HAL
open science

Improving 3D Lattice Boltzmann Method stencil with asynchronous transfers on many-core processors

Minh-Quan Ho, Christian Obrecht, Bernard Tourancheau, Benoît Dupont de Dinechin, Julien Hascoet

To cite this version:

Minh-Quan Ho, Christian Obrecht, Bernard Tourancheau, Benoît Dupont de Dinechin, Julien Hascoet. Improving 3D Lattice Boltzmann Method stencil with asynchronous transfers on many-core processors. 36th IEEE International Performance Computing and Communications Conference (IPCCC 2017), Dec 2017, San Diego, United States. hal-01652614

HAL Id: hal-01652614

https://hal.univ-grenoble-alpes.fr/hal-01652614v1

Submitted on 30 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

This is an accepted single-column version of a paper at the 36th IEEE International Performance Computing and Communications Conference (IPCCC 2017) December 10th-12th 2017, San Diego, California, USA.

URL: http://ipccc.org/ipccc2017/main.php

Access to the published version may require subscription.

©2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Improving 3D Lattice Boltzmann Method stencil with asynchronous transfers on many-core processors

Minh Quan Ho*‡, Christian Obrecht†, Bernard Tourancheau*,
Benoît Dupont de Dinechin‡ and Julien Hascoet‡
*CNRS, LIG UMR 5217, Grenoble Alps University, F-38058 Grenoble, France.
†Univ Lyon, CNRS, INSA-Lyon, Université Claude Bernard Lyon 1, CETHIL UMR5008, F-69621 Villeurbanne, France.
‡Kalray S.A., F-38330 Montbonnot, France.

Abstract

CPU-based many-core processors present an alternative to multicore CPU and GPU processors. In particular, the 93-Petaflops Sunway supercomputer, built from clustered 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 endeavor for optimizing one of the most data-intensive kind of stencil computations, namely the three-dimensional applications of the lattice Boltzmann method (LBM). We propose optimizations on many-cores processors by using local memory and asynchronous software-prefetching on a representative 3D LBM solver as an example. We achieve 33 % performance gain on the Kalray MPPA-256 many-core processor by actively streaming data from/to local memory, compared to the "passive" OpenCL programming model.

1. Introduction

Since the last decade, the lattice Boltzmann method (LBM) has become widely used in computational fluid dynamics for incompressible and weakly compressible flows, see e.g. [1]. An LBM model is characterized by its stencil type, denoted DdQq, where d is the number of space dimensions (one, two or three) and q is the number of *particle distribution functions* (PDFs), see e.g. [2]. Physically, an LBM time step on a lattice node consists of a *collision* and a *propagation* step (also known as *streaming* step). The collision applies a pre-defined physical model on the lattice distribution vector of q values. The propagation then updates these new distribution values to the node itself and $q - 1$ of its neighboring nodes. The most used stencil types are D2Q5, D2Q9 and D3Q19 (see Fig. 1), or D3Q27.

From a programming point of view, LBM kernels are easy to implement and well-suited for parallelization on recent multi-/many-core platforms. However, lattice Boltzmann methods are known for their low arithmetic intensity and particularly high memory bandwidth requirement. Taking the example of a basic LBM solver, depending on collision operator, between 200 and 400 floating-point operations are performed on a lattice node per time step. Most D3Q19 LBM implementations require storing all the 19 distribution values for each lattice node. A lattice domain $L \times L \times L$ contains $19 \times L^3$ single- or double-precision floating-point numbers. Updating this lattice grid in a single time-step requires $19 \times 2 \times L^3$ load/store memory operations for less than $400 \times L^3$ arithmetic operations. Thus, simulating the whole lattice domain through T time-steps will generate a huge amount of data movement of $19 \times 2 \times L^3 \times T$ floating-point numbers

for $400 \times L^3 \times T$ floating-point operations. While recent architectures gain computing performance by increasing the clock speed and multiplying the number of cores, evolution of memory systems still cannot fetch enough data to keep cores busy. The dataset cannot always fit in caches and must be stored in the main (even remote) memory with much higher latency. The low arithmetic intensity of stencil kernels like LBM is thus the limit of performance, as well as their poor data-locality which reduces significantly the cache-reuse ratio. Previous studies by [3] and [4] show that LBM implementations are memory-bound and hardly obtain good performance on CPU or Xeon Phi processors. GPU-based accelerators, thanks to their graphics-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 neighboring dependencies).

Although other clustered many-core processors have much less global memory bandwidth than GPUs do, they embed significant amount of fast local memory, see [5] and [6], and provide more predictability in both computing time and data transfer. This enables using explicit and efficient user buffers for elaborate optimizations, such as software prefetching and streaming. This motivates our approach in developing a pipelined 3D LBM algorithm on the Kalray MPPA processor, based on local memory exploitation and asynchronous communications. Our algorithm is described in every detail and can be used on similar many-core architectures.

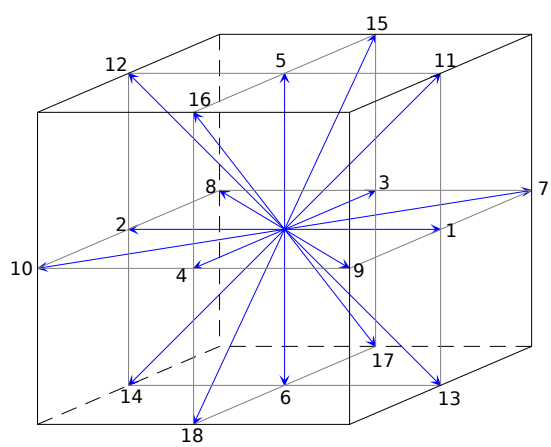

Figure 1. LBM D3Q19 stencil.

