Optimization of data transfer on many-core processors, applied to dense linear algebra and stencil computations

Minh Quan Ho

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Minh Quan HO

Thèse dirigée par **Bernard TOURANCHEAU**, Professeur, UGA

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Optimisation de transfert de données pour les processeurs pluri-coeurs, appliqué à l'algèbre linéaire et aux calculs sur stencils

Optimization of data transfer on many-core processors, applied to dense linear algebra and stencil computations

Thèse soutenue publiquement le **5 juillet 2018**,
devant le jury composé de :

Monsieur BERNARD TOURANCHEAU
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Directeur de thèse
Monsieur JOËL FALCOU
MAITRE DE CONFERENCES, UNIVERSITE PARIS-SUD, Rapporteur
Monsieur FRANCISCO DANIEL IGUAL PENA
ASSISTANT PROFESSEUR, UNIV. COMPLUTENSE DE MADRID - ESPAGNE, Rapporteur
Monsieur CHRISTIAN OBRECHT
MAITRE DE CONFERENCES, INSA LYON, Examinateur
Monsieur BENOÎT DUPONT DE DINECHIN
DIRECTEUR DE LA TECHNOLOGIE, KALRAY S.A. - MONTBONNOT-SAINT-MARTIN, Examinateur
Monsieur RAPHAËL COUTURIER
PROFESSEUR, UNIVERSITE BOURGOGNE FRANCHE-COMTE, Président

Abstract

Upcoming Exascale target in High Performance Computing (HPC) and disruptive achievements in artificial intelligence give emergence of alternative non-conventional many-core architectures, with energy efficiency typical of embedded systems, and providing the same software ecosystem as classic HPC platforms. A key enabler of energy-efficient computing on many-core architectures is the exploitation of data locality, specifically the use of scratchpad memories in combination with DMA engines in order to overlap computation and communication. Such software paradigm raises considerable programming challenges to both the vendor and the application developer. In this thesis, we tackle the memory transfer and performance issues, as well as the programming challenges of memory- and compute-intensive HPC applications on the Kalray MPPA many-core architecture.

With the first memory-bound use-case of the lattice Boltzmann method (LBM), we provide generic and fundamental techniques for decomposing three-dimensional iterative stencil problems onto clustered many-core processors fitted with scratchpad memories and DMA engines. The developed DMA-based streaming and overlapping algorithm delivers 33% performance gain over the default cache-based implementation. High-dimensional stencil computation suffers serious I/O bottleneck and limited on-chip memory space. We developed a new in-place LBM propagation algorithm, which reduces by half the memory footprint and yields 1.5 times higher performance-per-byte efficiency than the state-of-the-art out-of-place algorithm.

On the compute-intensive side with dense linear algebra computations, we build a matrix multiplication benchmark based on exploitation of scratchpad memory and efficient asynchronous DMA communication. This program delivers 350 GFLOPS, or 86% of theoretical performance of the MPPA. These techniques are then extended to a DMA module of the BLIS framework, which allows us to instantiate an optimized and portable level-3 BLAS numerical library on any DMA-based architecture, in less than 100 lines of code. We achieve 75% peak performance on the MPPA processor with the matrix multiplication operation (GEMM) from the standard BLAS library, without having to write thousands of lines of laboriously optimized code for the same result.

Résumé

La prochaine cible de Exascale en calcul haute performance (High Performance Computing - HPC) et des récent accomplissements dans l'intelligence artificielle donnent l'émergence des architectures alternatives non conventionnelles, dont l'efficacité énergétique est typique des systèmes embarqués, tout en fournissant un écosystème de logiciel équivalent aux plateformes HPC classiques. Un facteur clé de performance de ces architectures à plusieurs cœurs est l'exploitation de la localité de données, en particulier l'utilisation de mémoire locale (scratchpad) en combinaison avec des circuits d'accès direct à la mémoire (Direct Memory Access - DMA) afin de chevaucher le calcul et la communication. Un tel paradigme soulève des défis de programmation considérables à la fois au fabricant et au développeur d'application. Dans cette thèse, nous abordons les problèmes de transfert et d'accès aux mémoires hiérarchiques, de performance de calcul, ainsi que les défis de programmation des applications HPC, sur l'architecture pluri-cœurs MPPA de Kalray.

Pour le premier cas d'application lié à la méthode de Boltzmann sur réseau (Lattice Boltzmann method - LBM), nous fournissons des techniques génériques et réponses fondamentales à la question de décomposition d'un domaine stencil itérative tridimensionnelle sur les processeurs clusterisés équipés de mémoires locales et de circuits DMA. Nous proposons un algorithme de streaming et de recouvrement basé sur DMA, délivrant 33% de gain de performance par rapport à l'implémentation basée sur la mémoire cache par défaut. Le calcul de stencil multi-dimensionnel souffre d'un goulot d'étranglement important sur les entrées/sorties de données et d'espace mémoire sur puce limitée. Nous avons développé un nouvel algorithme de propagation LBM sur-place (in-place). Il consiste à travailler sur une seule instance de données, au lieu de deux, réduisant de moitié l'empreinte mémoire et cède une efficacité de performance-par-octet 1.5 fois meilleure par rapport à l'algorithme traditionnel dans l'état de l'art.

Du côté du calcul intensif avec l'algèbre linéaire dense, nous construisons un benchmark de multiplication matricielle, basé sur l'exploitation de la mémoire locale et la communication DMA asynchrone. Ce programme atteint 350 GFLOPS, soit 86% de la performance théorique de MPPA. Ces techniques sont ensuite étendues à un module DMA générique du framework BLIS, ce qui nous permet d'instancier une bibliothèque BLAS3 (Basic Linear Algebra Subprograms) portable et optimisée sur n'importe quelle architecture basée sur DMA, en moins de 100 lignes de code. Nous atteignons une performance maximale de 75% du théorique sur le processeur MPPA avec l'opération de multiplication de matrices (GEMM) de BLAS, sans avoir à écrire des milliers de lignes de code laborieusement optimisé pour le même résultat.

Acknowledgements

First, I would like to express my respectful thanks to my advisors: Bernard Tourancheau for his experience, patience and methodological research guidance; Christian Obrecht for his expertise in numerical simulation, relevant questions and thorough correction of all my writings; Benoît Dinechin for his deep knowledge, motivation and lightful recommendations.

