bandwidth are presented here in Table I and in Fig 2a. Three other tests (Copy, Mul, Add) give similar performance.

Platform	Peak DDR	Triad (GPU-STREAM)	% peak
Tesla C2070	144 GB/s	100 GB/s	69%
Xeon Phi 3100	240 GB/s	68 GB/s	28%
Xeon E5-2667v3	68 GB/s	35 GB/s	51%
MPPA-256 Bostan	8.5 GB/s	2.5 GB/s	29%

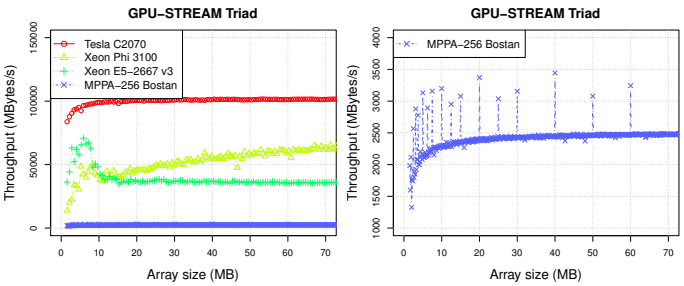
TABLE I: GPU-STREAM Triad bandwidth

Beside the stable expected high bandwidth on the GPU Tesla, we observe cache-size performance effect on CPU and MIC. As seen from Fig. 2a, last-level cache on these systems (20 MB on Xeon E5-2667v3 and 28.5 MB on Xeon Phi 3100) can hold entirely the three vectors of size less than 6.6 MB and 9.5 MB, respectively, thus reach highest "warm-up" bandwidth. In Fig. 2b, Triad bandwidth on MPPA obtains in most of the cases 2.5 GB/s. Some special vector sizes (such as 10, 20, 30 or 40 MB) can reach up to more than 3 GB/s, as they might be more favorable to the MPPA global cache system, nowadays emulated by software through network-on-chip with a high cache-miss cost.

McIntosh-Smith et al. [3] summarized the effective throughput B of a D3Q19 stencil in term of MLUPS performance P by the following formula:

$$B = \frac{P \times 10^6 \times 19 \times 2 \times 4}{1024^3} \ (GB/s) \qquad (1)$$

The term $P \times 10^6$ is the performance in LUPS, means the total number of lattices updated per second. The factors $19 \times 2 \times 4$ come from 19 speeds of a lattice node (D3Q19), with a read and a write operation (2) during each update step, and numbers are stored in single-precision floating point format



(a) Triad on four platforms (b) Triad on Kalray MPPA-256

Fig. 2: GPU-STREAM Triad on different platforms ($c = a + (\alpha \times b)$)

($sizeof(float) = 4$). The division by 1024^3 converts from B/s to GB/s. This formula can also be rewritten to get the best-case performance from the STREAM bandwidth:

$$P_{best} = \frac{B_{stream} \times 1024^3}{19 \times 2 \times 4 \times 10^6} \ (MLUPS) \qquad (2)$$

This means that we cannot get better than 700 MLUPS on Tesla C2070, 480 MLUPS on Xeon Phi 3100, 250 MLUPS on CPU Xeon and about 15 MLUPS on MPPA in any situation. Being aware of this limitation is important for us to propose optimizations on non-GPU many-core processors, by taking advantage of their local memory and by anticipating data transfers early enough to cover the global memory latency.

IV. Low-level 3D asynchronous API

Inspired from popular communication APIs such as Cray SHMEM [22], ORNL ARMCI [23] and one-sided MPI subset [24], Kalray developed an asynchronous communication library. This low-level API exploits DMA engines for one-sided *put/get* transfers between the global DDR memory (partitioned by `segment`) and the compute cluster's (CC) local memory [5]. The MPPA asynchronous API is designed for high performance and low memory footprint, with support of contiguous, strided, 2D/3D data transfers and also remote atomic operations. Inter-cluster communications are also supported in the library. There is no MPI-like rank for application processing the IO clusters, instead the IO cores run firmware providing event-based asynchronous services to the CCs. This approach is different from previous works on porting MPI on many-core processors [25] [26].

In this section, we briefly present some essential primitives from the MPPA Async API for performing asynchronous communications on 3D buffers while keeping their layout unchanged. An example is copying a sub-cube S from a remote cube R to a local buffer A, do some computations on S then re-copy it from A to R ($S \subset R, A$). These primitives are demonstrated in Fig. 3. In a put/get function, if `event` is set, the function fills a pending asynchronous job transaction ticket to the *event* and returns immediately (non-blocking). One can further come back and wait on it by calling `mppa_async_event_wait(event)`. Otherwise, when `event` is `NULL`, the function is blocking until the

buffer is ready to be reused (`put`) or when the data are received (`get`). The type `mppa_async_point3d_t` describes the copy-position and the three dimensions of the 3D buffer to read or write. The sub-cube S is represented by $width \times height \times depth$ elements, whose $size$ indicates the element size in bytes.

```
typedef struct {
  int xpos; int ypos; int zpos; /* copy index */
  int xdim; int ydim; int zdim; /* hosting cube
    dimensions */
} mppa_async_point3d_t;

/** 3D asynchronous transfer from remote to local */
int mppa_async_sget_block3d(void *local,
  mppa_async_segment_t *segment, uintptr_t offset,
  size_t size, int width, int height, int depth,
  const mppa_async_point3d_t *local_point,
  const mppa_async_point3d_t *remote_point,
  mppa_async_event_t *event);

/** 3D asynchronous transfer from local to remote */
int mppa_async_sput_block3d(const void *local,
  mppa_async_segment_t *segment, uintptr_t offset,
  size_t size, int width, int height, int depth,
  const mppa_async_point3d_t *local_point,
  const mppa_async_point3d_t *remote_point,
  mppa_async_event_t *event);

/** Wait for transfer-related event */
int mppa_async_event_wait(mppa_async_event_t *event)
```