Our key contributions are as follows:

1) Introduction of a new parallel algorithm for decomposing and streaming 3D stencil domains on local-memory-centric clustered many-core processors, by user-buffers and asynchronous software-prefetching to build a *pipelined 3D stencil* kernel. The proposed approach is implemented from the LBM compute kernel of OPAL [4] and delivers 33% performance gain compared to its original OpenCL code on the Kalray

MPPA-256 Bostan many-core processor.

2) This work provides fundamental responses and methods to further domain-decomposition algorithms on clustered many-core processors (2D/3D stencils, image processing). An API proposal is also given in designing user-friendly 2D/3D asynchronous copy on DMA-enabled platforms.
3) Detailed description of the use of generic equations to calculate decomposition indexes dynamically, subdomain dimension and halo size, usable with or without *ghost layer* as in [7].

The remainder of this paper is structured as follows. Section 2 presents some related works that are relevant for our contributions. Section 3 introduces the main characteristics of the MPPA architecture and some low-level asynchronous transfer primitives required for building 3D stencils streaming algorithm. Section 4 presents an overview and technical details of the new LBM streaming algorithm using these asynchronous transfers. Experimental results are presented in Section 5, and we conclude in Section 6.

2. Related work

The straightforward method for implementing LBM is to use two instances of the lattice grid. Collision is carried out on data read from the first grid and propagation consists in writing the new distribution values to the second one. At the next time step, the two grids are swapped and the same procedure is repeated. *One-step two-lattice* method with collision and propagation fused in a same kernel was first introduced by Massaioli and Amati [8]. 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 substantial memory allocations with large domains.

Most existing LBM implementations on GPU employ the fused two-lattice approach as the easiest and most computationally efficient method. In particular, OpenCL Processor Array LBM (OPAL) from [4] implements a one-step two-lattice 3D LBM solver based on the D3Q19 stencil. OPAL is designed to be simple and portable on GPUs, accelerators and other OpenCL-enabled devices.

In the related work on porting a 3D seismic wave propagation on the MPPA processor, Castro et al. [9] developed a *2D-prefetching* algorithm for anticipating data transfers between global memory and local memory. The 3D domain is decomposed in small 2D slices. These slices are copied to the local memory such that transfers overlap with computations. The authors observed important waiting time for data arrival without identifying clearly the DDR bandwidth limitation of the MPPA. The impingement of halo slices on data throughput was not studied either.

Raase and Nordstrom [10] presented a 2D and 3D LBM implementation on Epiphany, a clustered many-core architecture very similar to MPPA. The LBM domain is distributed

on 16 cores with user local memory of 24KB per core. Subdomain distribution is done by static mapping of a 4x4 topology on the 16 cores. This 2D mapping is also used on the 3D problem where the third dimension of subdomains is assigned with one global domain dimension, giving rectangular parallelepiped subdomains. These choices allow simulating only very small problem sizes (e.g 12x35x12) and can not be scaled to large simulations. The authors declined using the DRAM memory to implement a streaming algorithm for large LBM domains. Neither memory bandwidth optimization nor possibility of using DMA to perform asynchronous transfer on Epiphany was discussed. In this work, we aim to provide a generic and scalable 3D decomposition with its cuboid distribution function and asynchronous subdomain streaming to reduce data transfer time. Such algorithm can be used as a reference point to implement further high performance LBM or stencil applications on clustered many-core architectures.

Nagar et al. [11] implemented a similar cube-based decomposition and distribution function which maps on CPU threads in the shared-memory context of large-memory multi-socket systems. Halo exchange between threads is done by writing directly into the memory zone of the respective cube owners, protected by mutual locks thanks to the CPU cache system. This mechanism cannot be directly used onto clustered architectures like MPPA as it requires either: (1) explicit inter-cluster communications; or (2) committing changes to the global memory then fetched by other clusters. Solution (1) is not relevant in our scope due to (small) local memories, numerous subdomains must be streamed continuously in the MPPA's compute clusters. Such streaming should be done preferably by a self-governing and synchronization-free algorithm. Thus, keeping data in the local memory and waiting for communication does not seem appropriate, not to mention the complexity of managing the inter-subdomain spatial data dependency. In this work, we choose to adapt solution (2), consisting in continuously committing changes of subdomains to the global memory and performing one global synchronization between clusters at each simulation time step.

To the best of our knowledge, there is no work yet on solving the challenges of simulating large LBM domains on clustered many-core architectures. However, large LBM domains cannot fit into on-chip memory and must be stored in the off-chip DDR memory, which has much higher latency. Hence, using DMA to perform asynchronous transfers between off-chip and on-chip memories becomes a key performance factor in order to mask the memory latency. This involves important code re-structuration, as well as new communication primitives and algorithms. In this work, all these problems are addressed and solved while keeping a clear abstraction level from the underlaying target hardware for the sake of genericity.

3. MPPA-256 Bostan

3.1. Architecture overview

The second generation of Kalray MPPA-256 processor named Bostan (see Fig. [12]) embeds 256 VLIW compute cores grouped into 16 compute clusters (CC) and 16 system cores in two unified I/O subsystems (IOS). The processor delivers peak performance of 634 GFlops in single precision and 317 GFlops in double precision within a consumption of 20 W.