My special thanks are extended to the reviewers and examiners for their acceptation and their time to study my manuscript and my PhD defense.

I also express my great appreciation to my colleagues at Kalray and LIG: Julien Hascoet, Nicolas Brunie, Julien Lemaire, Romarik Jodin, Jérôme Reybert, Pierre Guironnet de Massas, Clément Leger, Michael Mercier, Van Toan Dao, Baptiste Jonglez, Elodie Morin, Henry-Joseph Audeoud, Pierre Brunisholz, Timothy Claeys and many others for their kindness, interesting discussions, technical support or coffee breaks and cookies during my thesis.

My particular thanks go to members of the SHPC group at the University of Texas at Austin: Field G. Van Zee, Robert A. Van de Geijn, Devangi Parikh, Tyler Michael Smith and Devin Matthews for their friendly contact and valuable discussions.

Some experiments presented in this manuscript would have not been possible without ample access to the PLAFRIM experimental testbed, being developed under the Inria PLAFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities.

I would also like to give special thanks to Jacques Sicard, a grandparent-friendship who helped me a lot in my studies and life, especially during the hard beginning of this thesis.

To my elder brother, our parents and family for their trust and continuous encouragement.

To my daughter Ivy.

And to my wife Linh for your love and givings.

Contents

List of Figures

List of Tables

Acronyms

BLAS Basic Linear Algebra Subprograms.

BLIS BLAS-like Library Instantiation Software.

CPU Central Processing Unit.

DLA Dense Linear Algebra.

GEMM General matrix multiplication.

GPU Graphical Processing Unit.

HPC High Performance Computing.

HPL High Performance Linpack.

KNL Knights Landing.

LAPACK Linear Algebra PACKage.

LBM Lattice Boltzmann method.

MPI Message Passing Interface.

MPPA Massively Parallel Processor Array.

NUMA Non-uniform Memory Access.

OpenCL Open Computing Language.

OpenMP Open Multi-Processing.

Pthreads POSIX threads.

ScaLAPACK Scalable LAPACK.

VLIW Very Long Instruction Word.

Introduction

This manuscript is the fruit of a three-year PhD enduring work on optimizing scientific applications on many-core processors. This thesis is founded by the CIFRE collaboration (French term of *Convention Industrielle de Formation par la Recherche* – Industrial convention of research-driven training, by the French Ministry of Research and Innovation) between the Kalray corporate and the Grenoble Informatics Laboratory (LIG) from the University of Grenoble Alps (UGA) and Centre for Energy and Thermal Sciences of Lyon (CETHIL) from the National Institute of Applied Sciences of Lyon (INSA Lyon), the National Center for Scientific Research (CNRS) and the University Claude Bernard Lyon 1.

The first four chapters in this manuscript introduce the current High Performance Computing (HPC) situation, the state-of-the-art and objectives of this thesis, and the target many-core platform. Contribution chapters in this manuscript are then presented as two main parts, organized in the thematic order and not in the chronological one. The first part presents approaches in optimizing data transfer and memory footprint of the three-dimensional Lattice Boltzmann method (LBM), which belongs to the memory-bound category of applications. The second part focuses on Dense Linear Algebra (DLA) operations and associated numerical libraries, belonging to the compute-bound class. Experimental results in this document are reported on the Kalray Massively Parallel Processor Array (MPPA) many-core architecture, as well as other latest mainstream computing platforms such as NVIDIA Pascal Graphical Processing Unit (GPU), Intel Xeon Haswell Non-uniform Memory Access (NUMA) Central Processing Unit (CPU) and Intel Xeon Phi Knights Landing (KNL) processor.

Some contribution chapters in this manuscript contain principal materials from published and submitted papers, authored or co-authored by the writer of this manuscript as well. These papers are:

1. Minh Quan Ho, Bernard Tourancheau, Christian Obrecht, Benoît Dupont de Dinechin, and Jérôme Reybert. MPI communication on MPPA many-core NoC: design,

modeling and performance issues. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans J. Peters, and Mark Sawyer, editors, *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, volume 27 of *Advances in Parallel Computing*, pages 113–122. IOS Press, 2015.

2. Julien Hascoët, Benoît Dupont de Dinechin, Pierre Guironnet de Massas, and Minh Quan Ho. Asynchronous one-sided communications and synchronizations for a clustered manycore processor. In *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia, ESTImedia 2017, Seoul, Republic of Korea, October 15 - 20, 2017*, pages 51–60, 2017.

3. Minh Quan Ho, Christian Obrecht, Bernard Tourancheau, Benoit Dupont de Dinechin, and Julien Hascoet. Improving 3D lattice Boltzmann method stencil with asynchronous transfers on many-core processors. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC) (IPCCC 2017)*, San Diego, USA, December 2017.

4. Minh Quan Ho, Christian Obrecht, and Bernard Tourancheau. New parallel inplace update algorithm for better memory usage in 3D lattice Boltzmann algorithm. In submission, 2017.

5. Minh Quan Ho, Benoit Dupont de Dinechin, Bernard Tourancheau, and Christian Obrecht. BLIS-RDMA: A portable and high performance level-3 BLAS for DMA-based many-core architectures. In submission, 2017.

Chapter 1

High Performance Computing: from Single-core to Many-core

Knowledge has a beginning, but no end.
– Geeta S. Iyengar.

1.1 Introduction

Micro-processor architectures have made a considerable evolution since the first days of computer science. Several decades ago, when the processor clock was of the same order as the main memory speed, about hundreds of kilohertz to several megahertz, performance concerns were mostly on the computation cycles, rather than optimizing memory transfers. With advances in semiconductors, the transistor number and computing performance has exponentially increased over the years. Moore's law (Fig. 1.1) is a model of the semiconductor evolution, which depicts the increasing transistor count that doubles approximatively every 24 months for a constant circuit price. This evolution results in a parallel increase in computing power. Despite the limit in transistor size, this law has been nevertheless sustained by processor manufacturers since the last decade, by increasing the clock frequency and integrating more cores into a same silicon die.