Fig. 3: A part of MPPA Async API for 3D transfer

V. Pipelined 3D stencil

A. Global algorithm and pipeline depth

Our implementation uses the MPPA POSIX programming model [5] where each compute cluster appears as and independent 16-core CPU, using the MPPA asynchronous operations library for data transfers between local and global memories. Although the MPPA platform also supports OpenCL, this standard [27] only proposes asynchronous 1D transfers with contiguous (`async_work_group_copy`) or strided (`async_work_group_strided_copy`) accesses on the global memory and contiguous accesses on the local memory. Thus, implementing any kind of `async_work_group_copy_3d` over these builtins will not be efficient. So the first step consists in re-writing OPAL's OpenCL kernels to standard C code to run on the compute clusters. The one-step two-lattice method with *pull* scheme originally implemented in OPAL is re-applied. Two instances of the 3D lattice grid (*LatticeEven*, *LatticeOdd*), each containing $Lx \times Ly \times Lz$ nodes, are allocated on the global DDR memory and are accessed in node-wise layout, i.e distribution values of a lattice are stored consecutively. The second step divides the lattice domain into many *mini-cubes* (see Fig. 4), then copy and compute them one-by-one on the CC local memory. Each mini-cube is defined as a $Cx \times Cy \times Cz$ sub-cube. From now on, we call $d \in \{x, y, z\}$ the dimension d of the three-dimensional Cartesian space. We define the total number of mini-cubes:

$$NB\_CUBES = \prod_{d \in \{x,y,z\}} NB\_CUBES\_d \tag{3}$$

as the dot product of the number of mini-cubes in each dimension. For the sake of simplicity, in this paper, we assume:

$$\begin{gathered} Cd | Ld \quad and \quad NB\_CUBES\_d = \frac{Ld}{Cd} = m_d \\ Cd \in \{2^{n_d}\} \quad n_d, m_d \in \mathbb{N}^+ \end{gathered} \tag{4}$$

Besides, we denote a constant $CFd = Cd + h$ (CF standing for cube-full) the extended cube size with halo layers (h) added ($h = 2$ on the D3Q19 stencil). Thus, updating a cube of $Cx \times Cy \times Cz$ lattices involves fetching $S = CFx \times CFy \times CFz$ storage to local memory. This requirement is true for most cases (non-boundary cubes - e.g cube 4 of Fig. 8). On boundary cubes (e.g cubes 0, 1, 2, 3 of Fig. 8), the copied cube (Sx, Sy, Sz) should be adjusted (say by adding a *halo-cutoff* - this will be explained later) to deal with solid nodes. A local cube-slot must so be allocated to $CFx \times CFy \times CFz$ to match any case.

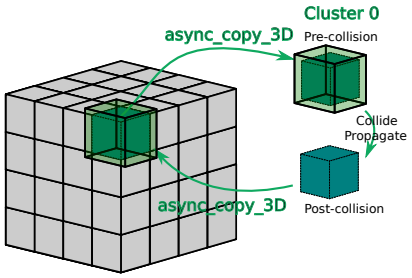


Fig. 4: 3D LBM decomposition
Main-block cube is copied with its surrounding halo layers (if exists). One extra cube is needed to store post-collision state.

Algorithm 1 sums up the mono-cluster context where compute cluster 0 (CC0) is updating all the `NB_CUBES` mini-cubes within an LBM timestep. These mini-cubes are ordered in a *macro-pipeline* using asynchronous *put* and *get* to overlap computation and communication. We also apply the two-lattice method on local memory, i.e the number of buffer slots is doubled, one for fetching pre-collision cube (S) from the first global lattice grid and one for storing post-collision cube (S') to put to the second lattice grid later. The pre-collision cube is allocated at $CFx \times CFy \times CFz$, while the post-collision cube only needs $Cx \times Cy \times Cz$ storage. Fig. 4 only draws one global lattice grid for compactness, but it should be understood that the local post-collision cube is *put* to the second lattice grid instead of the first one (then swapped at the next timestep).

In reality, we use all the 16 compute clusters and 16 cores per cluster on MPPA (multi-cluster multi-PE). Each core is referred to as a Processing Element (PE). Within a compute cluster, multi-threading is enabled by spawning up to 15 threads, one per PE, from the PE 0 in the POSIX-pthread fashion (`create`, `join`). Each CC is then responsible for `NB_CUBES_PER_CLUSTER` with three values of

NB_CUBES_PER_CLUSTER_d and so on. Note that in the two-lattice method (of global lattice grids), updating mini-cubes (even with halo copy) within a single timestep is really independent and can be spread on multiple CCs. Only one synchronization barrier at the end of each timestep is needed between all CCs to avoid *data-racing* at the next timestep. The barrier cost of this BSP-like model [28] is negligible since all CCs have the same workload (NB_CUBES_PER_CLUSTER) and no inter-cluster data exchange is required (only *put* and *get* to/from global memory).