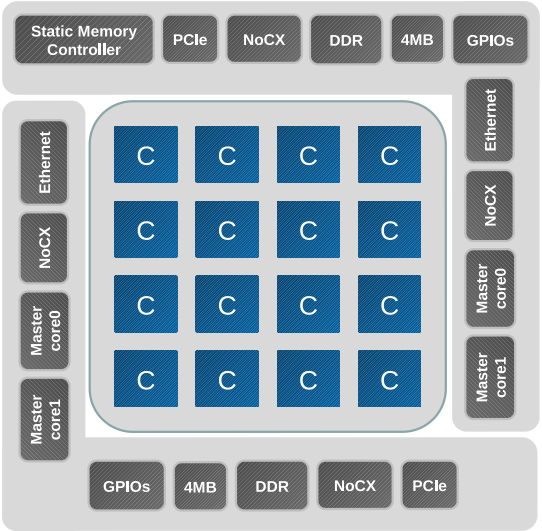

Figure 2. MPPA-256 processor overview (Source: Kalray).

Each compute cluster owns 2 MB of local memory (SMEM) shared between 16 user cores (Processing Elements-PEs) running at maximum frequency of 600 MHz. One system core, known as Resource Manager (RM), is reserved for running operating system and resources management. DMA engines of each compute cluster and I/O subsystem provide the system with high bandwidth and low latency transfers between SMEM-SMEM (symmetric inter-cluster) and SMEM-DDR memory (asymmetric cluster-IO).

3.2. Kalray OpenCL

The MPPA platform supports OpenCL 1.2 Data Parallel programming model. Such a model exposes each PE as a work-group with 64 KiB of `__local` memory. As defined in the OpenCL specification, these work-groups cannot share their `__local` memory to form somehow a large common memory. Moreover, we will see in further discussions that the global three-dimensional domain has to be cut down into smaller subdomains in order to fit in the local memory of clusters. The overhead of copying halo cells of small cuboids is more important than larger ones. In fact, subdomains should be sized to be cubic and

as large as possible. Given the maximum 64 KiB of `__local` memory per work-group, implementing a pipelined 3D LBM algorithm could not be efficient on the current Kalray OpenCL programming model.

On the other hand, in the POSIX programming model, each compute cluster appears as an independent 16-core CPU with 2 MB of shared memory. Thus large buffers can be allocated in the on-chip memories and this multi-core can be programmed using either OpenMP or Pthread multi-threading. This avoids useless data replication for each core and makes it possible to work on a larger common buffer inside the multi-core; therefore, increasing data reuse and reduces halo copy overhead. Such a model is relevant to our scope for efficient on-chip memory usage.

3.3. POSIX low-level 3D asynchronous API

In this section, we briefly present some essential primitives performing asynchronous 3D data transfers used to build our pipelined LBM algorithm onto the Kalray MPPA processor. As shown in Fig. 3, the `mppa_async_point3d_t` type describes copy-position and dimensions of the global and local 3D buffers. The subdomain is represented by $width \times height \times depth$ elements, whose each element has $size$ in bytes. We take an example to illustrate this specification design. In a common image processing decomposition, one may need to copy a 2D sub-image of 16×16 pixels to a larger local buffer, allocated at 18×18 pixels for instance. In this case, users must to explicitly deal with a local stride of two pixels between each data block, since the local buffer is sparse and the data should not be written contiguously to it. This is important when local buffers are declared as true multi-dimensional arrays in the C99 standard, a feature which particularly eases 2D and 3D stencil programming. With the convenient `mppa_async_point[2|3]d_t` data type (see Fig. 4), arbitrary positions and copy-block dimensions are automatically taken into account inside the 2D/3D *put* and *get* functions, facilitating subdomain copy and computation.

A structure `mppa_async_event_t` is also defined in the API to contain required information for performing an asynchronous transfer. In a *put/get* function, if the event structure is set, the function fills a pending transaction *event* and returns immediately (non-blocking paradigm). One can further come back and wait on this *event* by calling the `mppa_async_event_wait()` function for job completion. Otherwise, when the event structure is `NULL`, the function blocks and returns whether the buffer is ready to be reused (*put*) or the data are received (*get*).