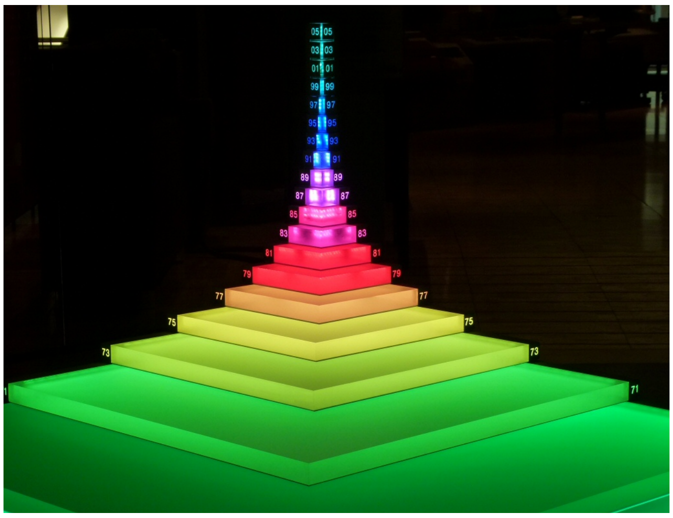


FIGURE 1.1: Moore's law illustration at the Heinz Nixdorf Museum.
Credit: Paul Townend.

High Performance Computing (HPC) has become an essential field to guide and to be guided by the moving computing architectures. Nowadays, HPC is almost used in any scientific domain, from molecular dynamics simulation, bio-informatics, medical drug discovery, to computational fluid dynamics (CFD), oil and gas, image and signal processing, ocean simulation and weather forecast, and recently huge explosion in computing workload for astrophysics radio telescopes (Square Kilometer Array – SKA) or deep learning. Nonetheless, performance walls in exploiting parallelism are, were, and will be the main challenge to computer scientists on the road of developing future HPC systems.

1.2 Performance walls

1.2.1 Frequency-wall

The higher the clock frequency operates, the higher the performance will be. However, there is a physical limit between the clock frequency and the processor design. In synchronous circuits, there exists a maximal frequency, under which the processor still manages to synchronize its components properly with the clock signal and to maintain an operating state. Beyond that frequency, the various distance between the clock source to the working components introduces a micro-dephasing that is in the same order of a single clock rate, results in erroneous synchronization and unstable circuits. That maximal clock frequency is conventionally fixed at 10 GHz. In this case, a single clock rate is roughly the time for the light to travel three centimeters, therefore the maximum possible distance difference on an electronic circuit.

Energy consumption and power dissipation is also an issue. High-frequency working system produces heat and must be accordingly cooled down. Current leakage is proportional to the voltage and increases power consumption. World records in over-clocking end-user computers are often limited at 8 GHz. To reach this level, the system is often cooled down by liquid nitrogen. An alternative to boost performance was, instead, doubling the core count and introducing execution concurrency, which opened the era of multi-core processors.

1.2.2 Energy-wall

In the last years, the energy-wall has emerged as the main limiting factor in designing supercomputers. Today, we are in the age of Petascale (10^{15} floating-point operations per second) within an energy budget of 20 MW. The first-ranked machine in the Top500 list (list of most powerful supercomputers in the world) delivers currently a power/energy ratio of about 10 GFLOPS/W. To reach Exascale (10^{18} floating-point operations per second), predicted to appear in the 2020s (Fig. 1.2), we need a system capable to deliver more than 50 GFLOPS/W, with a global power load of less than 20 MW.

To lower power consumption and to increase flops count, the only solution is to embed a large number of low-frequency cores into a same processor, sometimes up to hundreds of cores on a same die. Such a processor would require disruptive memory and interconnection technologies to feed the core cluster. This arises the question of how to write efficient codes on those architectures, which programming model to design to expose the

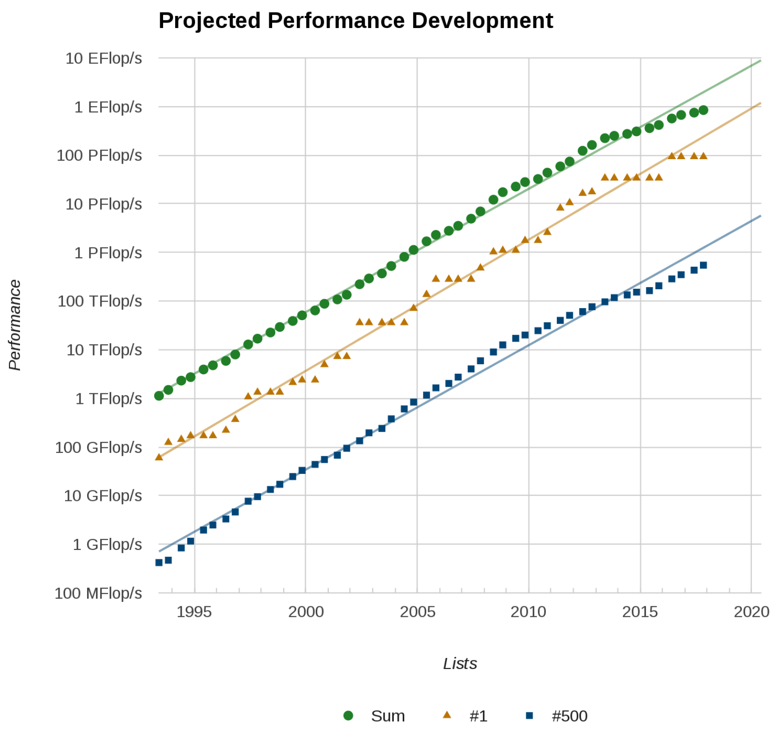


FIGURE 1.2: Projected performance of the Top500 list.
Source: https://www.top500.org.

massive parallelism, while ensuring ease of coding. It brings into light the challenges of many-core design and programming.

1.2.3 Memory-wall

There is an increasing gap between the computing performance and the memory speed. The computing performance slope is much steeper than the memory one. Since the overall performance amounts to the weakest part of the whole system, memory bandwidth turns into a bottleneck for most applications.