Algorithm 1 Explicit macro-pipeline of 3D stencil using double-buffering within a timestep

```
1: /* Prolog: get first cube */
2: prefetch_cube(0);
3: /* Pipeline */
4: for i in 0 .. NB_CUBES-1 do
5:    if i < NB_CUBES-1 then
6:       prefetch_cube(i+1);  // get next cube (non-blocking)
7:    end if
8:    wait_cube(i);          // wait current cube
9:    compute_cube(i);       // compute current cube
10:   put_cube(i);           // put back to global (non-blocking)
11: end for
12: /* Epilog: wait last put and barrier */
13: wait_cube(NB_CUBES-1);
14: barrier_all_clusters();
```

The double-buffering (2-depth) pipeline in Algorithm 1 is the most basic algorithm where communication is overlapped by only one compute-step. As computations are faster than data transfers, deeper pipelines such as triple- or quadruple-buffering (see Fig. 5 and Fig. 6) provide better overlap, but also require more local memory. In the three next sub-sections, we propose methods for solving following questions:

- How to distribute fairly and exclusively all mini-cubes across CCs with their proper cube-indexing ?
- Which cube size and pipeline depth to choose to fit the local memory and their best trade-off ?
- When copying a mini-cube, where is its correct begin position with halo added, also its dimensions ? Without using ghost layer, how should the halo layer and begin position be adjusted, especially for boundary cubes ?

B. Cube distribution

Given a CC identified by cid $\in$ [0..NB_CLUSTERS-1], its working cubes (icube) is in a one-dimensional range from cid*NB_CUBES_PER_CLUSTER to (cid+1)*NB_CUBES_PER_CLUSTER. Mapping **bijectively** this 1D domain (icube) to a 3D one (icubex, icubey, icubez) for spatial cube indexing (see Fig. 4) can be done by *space-filling-curves* [29], such as Morton or Hilbert etc.. These curves have been efficiently implemented elsewhere by bit-interleaving [30] or lookup-table [31]. However, Morton, Hilbert and other curves mostly work for square-cubic grid where the number of elements in all dimensions is equal. In our 3D decomposition scheme, despite the global lattice domain can be square ($Lx = Ly = Lz$), mini-cubes may be not ($Cx \neq Cy \neq Cz$) due to many reasons (see the next section), thus these curves is not always applicable for adressing mini-cubes, as NB_CUBES_d can be different.

For solving this problem, we implement a simple alternative bijective function $f : \mathbb{N} \rightarrow \mathbb{N}^3$ in Fig. 7, following the *3D-row-major* layout for indexing mini-cubes. Each conversion by f takes less than 10 instructions (same as Morton or Hilbert). If NB_CUBES_d macros are wisely chosen to be power-of-two, only *bit-shifting* operations will be generated by compiler instead of multiplication and division.

C. Local cube dimensions

In most of the cases, a square cube would be ideal for coding and optimizing. However, the local memory of 2 MB on each MPPA compute cluster [5] is quite small and is used for hosting an embedded operating system, services and storing user application. The remaining space of about 1.5 MB is available for local buffers. Some auxiliary storages are needed in LBM for macroscopic monitoring (velocity, density...). The maximal allocable space for local pre-collision and post-collision cubes is around 1.4 MB. Besides, copying halo cells not only requires local storage, but also consumes memory bandwidth. Hereafter, we refer to "*halo bandwidth*" (*halo BW*) as the bandwidth lost in fetching halo layers. This halo bandwidth is defined as the fraction between the number of halo cells and the total number of copied cells (main-block + halo). This ratio can be significant on small cubes. For example, given a square cube whose main-block size is $Cx \times Cx \times Cx$, its halo bandwidth is:

$$g(Cx, Cx, Cx) = \frac{(Cx+2)^3 - Cx^3}{(Cx+2)^3} = O(\frac{1}{Cx})$$
$$\lim_{Cx \to \infty} g(Cx, Cx, Cx) = 0 \quad (5)$$
$$\textit{Example: } g(16, 16, 16) = \frac{18^3 - 16^3}{18^3} \approx 0.29$$

The best performance is achieved when the *main-block* cube volume ($Cx \times Cy \times Cz$) is maximized and the halo bandwidth is minimized. Also, local storage should be reduced as much as possible. Let using single-precision type (float), applying a D-depth pipeline for the two-local-cube method described

	Prologue	icube=0	1	2	3	4	5	6	7	Epilogue
		ibuf=0	1	2	0	1	2	0	1	
buffers[0]	G	WCP	WG		WCP	WG		WCP	W	
buffers[1]	G		WCP	WG		WCP	WG		WCP	W
buffers[2]		G		WCP	WG		WCP	W		

Fig. 5: 3-depth pipeline (triple-buffering)
icube: index of cube to compute, ibuf: index of local buffer slot
`G = GET; P = PUT; W = WAIT; C = COMPUTE;`
`WCP = {WAIT + COMPUTE + PUT}; WG = {WAIT + GET}`
This pipeline allows 2-step distance between `GET` and `WAIT`, but only 1-step distance between `PUT` and `WAIT`, thus the `PUT` will not be well overlapped.