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

/* 3D asynchronous transfer from remote to local*/
int mppa_async_memsget_block3d(
  void *local, const void *global,
  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);
```

Figure 3. A part of MPPA Async API for 3D transfer. Prototype of *get* and *put* are similar.

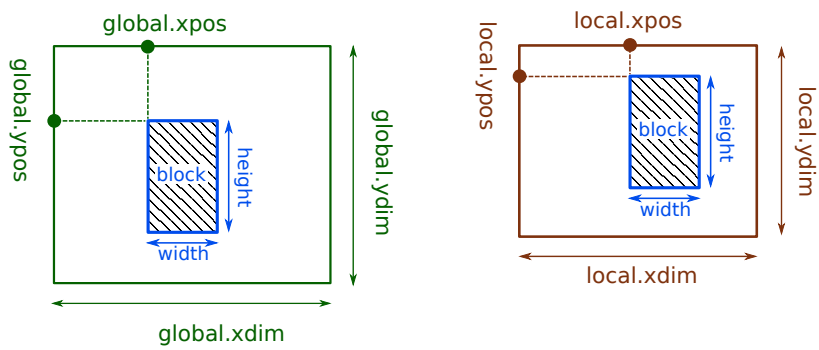

Figure 4. Illustration of `point2d_t` datatype for 2D copy. 3D copy is conceived by adding `depth` and `Z` fields.

4. Pipelined 3D LBM stencil on clustered many-core processors

4.1. Global algorithm

In the following, we take the D3Q19 LBM kernel from OPAL [4] as a reference point, from which we propose a generic 3D LBM streaming algorithm with domain decomposition, detailed index and halo size calculation in any configuration. The streaming method is used for updating the whole domain by one time step, then is repeated till the end of simulation duration. While we are focusing on optimizing LBM, our streaming method can also be generalized for other kinds of stencil codes, by adapting the compute kernel, some auxiliary settings (DdQq, halo size, number of time steps T), and a suitable set of asynchronous transfer primitives (2D/3D).

The first step consists in re-writing the LBM kernel of OPAL from OpenCL-C to a standard C99 code to run on CCs. Given the similarity between OpenCL-C and standard C99, the porting process did not raise much difficulty. 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 $L_x \times L_y \times L_z$ 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 subdomains (see Fig. 5), then copies and computes them one by one on the CC local memory. Each subdomain is defined as a $C_x \times C_y \times C_z$ cuboid. To avoid repetitions, we use the subscript

d as a symbol for the three Cartesian coordinates (x, y, z). Any variable or equation whose variables are subscripted by d should be interpreted as three variables or equations with x-/y-/z-subscripted terms respectively.

For the sake of simplicity, we assume that L_d and C_d are powers of two and define M_d the number of subdomains in each dimension ($M_d = L_d/C_d$). The total number of subdomains is the product of the number of subdomains in each dimension $M = M_x \times M_y \times M_z$. Besides, we denote the constant $F_d = C_d + h$ to be the extended subdomain size with halo layers (h) added[1]. Thus, updating a subdomain of $C_x \times C_y \times C_z$ nodes fetches an extended cuboid $F = F_x \times F_y \times F_z$ nodes to the local memory. This requirement is true for most cases (non-boundary subdomains - e.g subdomain 4 of Fig. 8). On boundary subdomains (e.g subdomains 0, 1, 2, 3 of Fig. 8), the extended cuboid should be adjusted by applying a *halo cutoff* to deal with solid nodes. A local subdomain slot must therefore be allocated for $F_x \times F_y \times F_z$ nodes to match any cases.

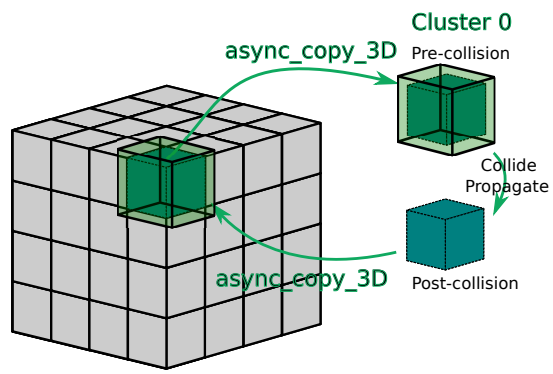

Figure 5. 3D LBM/stencil decomposition where *Main-block subdomain (green) is copied with its surrounding halo layers (if exists) and one extra subdomain (blue) is needed to store post-collision state.*

Algorithm 1 sums up the mono-cluster context where the compute cluster 0 (CC0) is updating M subdomains within an LBM time step. These subdomains are organized in a *macro-pipeline* using asynchronous 3D *put* and *get* functions 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 the pre-collision cuboid (S) from the first global lattice grid and one for storing the post-collision cuboid (S') that will be put in the second lattice grid. The pre-collision cuboid is allocated for $F_x \times F_y \times F_z$ nodes, while the post-collision cuboid only needs to store $C_x \times C_y \times C_z$ nodes. Fig. 5 only draws one global lattice grid for compactness, but it should be understood that the local post-collision cuboid will be put in the second grid. These two global grids are then swapped before starting the processing of the next time step.

Ideally, the algorithm should run on multiple compute clusters and exploit all processing cores (PEs) in each cluster (multi-cluster multi-PE). For instance, on MPPA, multi-threading within a compute cluster is enabled by spawning up to 15 threads, one per PE,

1. $h = 2$ with the D3Q19 stencil.

from the PE0 in the POSIX-pthread fashion (`create`, `join`). As there are 16 compute clusters available on MPPA, each CC is then responsible for $\frac{M}{16}$ subdomains. Note that depending on the value of M, there might be K trailing subdomains ($K \in [0..15]$). If $K > 0$, the algorithm must perform an extra step to copy, update and put back these K trailing subdomains by K compute clusters, while other clusters are waiting. A synchronization barrier at the end of each time step is needed between all CCs to avoid *data races* at the next time step. This procedure is then repeated as many times as the number of timesteps.

Algorithm 1 Explicit macro-pipeline of 3D stencil updates using double-buffering within a time step.

```
/* Prolog: get first subdomain */
prefetch_cube(0);
/* Pipeline */
for i in 0 .. M-1 do
    if i < M-1 then
        prefetch_cube(i+1);                    // get next cuboid
    end if
    wait_cube(i);                              // wait current cuboid
    compute_cube(i);                           // compute current cuboid
    put_cube(i);                               // put back to global
end for
/* Epilog: wait last put and barrier */
wait_cube(M-1);
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 (whose details are found in Fig. 6) provide better overlapping, but also require more local memory. Note that the time spent in `GET` and `PUT` is considered negligible (non-blocking) and transfers are executed in background. However, the time spent in `COMPUTE` depends on core speed, while the `WAIT` time depends on how fast the memory system is serving transfer requests and how they are hidden entirely or partially by the `COMPUTE` function.

In the next sub-sections, we propose methods to solve the following questions that immediately arise from Algorithm 1:

- How can we distribute fairly and exclusively all subdomains across CCs with their proper subdomain-indexing?
- Which subdomain size and pipeline depth should we choose to fit with the local memory size and to obtain the best trade-off?
- How to manage copy indexes and halo size of any subdomain, with or without using ghost layer?

	Prologue	m=0	1	2	3	4	5	6	7	Epilogue
		i=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		

Figure 6. 3-depth pipeline (triple-buffering) which allows 2-step distance between `GET` and `WAIT`, but only 1-step distance between `PUT` and `WAIT`, thus the `PUT` transfer will not be well overlapped (*m: index of subdomain to compute, i: index of local buffer slot;* `G = GET; P = PUT; W = WAIT; C = COMPUTE; WCP = {WAIT + COMPUTE + PUT}; WG = {WAIT + GET}`).