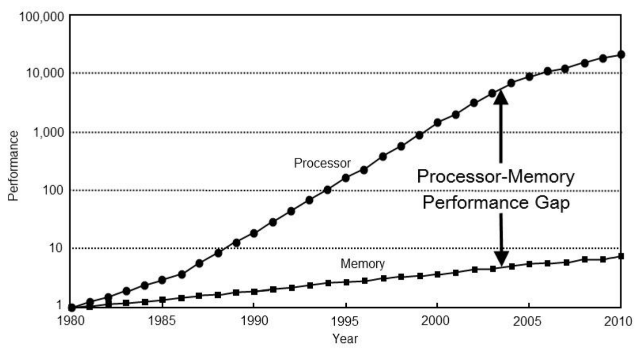


FIGURE 1.3: Processor-memory performance gap.
Source: *Computer Architecture: A Quantitative Approach* - David A. Patterson and John L. Hennessy.

In order to better understand the former issue, we will use the concept of arithmetic intensity and the roofline model [6]. The roofline model provides an easy way to predict performance of an application or more precisely its computation kernel. The concept of *arithmetic intensity* (I) (ops/byte or flops/byte for floating-point) is defined by the number of arithmetic operations performed on a given quantity of data loaded. Each implementation of a numerical method possesses a proper I related to its computation workload and data access pattern. Fig. 1.4 depicts the range of arithmetic intensity of several well-known computation kernels, such as sparse and dense linear algebra (DLA), lattice Boltzmann method (LBM) and fast Fourier transform (FFT).

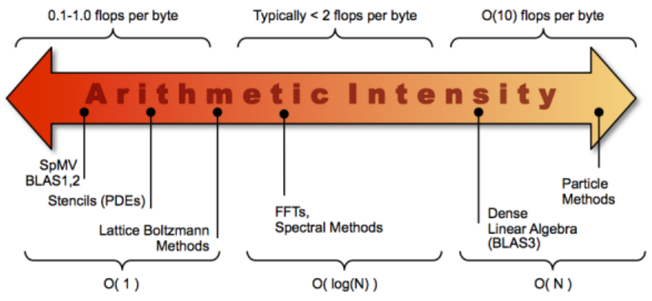


FIGURE 1.4: Arithmetic intensity.
Source: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/.

From the arithmetic intensity, one can assess the attainable performance P of an application on a computing system with a roofline figure (see Fig. 1.5). A roofline figure has an arithmetic intensity x-axis (flops/byte) and a performance y-axis (GFLOPS). Let π the peak theoretical performance of the processor, β the design memory bandwidth. With a given compiler and predefined optimization flags, the application produces a binary code that reaches a performance π'. Running a memory bandwidth benchmark on the system gives a sustained memory bandwidth β' ($\pi' < \pi$ and $\beta' < \beta$). The real performance P of the computation kernel is equal to $\min(\pi', \beta' \times I)$. When $P = \pi'$, the kernel is considered *compute-bound* since its performance does not rely on the memory but on the computing capacity of the processor, the applied compilation options, and the parallelization or vectorization method. This performance is a constant for each configuration variant (horizontal line in the roofline figure, see Fig. 1.5). When $P = \beta' \times I$, P follows a sloping line with a slope β', where comes the name of *roofline*. In this case, the kernel is considered *memory-bound*, since its performance relies on the memory bandwidth.

Each processor architecture has its own roof lines, and every computation kernel running on that processor has a performance bounded by those two lines. This performance model, despite being simple to characterize, allows doing a rapid comparison of computation kernels on an architecture (which ones are memory-bound, which ones are compute-bound), or comparing a specific kernel across multiple architectures (if it is memory-bound, so it should be better to use a higher memory-bandwidth system). Using the roofline model, a developer or an integration engineer can make appropriate choice to use/port their code on a specific platform, to adopt a more aggressive optimization method, or even to make architectural decisions.

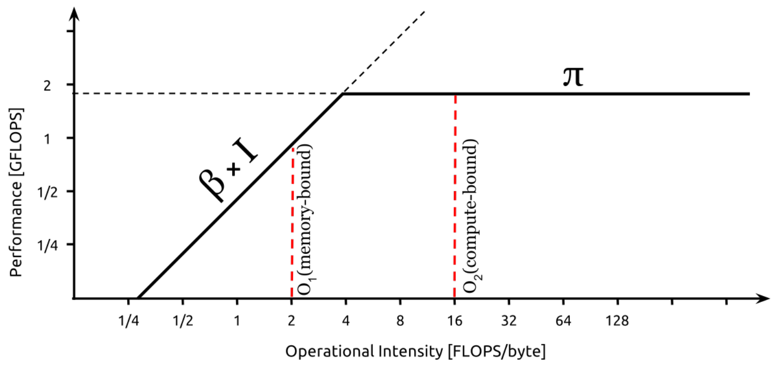


FIGURE 1.5: Roofline model.
Source: https://alchetron.com/Roofline-model.

Nevertheless, the roofline model has some weaknesses. First, the performance is assessed only based on the arithmetic intensity. Secondly, data locality and memory latency are not taken into account, yet those are often crucial to the real performance in a high concurrency context. Access pattern of applications impacts the hit or miss ratio in each cache level, thus can result in unpredictable behavior of the underlying memory system (latency, throughput), due to the coherency protocol, invalidation penalties and other side-effects.

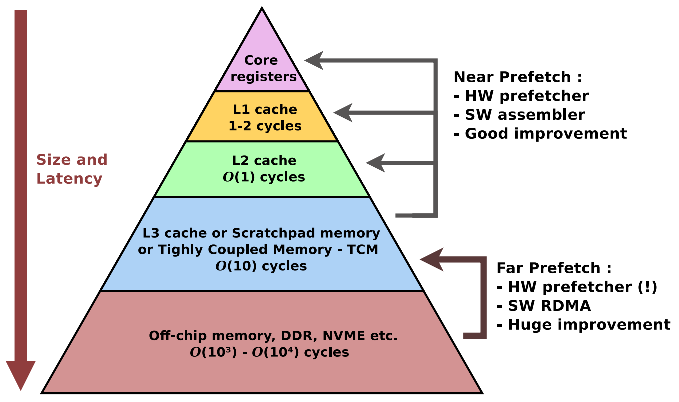


FIGURE 1.6: Different levels and latencies in memory hierarchy.