	Prologue	icube=0	1	2	3	4	5	6	7	Epilogue
		ibuf=0	1	2	3	0	1	2	3	
buffers[0]	G	WCP		WG		WCP		W		
buffers[1]	G		WCP		WG		WCP		W	
buffers[2]		G		WCP		WG		WCP		W
buffers[3]			G		WCP		WG		WCP	W

Fig. 6: 4-depth pipeline (quadruple-buffering)
This pipeline allows 2-step distance between both `GET-WAIT` and `PUT-WAIT`, better overlapping than on a 3-depth pipeline.

```
void cube_index_1to3(int icube,            /*input*/
  int* icubex, int* icubey, int* icubez) /*outputs*/
{
  int z = icube/(NB_CUBES_X*NB_CUBES_Y);
  int y = (icube-z*(NB_CUBES_X*NB_CUBES_Y)) /
    NB_CUBES_X;
  int x = icube-z*(NB_CUBES_X*NB_CUBES_Y) -
    y*NB_CUBES_X;
  *icubez = z; *icubey = y; *icubex = x; /*outputs*/
}
```

Fig. 7: 3D Row-major cube-indexing $f : \mathbb{N} \to \mathbb{N}^3$

above should satisfy the linear-programming formulation:

$$\begin{aligned}
&\textbf{Find: } (D, Cx, Cy, Cz)\\
&\textbf{Maximize: } Cx \times Cy \times Cz \quad \textit{(\# lattices updated per cube)}\\
&\textbf{Minimize: } CFx \times CFy \times CFz \quad \textit{(per-cube storage)}\\
&\textbf{Minimize: } \tfrac{(CFx\times CFy\times CFz)-(Cx\times Cy\times Cz)}{CFx\times CFy\times CFz} \quad \textit{(halo BW)}\\
&\textbf{Subject to:}\\
&\tfrac{D\times((CFx\times CFy\times CFz)+(Cx\times Cy\times Cz))\times 19\times 4}{1024^2} \leq 1.4\\
&CFd = Cd + 2\ ; \quad d \in \{x, y, z\}\\
&Cx, Cy, Cz \in \{2^n\}\ ; \quad N, n \in \mathbb{N}^+
\end{aligned} \tag{6}$$

We choose $D \geq 3$ in order to have better overlapping than on a 2-depth pipeline. This choice restricts to a very small searching domain ($Cd \leq 128$) and can be resolved by running *branch-and-bound* algorithm in a script. Solutions can be $(D, Cx, Cy, Cz) = (3, 16, 8, 16)$ (with 36% halo BW) or $(D, Cx, Cy, Cz) = (4, 8, 8, 16)$ (with 43% halo BW). Permutation of Cx, Cy, Cz also gives other satisfying solutions, with the same halo bandwidth percentage. On another hand, increasing pipeline depth is not relevant, because the higher D is, the smaller $\{Cx, Cy, Cz\}$ will be, thus halo bandwidth will become unacceptable to MPPA with its modest DDR bandwidth (Sect. III). Moreover, compute cores will switch between small cubes more often. Accumulated waiting time will also be more important due to exponential number of DMA requests and processing overhead by the DDR asynchronous server.

D. Local and remote copy-index management

Despite using *ghost layer* [7] surrounding the computational domain simplifies implementation of the *streaming* step at boundary cells, we choose not to use this in our work. The first reason is to minimize global memory allocation and avoid wasting bandwidth/storage in moving ghost cells (in addition to halo cells) to/from local memories, which is critical on a many-core processor like MPPA. Secondly, OPAL's boundary conditions were not implemented with *ghost layer*. Thus, rewriting its OpenCL kernels to C code will be trivial and less error-prone. However, in our 3D decomposition algorithm, this decision complicates copy indexing, cube-size and *halo-cutoff* management depending on sub-cube's geometric position (see Fig. 8, drew in 2D for simplicity).

In this section, we present generic formulae for calculating dynamically these variables (`BAd, BRd, Sd`) from cube indices. Our method is detailed in Eq. 7, Table II and illustrated by Fig. 8. Then Fig. 9 demonstrates how we use these variables in arguments of `async_{get|put}` functions of the MPPA Async library. These formulae can also be used on other implementations that use *ghost layer*, by setting all `remote_offset` to -1 (i.e allowed to jump out of the computational domain, thanks to *ghost layer*) and all

`local_offset`, `halo_cutoff` to zero (i.e always copy "full cube" CFd to A, instead of copying Sd to BAd).

$$
\begin{aligned}
const\ A &= (Ax, Ay, Az) = (0,0,0) \\
R &= (Rx, Ry, Rz) \\
&= (icubex \times Cx,\ icubey \times Cy,\ icubez \times Cz) \\
BAd &= Ad + local\_offset(icubed, NB\_CUBES\_d) \\
BRd &= Rd + remote\_offset(icubed, NB\_CUBES\_d) \\
Sd &= CFd + halo\_cutoff(icubed, NB\_CUBES\_d) \\
d &\in \{x, y, z\}
\end{aligned} \tag{7}
$$