4.2. Subdomain distribution

Given a CC identified by $cc_{id} \in [0..15]$, its working subdomains is indexed by a one-dimensional range m as $cc_{id} \times \frac{M}{16} \leq m < (cc_{id} + 1) \times \frac{M}{16}$ (assume $K = 0$). Mapping *bijectively* this 1D domain (m) to a 3D one (m_x, m_y, m_z) for spatial cube indexing (see Fig. 5) was done by *space filling curves*, such as Morton or Hilbert curves, in [13]. These curves have been efficiently implemented by bit-interleaving in [14] or lookup-table in [15]. However, Morton, Hilbert and other curves are better suited for square or cubic grids where the number of elements in all dimensions is equal. In our 3D decomposition scheme, despite the fact that global lattice domain may be cubic ($L_x = L_y = L_z$), subdomains may be not ($C_x \neq C_y \neq C_z$) due to many reasons (see next section), thus these curves are not always suitable for subdomains, as M_d can be different.

In order to solve this problem, we implement a simple alternative bijective function $f : \mathbb{N} \rightarrow \mathbb{N}^3$ in Fig. 7. It follows the *3D-row-major* layout, which is also a space-filling curve, to index subdomains. Each conversion of the *3D-row-major* curve implemented by f takes less than 10 instructions and is as fast as Morton or Hilbert curves.

```c
void cuboid_index_1to3(int m,   /*input*/
  int* mx, int* my, int* mz)    /*outputs*/
{
  int z = (m / (Mx * My));
  int y = (m - (z * (Mx * My))) / Mx;
  int x = (m - (z * (Mx * My))) - (y * Mx);
  *mx = x; /*outputs*/
  *my = y; /*outputs*/
  *mz = z; /*outputs*/
}
```

Figure 7. 3D Row-major subdomain-indexing $f : \mathbb{N} \rightarrow \mathbb{N}^3$.

4.3. Local subdomain dimensions

In most of the cases, a cubic subdomain would be ideal for coding and optimizing. However, the local memory of clustered many-core processors is usually limited but plays an important role. On each MPPA's compute cluster, 2 MB local memory is quite small and should also host an embedded operating system, services and the user application binary.

A remaining space of about 1.5 MB is available for dynamic buffer allocations. Some auxiliary variables are also needed in LBM for macroscopic monitoring (velocity, density...). The maximal allocatable space for local pre-collision and post-collision cuboids is around 1.4 MB. Halo copy also consumes memory bandwidth. Hereafter, we refer to "*halo bandwidth*" HBW as the bandwidth lost in fetching halo layers. The HBW ratio is defined as the quotient of the number of halo cells by the total number of copied cells (main block and halo). On small subdomains, this ratio can be significant. For example, given a cubic subdomain whose main block size is $C_x \times C_x \times C_x$, its HBW ratio is:

$$\begin{gathered} g(C_x, C_x, C_x) = \frac{(C_x+2)^3 - C_x^3}{(C_x+2)^3} = \mathcal{O}(\frac{1}{C_x}) \\ \lim_{C_x \to \infty} g(C_x, C_x, C_x) = 0 \\ \textit{Example: } g(16,16,16) = \frac{18^3 - 16^3}{18^3} \approx 0.29 \end{gathered} \tag{1}$$

The best performance is achieved when the volume of the main block ($C_x \times C_y \times C_z$) is maximized and the HBW is minimized. Likewise, local storage should be reduced as much as possible. Let's assume single-precision floating-point representation, applying a D-depth pipeline for the two-local-cuboid method described above must fit into 1.4 MB of local memory and satisfy the linear-programming formulation below:

$$\begin{aligned} &\textbf{Find: } (D, C_x, C_y, C_z) \\ &\textbf{Maximize: } C_x \times C_y \times C_z \ \textit{(nodes updated per subdomain)} \\ &\textbf{Minimize: } F_x \times F_y \times F_z \ \textit{(per subdomain storage)} \\ &\textbf{Minimize: } \tfrac{(F_x \times F_y \times F_z) - (C_x \times C_y \times C_z)}{F_x \times F_y \times F_z} \ \textit{(HBW)} \\ &\textbf{Subject to:} \\ &\frac{D \times ((F_x \times F_y \times F_z) + (C_x \times C_y \times C_z)) \times 19 \times 4}{1024^2} \leq 1.4 \\ &F_d = C_d + 2 \ ; \ \ C_d \in \{2^n\} \ ; \ \ D, n \in \mathbb{N}^+ \end{aligned} \tag{2}$$

For instance, using $D \geq 3$ in order to have better overlapping than with a 2-depth pipeline, restricts to a very small search domain ($C_d \leq 128$) that can be resolved by running the branch-and-bound algorithm in a script. Solutions can either be (D, C_x, C_y, C_z) = $(3, 16, 8, 16)$ with 36% HBW ratio or (D, C_x, C_y, C_z) = $(4, 8, 8, 16)$ with 43% HBW ratio. A permutation of C_x, C_y, C_z also gives other satisfactory solutions, with the same HBW ratio. On the other hand, note that increasing pipeline depth is not relevant, because the higher D is, the smaller (C_x, C_y, C_z) will be, thus HBW will become unacceptable. Moreover, compute cores will switch between small subdomains more often. The accumulated waiting time will also be more important due to the exponential number of DMA requests and processing overhead of the DDR asynchronous services.

4.4. Local and remote copy-index management

In this section, we present generic analytic formulæ to process dynamically copy indexing, subdomain size computation and halo cutoff management depending on geometric position of the subdomain. Adding a *ghost layer* surrounding the computational domain is a common technique to simplify the implementation of the streaming step at boundary cells, see e.g. [7]. However, we choose not to use this approach in our work, mainly to minimize global memory allocations and avoid wasting bandwidth/storage in moving ghost cells.