To overcome the memory latency and keep cores busy, computer architectures implement *data prefetching*. As can be seen in Fig. 1.6, near-core memory levels (caches or scratchpad, of the order of kilobyte and megabyte) can prefetch data either by implicit hardware mechanism or explicit built-in instructions. They were demonstrated to work efficiently on contemporary CPUs. However, when off-chip memories (DDR, NVMe etc.) come up with supplementary physical media (NoC, PCIe), performing implicit hardware prefetching is much harder and not relevant anymore, as the memory scope is becoming too large (order of giga-byte and tera-byte). Instead, a software approach based on asynchronous RDMA communication libraries, despite requiring additional programming efforts, can improve data locality and deliver satisfactory performance.

1.2.4 Software-wall

Exposing parallelism while keeping the programming model simple is a hard question. NUMA CPU processors can be programmed with Open Multi-Processing (OpenMP), POSIX threads (Pthreads), Intel Threading Building Blocks (TBB) or Intel Cilk. With the rise of heterogeneous systems, programming models must evolve as well. The OpenMP 3 for shared and NUMA architecture was revised to OpenMP 4 to support *target* devices. OpenACC [7], CUDA and Open Computing Language (OpenCL) [8] are also other programming models and APIs for accelerators. Programming language for distributed memory has longly been dominated by Message Passing Interface (MPI) [9] with incremental features leveraging the increasing node count: from two-sided communication in MPI 1 to one-sided communication since MPI 2, non-blocking collective operation since MPI 3 as well as a re-enforced one-sided specification.

Code and performance portability is also an issue due to the hardware diversity. To reduce the programming efforts for non-computer scientists, additional tools and meta-languages were introduced: domain specific languages (DSL) like Halide [10], high-level language and automatic framework (SYCL [11]), data-flow and DAG-based (Directed Acyclic Graph) analysis tools (StarPU [12]). These high-level tools allow users to express their processing kernels and get their code automatically generated, compiled and deployed on multiple computation units.

We believe that future high performance systems will be a combination of various computing platforms. Applications (and their sub-modules), upon their arithmetic intensity range, will be deployed and run on the most suitable platform. Such a system will be highly heterogeneous and non-uniform in terms of performance and memory bandwidth of each sub-platform, where developers sometimes need to hand-tune the code to obtain the best performance, especially on embedded and non-conventional hardware. Programming and running these all-in-one systems raises considerable complexity in design, scheduling, debugging, isolation and security.

Fault tolerance will also be a big concern on large-scale systems to resist against fail-stop failures, due to the fast decline of mean time between failure (MTBF) with the growing system size. Hardware-based fault tolerance mechanisms tend to be vendor-dependent at a certain degree. Without advances in research, these mechanisms may suffer significant overhead and would be difficult to optimize. Algorithm-Based Fault Tolerance (ABFT), since the last few years, has achieved important results by the research community [13, 14, 15, 16, 17]. However, relevance of following these approaches to the coming Exascale context is still unclear as of today.

1.3 Summary

In this chapter, we briefly present the high performance computing (HPC) and its main three performance walls: the frequency-wall, the energy-wall and the memory-wall. We also identify software challenges for the upcoming HPC applications, that we believe to be another obstacle to performance: the software-wall.

In the next chapter, we will introduce the first type of HPC application studied in this thesis: the lattice Boltzmann method, a stencil-based computation, known to be one of the most memory-bound applications.

Chapter 2

Lattice Boltzmann method (LBM)

If you can't explain it simply, you don't understand it well enough.

– Albert Einstein.

2.1 Background

2.1.1 Theory

Inspired from the lattice gas automata theory [18] and first introduced by McNamara and Zanetti [19], the lattice Boltzmann method has become widely used in computational fluid dynamics (CFD) as an alternative to the solving of Navier–Stokes equations. Belonging to the structured grid-based discrete method, the LBM is known for its advantages such as straightforward meshing, ability to model complex geometries, and most of all its inherent parallelism, well-suited to massively parallel computing architectures.

An LBM model is characterized by a 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) [20]. Particle distribution functions describe the interaction between a lattice node and its surrounding neighborhood. More precisely, q relates to the number of neighboring nodes that will be involved into interaction with the lattice node of interest. In most cases, the node itself is taken into account and the number of neighboring nodes equals $q - 1$. The most used stencil types for LBM are D2Q5 and D2Q9 for two-dimensional domains, or D3Q19 and D3Q27 for three-dimensional domains (see Fig. 2.1).

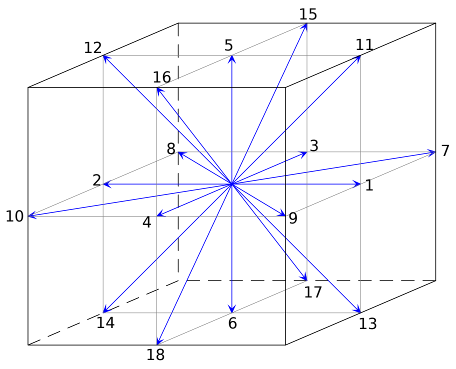


FIGURE 2.1: LBM D3Q19 stencil.

Three-dimensional LBM often operates on D3Q19 or D3Q27 stencils. The LBM spatial domain is represented by a grid of nodes, discretized with a mesh size δx. The simulation duration is discretized in constant time steps δt. The LBM updating rule for each node at each time step is defined by the following equation:

$$\big|f_i(\boldsymbol{x} + \delta t \boldsymbol{\xi}_i, t + \delta t)\big\rangle - \big|f_i(\boldsymbol{x}, t)\big\rangle = \boldsymbol{\Omega}\big|f_i(\boldsymbol{x}, t)\big\rangle \tag{2.1}$$

in which $\boldsymbol{\Omega}$ is a pre-defined *collision operator*. The collision operator implements the time evolution of particle distribution functions f_i ($i \in \{0, ..., q-1\}$) of a given node towards its nearest neighbors with respect to the $\boldsymbol{\xi}_i$ velocities. For better presentation in LBM codes, Eq. 2.1 is often split into two sub-steps:

$$\big|f_i^*(\boldsymbol{x}, t+\delta t)\big\rangle = \big|f_i(\boldsymbol{x}, t)\big\rangle + \boldsymbol{\Omega}\big|f_i(x, t)\big\rangle \tag{2.2}$$

$$|f_i(\boldsymbol{x}+\delta t\boldsymbol{\xi}_i, t+\delta t)\rangle = |f_i^*(\boldsymbol{x}, t+\delta t)\rangle \tag{2.3}$$

in which, Eq. 2.2 applies the $\boldsymbol{\Omega}$ operator to the current state $\big|f_i(\boldsymbol{x}, t)\big\rangle$ – a ket vector containing q PDFs of the lattice node. This reduces to local computations (also known as *collision* step), translated into floating-point arithmetic operations. Results of this sub-step are new PDFs of the next time step $\big|f_i^*(\boldsymbol{x}, t+\delta t)\big\rangle$, which temporarily remain within the local node. Eq. 2.3 then streams these PDFs into neighboring nodes (also known as *streaming* step), with the notation of spatial directions $\boldsymbol{x}+\delta t\boldsymbol{\xi}_i$. This sub-step is translated into memory load/store instructions.