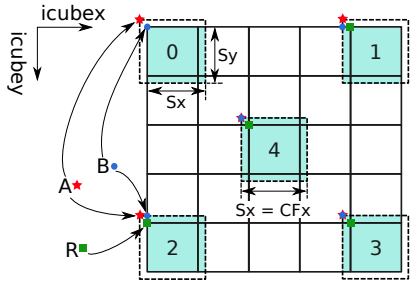


Fig. 8: Local/Remote copy-index in 2D (in lattice node)
A: origin of the local buffer = (0,0)
R: begin of the remote main-block cube (without halo)
B: begin of the copied cube (S), represented by:
BA: index of S on local memory (from A)
BR: index of S on global memory (from R)

	local_offset (from point A)	**remote_offset** (from point R)	**halo_cutoff** (from CFd)
icubed == 0	1	0	-1
0 <icubed && icubed <NB_CUBES_d-1	0	-1	0
icubed == NB_CUBES_d-1	0	-1	-1

TABLE II: Copy-index offset and halo-cutoff (in lattice node). Fig. 8 illustrates values in this table.
This table is implemented using three macros:

```
#define local_offset(icubed, NB_CUBES_d)
#define remote_offset(icubed, NB_CUBES_d)
#define halo_cutoff(icubed, NB_CUBES_d)
```

VI. 3D LOCAL IN-PLACE LATTICE UPDATE

In the previous section, we introduced how to apply the two-lattice method in CC local memory through pre- and post-collision cubes. This method requires important storage that limits the maximum cube size (D, Cx, Cy, Cz) on MPPA to $(3, 16, 8, 16)$ with 36% halo BW (see Sect. V-C). In this section, we present a new LBM propagation algorithm, named *two-wall* method, in which local pre-collision cube (S) is updated "in-place" (*one-step one-lattice*) instead of writing post-collision data to the second cube S' (*one-step two-lattice*). This method reduces almost by half the local memory requirement, by introducing two small "wall" buffers (`float tmp_wall[2][CFz][CFx][19]` $\sim O(30KB)$) shared among 16 PEs of each compute cluster. By removing the post-collision cube S', the pre-collision cube S can be

```
/*************************************************/
/*    Arguments for fetching S (with halo)       */
/*(remote to pre-collision cube (CFx * CFy * CFz))*/
/*************************************************/
mppa_async_point3d_t local_point = {
  .xpos=BAx, .ypos=BAy, .zpos=BAz, /*local index*/
  .xdim=CFx, .ydim=CFy, .zdim=CFz  /*local buffer*/
};
mppa_async_point3d_t remote_point = {
  .xpos=BRx, .ypos=BRy, .zpos=BRz, /*remote index*/
  .xdim=Lx,  .ydim=Ly,  .zdim=Lz   /*remote buffer*/
};
size_t size = 19 * sizeof(float); /* D3Q19 */
int width  = Sx;                  /* width  of S */
int height = Sy;                  /* height of S */
int depth  = Sz;                  /* depth  of S */

/*************************************************/
/*    Arguments for putting S' (no halo)         */
/*(post-collision cube (Cx * Cy * Cz) to remote) */
/*************************************************/
mppa_async_point3d_t local_point = {
  .xpos=0,  .ypos=0,  .zpos=0,   /*local index*/
  .xdim=Cx, .ydim=Cy, .zdim=Cz   /*local buffer*/
};
mppa_async_point3d_t remote_point = {
  .xpos=Rx, .ypos=Ry, .zpos=Rz,  /*remote index*/
  .xdim=Lx, .ydim=Ly, .zdim=Lz   /*remote buffer*/
};
size_t size = 19 * sizeof(float); /* D3Q19 */
int width  = Cx;                  /* width  of S' */
int height = Cy;                  /* height of S' */
int depth  = Cz;                  /* depth  of S' */
```

Fig. 9: Copy index and cube size in MPPA Async

enlarged to $(3, 16, 16, 16)$ (29% halo BW) or $(4, 16, 8, 16)$ (36% halo BW) by a new linear equation in place of Eq. 6:

$$\frac{((D \times CFx \times CFy \times CFz) + (2 \times CFx \times CFz)) \times 19 \times 4}{1024^2} \leq 1.4 \tag{8}$$

Fig. 10 illustrates the two-wall method on a pre-collision cube S of size $CFx \times CFy \times CFz$. The main-block $Cx \times Cy \times Cz$ cube is arranged in Cy walls, each of size $Cx \times Cz$ and is updated wall-by-wall. Before updating the wall y ($y \in [1..CFy - 1]$) from the current timestep t to $t+1$, the `current_wall` is used for saving pre-collision data of the wall y before y gets modified by the in-place post-collision propagation. Idem, the `past_wall` contains pre-collision data of the wall $y-1$, as this wall in S has been changed to the timestep $t+1$ previously, thus can not be used here for the wall y at time t. Note that the halo wall $y = 0$ is copied firstly to the `past_wall` as prologue in order to begin at $y = 1$. Furthermore, in the D3Q19 stencil, only 15 distribution values are needed in each wall element, instead of 19, as the 4 speeds are read directly from S for $y+1$ neighbors ("future wall" - see Fig. 1). Hence, the two wall buffers can even be tuned to save some more kilobytes.