However, in our 3D decomposition algorithm, this decision requires careful calculation of copy parameters from subdomain indexes. It is important to note that as the pre-collision cuboid S embeds two additional halo layers for each dimension (F_d), its computational space begins at $(1, 1, 1)$ and ends at $(F_x - 2, F_y - 2, F_z - 2)$ included. When fetching a non-boundary subdomain (main block + halo) from global memory to S, the arrival point of data at the local buffer is set to $(0, 0, 0)$, and the remote point is computed as the global beginning position of the subdomain minus one (back-off) in each dimension ($(m_d \times C_d) - 1$).

As *ghost layers* are not used in our implementation, a boundary subdomain can have up to three missing sides, depending on its location (see Fig. 8). Consequently, the halo layer of these missing sides needs to be pruned from the copied cuboid. The remote read-point and local write-point must also be adjusted as well. In order to generalize the solution, we introduce here three parameters associated respectively to these three adjustments: `halo_cutoff`, `remote_offset` and `local_offset`.

We present in the following, generic formulæ which determines copied positions and halo cutoffs of a given 3D cuboid subdomain (m_x, m_y, m_z), generalized from the 2D representation of Fig. 8.

$$\begin{aligned} \textit{const } A &= (A_x, A_y, A_z) = (0, 0, 0) \\ R &= (R_x, R_y, R_z) \\ &= (m_x \times C_x,\ m_y \times C_y,\ m_z \times C_z) \\ B_{ad} &= A_d + \textit{local_offset}(m_d, M_d) \\ B_{rd} &= R_d + \textit{remote_offset}(m_d, M_d) \\ S_d &= F_d + \textit{halo_cutoff}(m_d, M_d) \end{aligned} \tag{3}$$

The point $A = (A_x, A_y, A_z) = (0, 0, 0)$ is the start point of the local buffer. The point $R = (R_x, R_y, R_z) = (m_x \times C_x,\ m_y \times C_y,\ m_z \times C_z)$ is the start point of the remote subdomain, without its halo layers. The fetched cuboid S is sized at $S_d = F_d + \textit{halo_cutoff}(m_d, M_d)$. It is read from the remote position $B_{rd} = R_d + \textit{remote_offset}(m_d, M_d)$ and written to the local position $B_{ad} = A_d + \textit{local_offset}(m_d, M_d)$. The collision is performed on the main block of S and the result is then written to S' for the propagation step. However, managing

copied parameters of S' is simpler than on S. Since S' contains exactly the main block of the subdomain, updated data from a collision can be written to $(0, 0, 0)$, which is also the local copied position for sending to remote memory $R = (R_x, R_y, R_z)$. The parameters `halo_cutoff`, `remote_offset` and `local_offset` are implemented as macros with rules in Tab. 1.

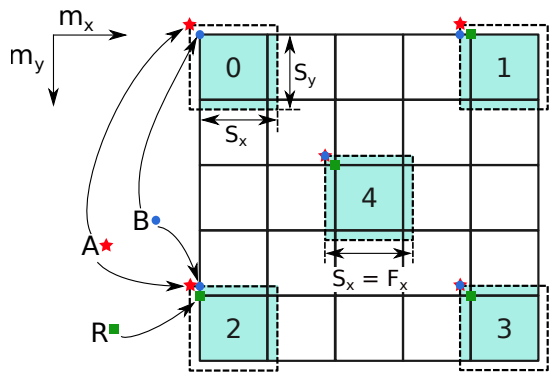

Figure 8. Local/Remote copied index in 2D (in lattice node) with *A: begin of the local buffer = (0,0); R: begin of the remote main block cuboid (without halo); B: begin of the copied cuboid (S), represented by:* B_a*: index of S on local memory (from A) and* B_r*: index of S on global memory (from R).*

TABLE 1. COPIED INDEX OFFSET AND HALO CUTOFF OF A SUBDOMAIN.

	local_offset (from point A)	**remote_offset** (from point R)	**halo_cutoff** (from F_d)
$m_d = 0$	1	0	-1
$0 < m_d$ && $m_d < M_d - 1$	0	-1	0
$m_d = M_d - 1$	0	-1	-1

These position computations can also be applied on other implementations which use *ghost layer*, by setting all `remote_offset` to -1, i.e. allowing to jump out of the computational domain, and all `local_offset`, `halo_cutoff` to zero, i.e. imposing to copy extended subdomain F_d to A_d, instead of copying S_d to B_{ad}.

5. Results and discussions

5.1. Pipelined 3D LBM stencil on MPPA

We implement the pipelined 3D LBM algorithm on the MPPA-Bostan platform using the POSIX programming model and asynchronous 3D primitives from the MPPA Asynchronous One-Sided library. 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 embeds two DDR interfaces (North and South) and the current OpenCL runtime only uses one DDR and exposes 1 GB of available global device memory, while the MPPA Asynchronous One-Sided library exposes both single and double DDR modes. Different cubic cavity sizes, varying from 64 to 224 are used in our tests, with some exceptions.

Problem sizes larger than 160 can not be run in OpenCL on MPPA due to the 1 GB device memory limit. Local work-group size in OPAL OpenCL is always set to $32 \times 1 \times 1$, as it delivers the best performance in most of the cases.

In single-DDR mode (POSIX and OpenCL), both *LatticeEven* and *LatticeOdd* are allocated on the North DDR. In double-DDR mode (POSIX-only), 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 $2\times$ performance is expected. We present here results of the OPAL kernel rewritten with our new POSIX pipelined algorithm on the MPPA-256, called *OPAL_async*, in 3-depth and 4-depth pipelines and following the local two-lattice method (S and S') on various cavity sizes. These tests are further run in both single- and double-DDR modes. All these runs are checked for correctness against the original OPAL code on GPU.