2.1.2 Memory requirement

At each time step, the whole spatial domain must be updated before being able to start the next iteration. This spatio-temporal dependency of the LBM (shared by other stencil numerical schemes) compels developers, for the sake of code simplicity, to allocate two instances of the computational array, one as input of Eq. 2.2 (read-only) and one as output of Eq. 2.3 (write-only). This technique is usually known as *two-lattice* [21] (see the next section), whose the main drawback is the doubled memory consumption which significantly reduces the maximal reachable spatial resolution. It requires scientists to run their code on more machines with a larger aggregate memory space, thus resulting in larger cost and energy consumption.

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. Fig. 2.2 illustrates the memory-bound aspect of a D3Q19 LBM model.

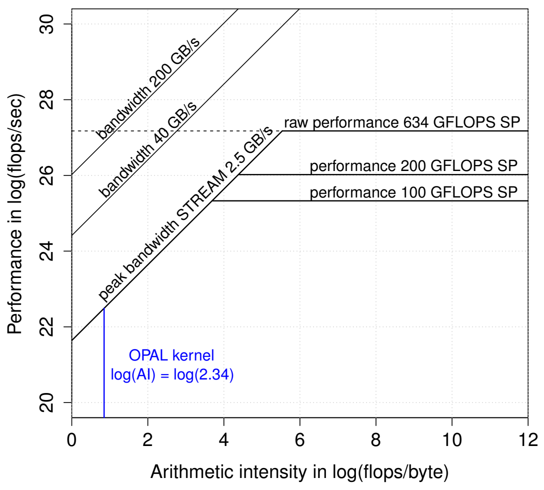


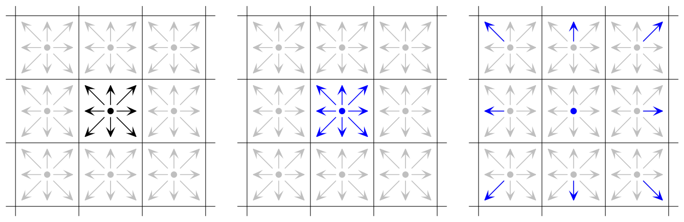
FIGURE 2.2: D3Q19 LBM applied on MPPA2 roofline model.

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 in [22] and [23] 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.

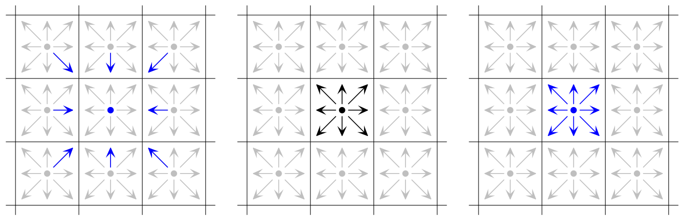
2.2 Propagation algorithms

2.2.1 One-step two-lattice (OT)

One-step two-lattice (also known as *two-lattice*) is the most employed algorithm in LBM implementations on massively parallel architectures. The collision and streaming steps are fused into one compute kernel, either in *pull* or *push* scheme (see Fig. 2.3). This kernel loops on all lattice nodes and updates the whole domain at each time step. The two lattice arrays (A and B) which are swapped at the end of each time step, differ from each other by their access type within the compute kernel: one is read-only and one is write-only. Using non-temporal streaming store to perform the write operation in the streaming step is thus a good reason to use this algorithm. But this feature is not widely available on all architectures, due to its hardware cost.



(A) *Push* propagation.



(B) *Pull* propagation.

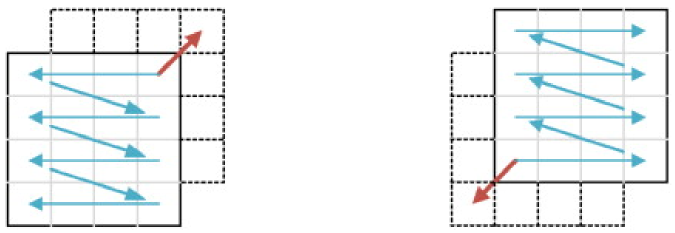
FIGURE 2.3: Propagation schemes of LBM in the D2Q9 sketch.

2.2.2 One-step one-lattice (OO)

Different approaches, known as *one-lattice* algorithms, were introduced to reduce the memory footprint by working on only one lattice array and to improve data locality of the LBM. Most of them operate elaborate exchange of PDFs between neighboring lattice nodes in the parallel execution context.

Pohl et al. [24] proposed *compressed-grid* (also known as *shift*, see Fig. 2.4) approach to reduce the memory requirement of the two-lattice algorithm. With the same objective, Mattila et al. [25] developed a *swap* algorithm that requires almost half of memory space compared to the two-lattice algorithm. Comparisons of these algorithms with varying lattice-indexing and data layouts were carried out in [26, 27]. These studies show equivalent computational efficiency of compressed-grid and swap algorithm compared to the two-lattice approach, while consuming less memory. However, these two approaches both require definite iteration order and complex index calculations for shifting the two lattice grids (*shift*) or swapping distribution values between neighbors (*swap*). These dependencies make it very hard to implement *shift* and *swap* algorithms on highly parallel GPUs and accelerators in offloading mode (CUDA, OpenCL). Today, they are implemented only as sequential CPU code inside a subdomain and are scaled up by using MPI for inter-domain halo exchange [27]. This configuration yields satisfactory weak-scaling but cannot enable strong-scaling, since execution of each shared-memory subdomain cannot be parallelized by either OpenMP or Pthreads.