Once the wall y is updated, all PEs perform a local synchronization barrier for memory coherency and to avoid *data-racing*, move to $y+1$ and swap `current_wall` and `past_wall` (e.g `tmp_wall[(y+1)% 2]`). The `current_wall` of the wall y now becomes the `past_wall` of the wall $y+1$. Inversely, the `past_wall` of the wall y is

now the `current_wall` of the wall $y+1$ and can be recycled for saving pre-collision state of the wall $y+1$, before itself, in turn, is overwritten.

In detail, each PE is working on p lattice-plans, each of size $Cx \times Cy$, with $p = \frac{Cz}{NB\_PE}$. For the sake of simplicity, we choose $Cz = 16$ so that $p = 1$. A wall-copy is performed by 16 PEs: each PE copies one line, except the PE 0 and 15 which will copy two lines to cover the two extra halo lines at the top and bottom of the wall ($CFd = Cd+2$). A lattice-plan, associated to a PE `pid`, is updated line-by-line (Cy lines of Cx nodes per line), following the Algorithm 2. A local barrier of 16 PEs is done at the end of each line to create a wall-update effect as mentioned in the above paragraph.

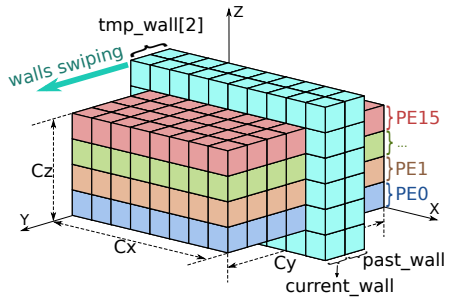


Fig. 10: In-place update on a local cube with a 16-PE cluster using *two-wall* method.

Algorithm 2 Run on each PE: In-place update

```
1:  Input: pre-collision cube (S), PE id (pid)
2:  z = pid;                          // my lattice-plan
3:  All PEs copy wall zero of S to past_wall;
4:  // walls swipping
5:  for y in 1 .. CFy-1 do
6:     All PEs copy wall y from S to current_wall;
7:     // update line of Cx nodes
8:     for x in 1 .. CFx-1 do
9:        collide and in-place update node (x,y,z) in S, using
          current_wall, past_wall, and wall y + 1;
10:    end for
11:    local barrier all PEs; // before advancing in y
12:    swap current_wall and past_wall;
13: end for
```

VII. Results and Discussions

A. Original OPAL OpenCL

In this section, we reproduce experimental results of OPAL in [4] on our available platforms (GPU, CPU and MIC). Moreover, in this work, we add OPAL performance on the MPPA-256 processor, then compare the relative throughput and power efficiency of the four architectures. Vendor OpenCL libraries used in this work are as follows:

- GPU Tesla C2070: NVIDIA linux driver v352.55
- Xeon Phi 3100 KNC: Intel OpenCL SDK 14.2
- Xeon CPU E5-2667v3: Intel OpenCL SDK 14.2
- MPPA-256 Bostan: Kalray Accesscore 2.6-rc, for both OpenCL and Async-copy

By default, MPPA-256 cores are set to run at 400 MHz and LP-DDR3 frequency is configured at 1066 MHz (i.e ∼8.5 GB/s peak per DDR). Note that MPPA disposes of two DDRs (North and South) and the current OpenCL runtime only uses one DDR and exposes 1 GB of available global device memory. Different cavity sizes, varying from 32 to 256 are used in our tests, with some exceptions. Problem sizes larger than 160 can not be run on MPPA due to 1 GB of device memory, and ones smaller than 128 crashed on MIC by unknown reasons. Local work-group size in OPAL OpenCL is always set to $32 \times 1 \times 1$, as it delivers best performance on the four experimental architectures, in most of the cases.

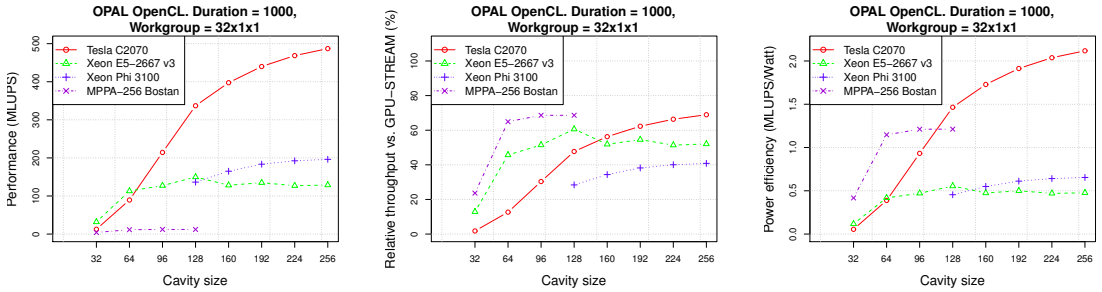
Fig. 11a shows performance in MLUPS on different architectures. These performance match (more or less) the baselines predicted in Sect. III, based on the GPU-STREAM bandwidth. Fig. 11b illustrates the relative percentage of LBM throughput compared to the GPU-STREAM. We observed that a highly strided-memory kernel like LBM on MPPA and GPU can reach up to 70% efficiency of GPU-STREAM, which accesses memory in a very contiguous layout. This efficiency on CPU and MIC is about 50% and 40%, respectively.