As one can notice in Fig. 9, the OPAL_async algorithm outperforms the OpenCL version by more than 30% on the single-DDR mode (from 12 MLUPS to 16 ± 1 MLUPS). We also see that the configuration with less HBW (3-depth, 36% HBW) delivers higher performance than the 4-depth configuration (43% HBW). While consuming memory bandwidth, halo cells are copied because of the read-dependency between neighbors. This does not contribute to the final performance. Fig. 9 shows that the less memory bandwidth halo cells take up, the more performance we obtain. This leads to think that the HBW of 2D/3D stencil computations aimed to reach Exascale, like weather forecast, ocean simulation and CFD, should be lessened on future clustered many-core processors. For this to happen, these many-core chips should embed bigger local memory on each compute unit to tear down the useless part of halo exchange due to domain decomposition. Finally, Fig. 9 also shows the expected $2\times$ performance speedup by using two DDRs compared to the single-DDR mode.

5.2. Performance extrapolation

For a better understanding of the benefit of our streaming algorithm, we modified the OPAL_async code to be able to work with arbitrary values of pipeline-depth. Different pipeline depths were then tried out (1, 2, 4, 6, 8) to see if increasing the number of asynchronous buffers can improve the performance. The block size is thus reduced to $8 \times 8 \times 8$ so that up to eight subdomains can be stored in the local memory. Moreover, instead of using all the 16 compute clusters, we now vary this number of clusters and set the domain size to 128^3 to study the strong scalability of the algorithm. We consider using only the double-DDR mode this time to obtain the best performance.

In Fig. 10, as expected, the 1-depth code (blue line) is slower than other version with communication-computation overlapping. However, we obtain exactly the same performance as the double-buffering case when using more than two buffers (4, 6, 8). The

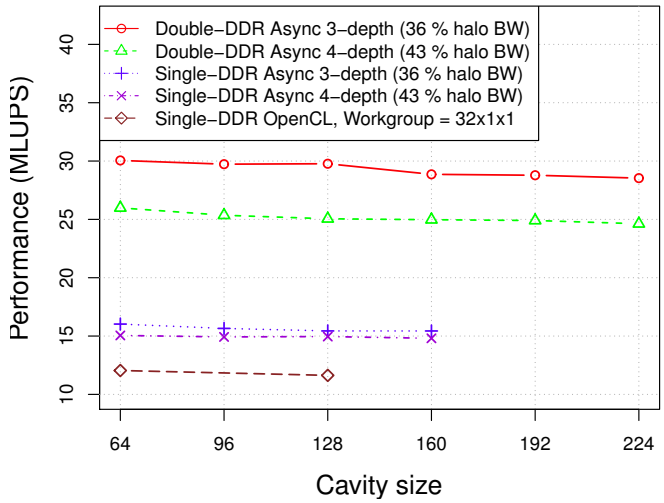

Figure 9. OPAL_async vs. OPAL OpenCL on MPPA for duration = 1000 steps.

performance line scales from 1 cluster to 8 clusters, then reaches almost a stable value of between 20-22 MLUPS from 8 clusters to 16 clusters. To explain this, we added the sustained throughput of 3D transfer (red line) from the Kalray unit test dedicated to 3D asynchronous copy. This test only does some ping-pong copies to the DDR and does not perform any calculation (Arithmetic Intensity (AI) = 0 flops/byte). We observe that the native 3D copy reaches the maximum throughput with as few as four clusters (6GB/s), then remains the same for higher numbers of clusters (which is the same trend as the performance of OPAL_async.). Four clusters are thus enough to saturate the DDR bandwidth. Unlike the 3D unit test, our LBM code performs real computation on the copied data. Its AI is about $350/(2 * 19 * 4) = 2.3$ flops/byte, which means that each CC spends more time working on a 3D data block. This explains in Fig. 10 the MLUPS performance which reaches its upper bound for 8 clusters, instead of 4 clusters of the 3D unit test.

Another precise way to interpret the performance of 20-22 MLUPS is to apply the performance estimation formula presented by McIntosh-Smith et al. [3]:

$$P = \frac{B \times 10^9}{19 \times 2 \times 4 \times 10^6} \, (MLUPS) \tag{4}$$

in which B is the effective memory bandwidth in GB/s. In order to take into account the additional cost of halo copy in our decomposition algorithm, we multiply P by $(1 - HBW)$, the effective part of bandwidth (main block) which generates the real performance:

$$P_h = \frac{6.0 \times 10^9}{19 \times 2 \times 4 \times 10^6} \times \frac{8^3}{10^3} = 20.2 \, MLUPS \tag{5}$$

This estimation P_h, shows that there is seemingly a little performance gain to perform asynchronous transfers on clustered many-core processors (here MPPA as an example)

as for today. This is not because the streaming algorithm is not good, but because the overlapping gain time is too small compared to the lengthy waiting time for data due to the DDR3 bottleneck. This also demonstrates the memory-bound property of general stencil computations and leads to think that newer memory technologies, such as DDR4 and others, will be a performance boost on these architectures.

Notice that the scale-down of the 3D throughput versus the peak 17GB/s of two DDRs is caused by the fact that strided copies (2D/3D) must read data from a lot of different DDR memory banks. Furthermore, these copies can unavoidably suffer bad alignments due to the access pattern of application ($Q = 19$ floats), thus bear an efficiency factor of 3D transfer compared to the linear copy. For instance, on the current MPPA Bostan platform, if the linear transfer factor is normalized at 1, the 3D factor lies often in between 0.35 and 0.42, depending on the copy layout (size of each contiguous block, alignment of strides and dataset).