Bailey et al. [28] presented the *AA-pattern* which overwrites read input PDFs by new collided data via two different kernels (even and odd time steps) (see Fig. 2.5). Geier and Schönherr [29] introduced *Esoteric twist* (shortened to *Esotwist*) as an improvement of



(A) Even step. (B) Odd step.

FIGURE 2.4: Compressed-grid (shift) propagation algorithm. Source: Wittmann et al. [27].

AA-pattern, by interacting only with neighboring nodes in positive xyz-direction. These two later algorithms work on one lattice array, are inherently asynchronous and thus are attractive for GPU and similar parallel architectures. However, they are more complex to implement than other algorithms mentioned above. Compressed-grid, AA-pattern and Esotwist need two kernels for even and odd time steps respectively. Esotwist can be implemented with one kernel, but requires imperatively the SoA (structure of arrays) storage layout to swap the Q pointers to their opposite direction after each collision, thus mostly only interesting for GPU architectures.

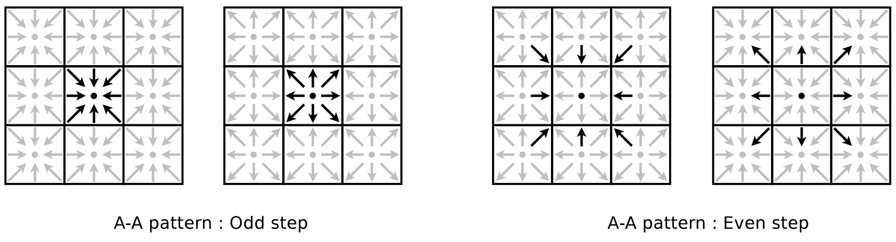


FIGURE 2.5: D2Q9 version of AA-pattern with two kernels at odd and even time steps. The odd step reads local PDFs in their opposite order, collides and stores back locally in natural order. The even step performs reads of PDFs from neighboring nodes (pull), collides and writes back (push) to the same place on these nodes, with opposite PDFs.

2.3 Summary

The lattice Boltzmann method is one of the most memory-bound applications among other iterative stencil-based methods, such as image processing and computer vision. Improving LBM performance lies on optimizing data-locality for better utilization of cache memories, as well as reducing the global memory footprint by inventing new implementation methods.

The fundamental challenge of any *one-lattice* algorithm is that memory accesses (read and write) must be performed carefully on the same lattice buffer to enforce the spatio-temporal dependency between nodes and time steps. More over, implementation often requires two versions of kernel code, adding more programming effort and reducing the maintainability of the application. On another hand, LBM boundary conditions on new physical models tend to be more and more elaborate. Typical LBM boundary conditions, such as simple bounce-back or interpolated bounce-back [30], imposes specific exchange rules of PDFs between adjacent nodes. Combining these conditions with existing *one-lattice* algorithms raises considerable complexity in implementation and validation, especially for 3D domains.

Other clustered many-core processors, despite a much lower global memory bandwidth with respect to GPUs, embed significant amount of fast local memories [31, 32]. They also 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, based on local memories and asynchronous DMA engines.

Chapter 3

Basic Linear Algebra Subprograms (BLAS)

Today, most software exists, not to solve a problem,
but to interface with other software.
– I. O. Angell.

3.1 Background

3.1.1 Introduction

Since its first release in the 1980s, the Basic Linear Algebra Subprograms (BLAS) [33, 34] has been widely used as the *de facto* foundation of high-level dense linear algebra libraries such as Linear Algebra PACKage (LAPACK) [35] and Scalable LAPACK (ScaLAPACK) [36], as well as in many inter-disciplinary scientific software. BLAS was designed to provide an unified and portable interface of numerical linear algebra operations on various computer architectures. Along other must-have software tools and libraries, BLAS is the first numerical API to be implemented and optimized on any architecture that targets high-performance computing.

The BLAS API [1] defines three levels of numerical operations: (1) level-1 within or between scalar vectors, (2) level-2 between vector and matrix, and (3) level-3 between matrix and matrix. A typical example of a level-1 operation is the vector-vector addition (`AXPY`) (equivalent Triad in the STREAM benchmark [37]). The level-2 performs, for instance, vector-matrix multiplication (`GEMV`), or the well-known solver of linear system of equations $A \cdot x = b$ (`TRSV`). For level-3, one should mention the General matrix multiplication (GEMM) operation $C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$, which is used as the core block of many computation-intensive benchmarks and applications. The High Performance Linpack (HPL) benchmark [38, 39] from which the Top500 list is built, as well as many other scientific applications are designed to map on this operation as much as possible to reach the maximal computing capacity of the target platform. In the latest years, `GEMM` has also become the kingpin of machine learning and deep learning advances.

Nevertheless, implementing and optimizing BLAS (typically level-3) on a given architecture has never been a trivial task. Straightforward implementations seldom deliver satisfying performance without advanced tiling and blocking techniques. To fill up the core pipeline, eliminate stall cycles and reach near-peak performance, developers compulsorily need to understand the low-level functionality of the hardware and write optimal assembly-level kernels. Multiplicity of BLAS parameters and their combinatorial cases yield up to several hundreds of assembly kernels to hand-tune and to maintain; in which to add extensions of instruction set architecture (ISA) and cache size evolution throughout processor generations. Developing and maintaining such an optimal library requires substantial time and expertise, that sometimes can only be afforded by the processor manufacturers or specialized research groups.

[1] `http://www.netlib.org/blas/blasqr.pdf`

Conventional CPUs are shipped with proprietary libraries that implement BLAS, sparse BLAS routines, and BLAS-like extensions, such as: Intel Math Kernel Library (MKL) [40], AMD Core Math Library (ACML) [41] and IBM Engineering and Scientific Subroutine Library (ESSL) [42]. Open-source options for BLAS-like functionality include the hand-optimized GotoBLAS [43] [44] and its derivative OpenBLAS [45], auto-tuning solution such as Automatically Tuned Linear Algebra Software (ATLAS) [46], and projects that target modern multi-core processors such as Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [47]. The BLAS-like Library Instantiation Software (BLIS) framework [48] is a recent development in the area of open-source and portable BLAS solutions for CPU-based architectures.