Interestingly, we noted that LBM bandwidth on MPPA is even more optimized than GPU and CPU on small problem sizes. The explanation of this high "reactivity" of MPPA would be on the memory system. GPU memory (GDDR5) was designed for high bandwidth (and also high latency) for processing huge amount of data by thousands of cores. In revenge, other processors like MPPA, Tilera or Epiphany, with less cores (< 300) and network-on-chip (NoC), privileges memory latency (DDR3) and determinism, rather than massive data throughput. Power consumption is also a good argument for these two architectures (see Fig. 11c), by being twice (on MPPA) and four times (on GPU) more efficient than on CPU and MIC.

B. Low-level OPAL asynchronous on MPPA

The MPPA Async library, at lower level, exposes both single and double DDR mode. In single-DDR mode, its peak bandwidth is the same as in the current OpenCL mode (8.5 GB/s). In double-DDR mode, the *LatticeEven* buffer is allocated on the North DDR and the *LatticeOdd* is on the South DDR. The effective throughput of the double-DDR mode can be considered as twice as one of the single-DDR mode, thus 2x performance is expected. We present here results of the OPAL kernel rewritten in our new pipelined algorithm on MPPA-256, called *OPAL_async*, in 3-depth and 4-depth pipeline. Each P-depth pipeline is also run using either local out-of-place update (two-lattice: S and S') or local in-place update (two-wall: only S). These tests are further run in both single and double DDR mode, in total of eight configurations finally. All these configurations are run and checked for correctness against the original OPAL code on GPU.

As one can observe in Fig. 12a, the OPAL_async algorithm outperforms the OpenCL version by more than 30% (from 12



(a) Performance in MLUPS

(b) Relative throughput to GPU-STREAM (%)

(c) Power efficiency (MLUPS/W)

Fig. 11: Original OPAL OpenCL on GPU, CPU, MIC and MPPA

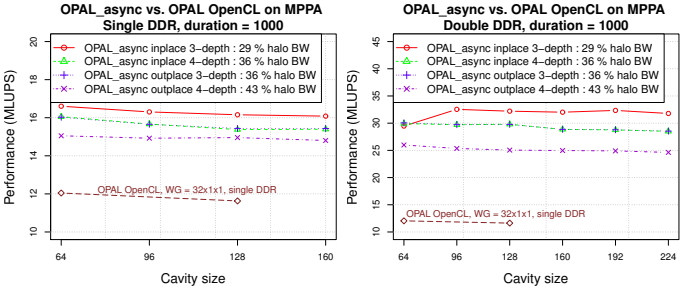
MLUPS to 16±1 MLUPS). We also see that the configuration with least halo bandwidth (in-place 3-depth, 29% halo BW) delivers highest MLUPS score. Other configurations, whether using in-place or out-place method, at 3-depth or 4-depth pipeline, always result a performance corresponding to their halo BW characteristic. The lower halo BW yields the better performance.

Moreover, using our *two-wall* method to perform in-place update helps optimizing local memory usage and increasing cube size, hence reduce the halo BW from 36% to 29% on a 3-depth pipeline, or from 43% to 36% on a 4-depth pipeline. This bandwidth-saving adds 5% performance gain (MLUPS) to the out-place version, which itself already outperforms the OpenCL code by 33%. Once again, this confirms that LBM (and other stencil codes) is memory-bound and its performance relies heavily on the hardware's sustained bandwidth. Finally, Fig. 12b shows the expected 2x performance speedup by using two DDRs compared to the single-DDR mode. The impact of halo BW on different cube sizes is also valid in this case.

These results show the adequacy of using GPU and other many-cores like MPPA for many kinds of stencil computations: CFD, image processing, geoscience etc. which are nowadays ones of the most data and computational intensive applications. The latest first-rank supercomputer, TaihuLight, built from Sunway processors [6], a NoC-based many-core architecture similar to others cited in this paper, opens a new trend for designing future Exascale/HPC systems. The newest generation of NVIDIA GPU (Pascal) is known to be 5 to 6 times more power-efficient than our in-hand Fermi C2070, making GPU still a big competitor to other architectures. The second Xeon Phi generation, Knight Landing, with a new architecture from the first one, high-bandwidth on-chip memory and large amount of off-chip memory, would also deliver interesting performance.

VIII. Conclusions

We introduce a decomposition approach for 3D stencil problems with generic equations for dynamically calculating copy position indexes, cube addressing, cube size and halo cells. These formulae work with or without using *ghost layer*. Based on this decomposition, our new pipelined 3D stencil



(a) Single-DDR
Both LatticeEven and LatticeOdd on North DDR

(b) Double-DDR
LatticeEven on North DDR, LatticeOdd on South DDR

Fig. 12: OPAL_async vs. OPAL OpenCL on MPPA

code outperforms the original OpenCL version by 33%, by overlapping computation and communication.

We observe that anticipating data requests by asynchronous memory transfers helps overcome the bandwidth limitation and turn a memory-bound kernel into compute-bound, by introducing enough pipeline depth to mask the global memory access latencies. However, local memory capacity hardly supports very deep pipelines. On another hand, reducing cube size to increase pipelining depth results in significant bandwidth consumption for halo copying. We then present comprehensive linear-programming equations for finding the best trade-off between all of these structuring parameters.