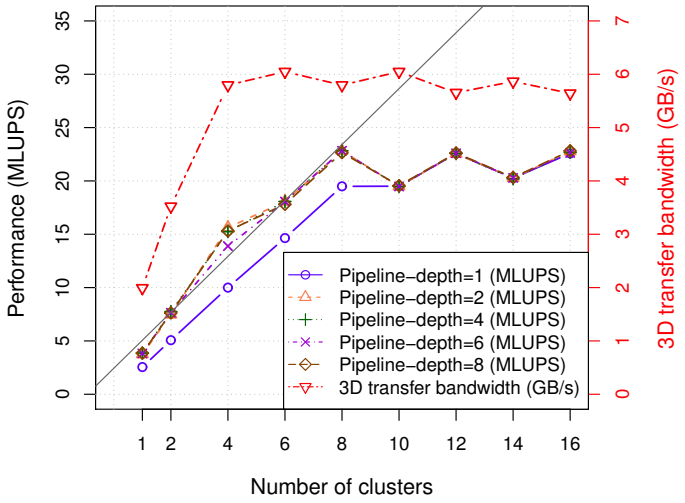

Figure 10. Performance extrapolation of OPAL_async on $8 \times 8 \times 8$ subdomains with the first eight clusters correlation represented by a gray line for 1000 timesteps and cavity size 128.

A correlation, computed by the `lm` function in `R`, from 1 to 8 clusters gives the performance expectation of our streaming algorithm if we were not bounded by the memory bandwidth (gray line). These results confirm that our pipelined LBM algorithm is strongly scalable, but is quickly memory-bound on MPPA and that its performance heavily depends on the hardware memory bandwidth. Our results also show that the imbalance between computing power and data throughput is one of the largest drawbacks of actual clustered many-core processors, and demonstrate the interest of future high-bandwidth memory technologies.

6. Conclusions

We introduced a decomposition approach for generic 3D stencil problems with formulations for calculating dynamically copied position indexes, subdomain addresses, subdomain size and halo cells. These analytic results work with or without using *ghost layers* and are also usable for 2D problems. Based on this decomposition, our new pipelined 3D LBM code outperforms the original OpenCL version by 33 %, by overlapping computation and communication.

We expected that anticipating data requests by asynchronous memory transfers would improve effective throughput and that we could overcome the memory bound of the studied LBM kernel, by introducing enough pipeline depth to hide the global memory access latency. In practice, performance results are still bound by memory bandwidth and increasing the number of buffers (pipeline depth) does not improve performance, as the DDR3 memory is already fully loaded. Moreover, reducing subdomain size to increase pipeline depth induces significant bandwidth consumption for halo copy. Furthermore, the impact of HBW on small local memories was also identified as a governing factor of performance in our algorithm. We found out that the best strategy is to have cubic subdomains as large as possible and that the double-buffering scheme is enough on the current generation of MPPA processor. We furthermore presented comprehensive linear-programming equations which give the best trade-off between these structuring parameters.

In the future, we plan to study a new LBM propagation method which performs in-place lattice update (*one-step one-lattice*). Such a method will reduce by half the local memory requirement, thus increase the subdomain size, trim down halo bandwidth and improve performance. Porting `async_work_group_copy_{2D|3D}` primitives to the next OpenCL specification is also under consideration, as this would considerably improve the exploitation of local memory on clustered many-core processors.

Acknowledgments

The authors would like to thank some members from the Kalray software team: Nicolas Brunie, Romarik Jodin, Pierre Guironnet de Massas and Clément Leger for interesting discussions and technical support during this work.

References

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

[2] N. Cao, S. Chen, S. Jin, and D. Martinez, "Physical symmetry and lattice symmetry in the lattice boltzmann method," *Physical Review E*, vol. 55, no. 1, p. R21, 1997.

[3] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the performance portability of structured grid codes on many-core computer architectures," in *Supercomputing*. Springer, 2014, pp. 53–75.

[4] C. Obrecht, B. Tourancheau, and F. Kuznik, "Performance evaluation of an opencl implementation of the lattice boltzmann method on the intel xeon phi," *Parallel Processing Letters*, vol. 25, no. 03, p. 1541001, 2015.

[5] B. Dupont de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

[6] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao *et al.*, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[7] K. Mattila, J. Hyväluoma, J. Timonen, and T. Rossi, "Comparison of implementations of the lattice-boltzmann method," *Computers & Mathematics with Applications*, vol. 55, no. 7, pp. 1514–1524, 2008.

[8] F. Massaioli and G. Amati, "Achieving high performance in a lbm code using openmp," in *The Fourth European Workshop on OpenMP, Roma*, 2002.

[9] M. Castro, F. Dupros, E. Francesquini, J.-F. Méhaut, and P. O. 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*. IEEE, 2014, pp. 57–64.

[10] S. Raase and T. Nordström, "On the use of a many-core processor for computational fluid dynamics simulations," *Procedia Computer Science*, vol. 51, pp. 1403–1412, 2015.

[11] P. Nagar, F. Song, L. Zhu, and L. Lin, "Lbm-ib: A parallel library to solve 3d fluid-structure interaction problems on manycore systems," in *Parallel Processing (ICPP), 2015 44th International Conference on*. IEEE, 2015, pp. 51–60.

[12] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*. IEEE, 2015, pp. 220–229.

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

[14] D. S. Wise, "Ahnentafel indexing into morton-ordered arrays, or matrix locality for free," in *European Conference on Parallel Processing*. Springer, 2000, pp. 774–783.

[15] J. Thiyagalingam, O. Beckmann, and P. H. Kelly, "Is morton layout competitive for large two-dimensional arrays yet?" *Concurrency and Computation: Practice and Experience*, vol. 18, no. 11, pp. 1509–1539, 2006.