For optimizing local memory usage, we introduce *two-wall*, a new and simple LBM propagation method which performs in-place lattice update and reduces memory requirement almost by half. This method helps increasing local cube size and reducing halo transfer bandwidth. Thus, another 5% performance improvement is yielded, in addition to the previous 33% gain. Our results show that the DDR bandwidth is the bottleneck for optimizing stencil codes on a many-core processor like the Kalray MPPA-256. Besides, impact of halo bandwidth on small local memories was also identified as a governing factor of performance in our algorithm.

Finally, the *two-wall* method can be implemented directly in OpenCL code with only one global lattice grid, thanks to its parallel-ready nature. We will tackle this in a future work. Bringing these `async_work_group_copy_{2D|3D}` to

the next OpenCL specification is also in our perspective, as this enables exploiting in maximum other non-GPU many-core processors with their considerable local memory.

ACKNOWLEDGMENT

We would like to thank some members from the Kalray software team: Nicolas BRUNIE, Romaric JODIN, Pierre GUIRONNET DE MASSAS and Clément LEGER for interesting discussions and technical support during this work.

REFERENCES

[1] Sauro Succi. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.

[2] Nianzheng Cao, Shiyi Chen, Shi Jin, and Daniel Martinez. Physical symmetry and lattice symmetry in the lattice boltzmann method. *Physical Review E*, 55(1):R21, 1997.

[3] Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price. On the performance portability of structured grid codes on many-core computer architectures. In *Supercomputing*, pages 53–75. Springer, 2014.

[4] Christian Obrecht, Bernard Tourancheau, and Frédéric Kuznik. Performance evaluation of an opencl implementation of the lattice boltzmann method on the intel xeon phi. *Parallel Processing Letters*, 25(03):1541001, 2015.

[5] Benoıt Dupont de Dinechin, Renaud Ayrignac, P-E Beaucamps, Patrice Couvert, Benoıt Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[6] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[7] Keijo Mattila, Jari Hyväluoma, Jussi Timonen, and Tuomo Rossi. Comparison of implementations of the lattice-boltzmann method. *Computers & Mathematics with Applications*, 55(7):1514–1524, 2008.

[8] Tom Deakin and Simon McIntosh-Smith. Gpu-stream: Benchmarking the achievable memory bandwidth of graphics processing units.

[9] Federico Massaioli and Giorgio Amati. Achieving high performance in a lbm code using openmp. In *The Fourth European Workshop on OpenMP, Roma*, 2002.

[10] Thomas Pohl, Markus Kowarschik, Jens Wilke, Klaus Iglberger, and Ulrich Rüde. Optimization and profiling of the cache performance of parallel lattice boltzmann codes. *Parallel Processing Letters*, 13(04):549–560, 2003.

[11] Keijo Mattila, Jari Hyväluoma, Tuomo Rossi, Mats Aspnäs, and Jan Westerholm. An efficient swap algorithm for the lattice boltzmann method. *Computer Physics Communications*, 176(3):200–210, 2007.

[12] Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. Comparison of different propagation steps for lattice boltzmann methods. *Computers & Mathematics with Applications*, 65(6):924–935, 2013.

[13] DV Patil, KN Lakshmisha, and B Rogg. Lattice boltzmann simulation of lid-driven flow in deep cavities. *Computers & fluids*, 35(10):1116–1125, 2006.

[14] Dominique d'Humières. Multiple–relaxation–time lattice boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, 2002.

[15] Márcio Castro, Fabrice Dupros, Emilio Francesquini, Jean-François Méhautk, and Philippe OA Navaux. Energy efficient seismic wave propagation simulation on a low-power manycore processor. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 57–64. IEEE, 2014.

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

[17] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[18] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[19] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[20] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[21] Juan Carlos Juega, Sven Verdoolaege, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. *Patterns for parallel programming on GPUs*, chapter Evaluation of State-of-the-Art Parallelizing Compilers Generating CUDA Code for Heterogeneous CPU/GPU Computing. Saxe-Coburg Publications, Stirling, UK, 2014.

[22] Stephen W Poole, Oscar Hernandez, Jeffery A Kuehn, Galen M Shipman, Anthony Curtis, and Karl Feind. Openshmem-toward a unified rma model. In *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer, 2011.

[23] Jarek Nieplocha and Bryan Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler runtime systems. In *Parallel and Distributed Processing*, pages 533–546. Springer, 1999.

[24] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[25] Minh Quan HO, Bernard TOURANCHEAU, and Christian OBRECHT. Mpi communication on mppa many-core noc: design, modeling and performance issues. *Parallel Computing: On the Road to Exascale*, 27:113, 2016.

[26] David Richie, James Ross, Song Park, and Dale Shires. Threaded mpi programming model for the epiphany risc array processor. *Journal of Computational Science*, 9:94–100, 2015.

[27] Khronos OpenCL Working Group et al. The opencl specification, version 1.2, 15 november 2011. *Cited on pages*, 18(7):30.

[28] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[29] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[30] David S Wise. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. In *European Conference on Parallel Processing*, pages 774–783. Springer, 2000.

[31] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul HJ Kelly. Is morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience*, 18(11):1509–1539, 2006.