Vendors of GPU and other CPU accelerators also develop proprietary BLAS implementations adapted to their heterogeneous computing context: NVIDIA's CuBLAS [49], AMD's clBLAS[50] and rocBLAS [51]. Open-source projects for such architectures include MAGMA [52], clBLAST [53], and KAUST-BLAS [54]. These libraries, written in C-like languages (CUDA, OpenCL), rely on the vendor compiler and runtime API to generate executable code and offload computation onto the device. In order to abstract the hardware complexity and reduce programming effort, scheduling and memory management is hidden as much as possible to the developer and is managed by the deployment runtime and the device driver. Despite facilitating usage by non-expert users, the application portability and performance crucially depends on the vendor's or the open-source community's ability to implement and maintain an optimized BLAS library across multiple architectural generations.

3.1.2 General Matrix Multiplication (GEMM)

3.1.2.1 Basic implementation

The `GEMM` operation is the most widely used function in the BLAS API. Fig. 3.1 depicts the `GEMM` function which performs matrix product between a matrix A of size $m \times k$ and a matrix B of size $k \times n$. This product is then scaled by an α scalar and is accumulated into a matrix C of size $m \times n$, pre-scaled by a β scalar: $C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C$.

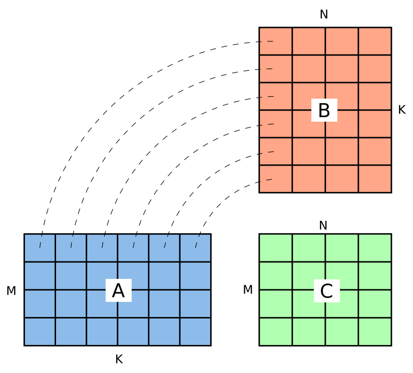


FIGURE 3.1: Matrix multiplication.

Let us assume that matrices are square ($m = n = k$). The basic implementation of `GEMM` is based on the following the loop-based approach:

$$C_{i,j} = \alpha \times \Big(\sum_{t=0}^{n-1} A_{i,t} \times B_{t,j}\Big) + \beta \times C_{i,j} \qquad i,j,t \in [0,n) \tag{3.1}$$

Eq. 3.1 performs, per each $C_{i,j}$ element, $2n+2$ floating-point operations (flops),[2] for a data traffic of $2n+2$ words.[3] Commonly, those floating-point operations can be performed within $n+1$ *fused multiply-add* (FMA) instructions.[4] This naive implementation has a poor arithmetic intensity. Spatial and temporal locality of data accesses in cache levels are not optimal either.

3.1.2.2 Blocked (Tiled) implementation

Goto et al. [43] revisited `GEMM` algorithms with multi-layer *blocking* (or *tiling*), in which each layer is corresponding to a memory level. For simplicity, let consider a two-layer configuration between a slow memory (DDR) and a fast memory (scratchpad or L1 cache). The matrix C on the slow memory is divided into $N \times N$ blocks, each block is of size $b \times b$ ($N = \frac{n}{b}$). The blocksize b is chosen so that the fast memory, whose size is S, can hold at least one block A and one block B ($S \geq 2b^2$ words), or preferably one block C as well ($S \geq 3b^2$ words). Let consider that three blocks A, B and C can fit into S, the communication cost between the slow and fast memory, the I/O traffic and arithmetic intensity (AI) of the blocked algorithm are written as follows:

$$\begin{aligned}
\textit{I/O traffic} \; &= \quad N^2b^2 \times N \quad \text{(read every block of A } (N) \text{ times)}\\
&\quad + \; N^2b^2 \times N \quad \text{(read every block of B } (N) \text{ times)}\\
&\quad + 2N^2b^2 \times 1 \quad \text{(read and write every block of C once)}\\
&= 2N^3b^2 + 2N^2b^2 = \frac{2n^3}{b} + 2n^2 = 2n^3\Big(\frac{1}{b} + \frac{1}{n}\Big)\\
\textit{Complexity} \; &= 2n^3
\end{aligned}$$

$$AI = \frac{\textit{Complexity}}{\textit{I/O traffic}} = \frac{nb}{n+b} = \frac{b}{1+\frac{b}{n}} \approx b \quad (n \gg b) \tag{3.2}$$

[2] $n+2$ multiplications and n additions

[3] $2n$ loads for $A_{i,t}$ and $B_{t,j}$, one load and one store for $C_{i,j}$

[4] Included in the IEEE 754-2008 standard and largely available on modern CPU and GPU architectures. The FMA instruction has advantage of significantly reducing the number of CPU cycles and minimizing the accumulated error due to successive rounding steps.

As can be seen from Eq. 3.2, we can improve performance of the blocked algorithm between two successive memory levels by increasing b. Applying recursive tiling on contemporary CPU and GPU architectures is apparently the optimal approach for `GEMM`. Furthermore, Goto et al. [43] also proposed an additional *packing* step before inner computations, which consists in reordering A and B data blocks into a pre-defined contiguous layout,[5] in order to maximize cache hit ratio and minimize penalty of Translation Lookaside Buffer (TLB) misses. Through the packing step, the implementation will also be able to handle multiple parameter combinations (*transa, transb, uplo, sidea* etc.) with only several well-defined inner micro-kernels. This allows implementing BLAS functions without tuning hundreds of assembly kernels, produces well-structured and highly maintainable code, whose a successful example is the BLIS framework.

3.2 BLAS-like Library Instantiation Software (BLIS)

BLIS, stands for *BLAS-like Library Instantiation Software* [48], is a sub-project of libflame [55]. The libflame project implements LAPACK-like features and lies closely on BLIS for the BLAS support. Both BLIS and libflame are developed by the Science of High-Performance Computing (SHPC) group at the University of Texas at Austin. They are released under the open-source BSD 3-clause license, facilitating adoption by industry.

During many years, the research community was missing a well-structured, open-source, light-weight, portable and high performance BLAS library. Proprietary implementations are considered as black-boxes and platform-specific. Open-source libraries like ATLAS and OpenBLAS appear too bloated or difficult to port and to optimize on a new architecture. Researchers and vendors need an easy and extensible framework as an experimental tool, not only to implement and tune new linear algebra algorithms, but also to maintain and optimize easily BLAS functions on next-generation architectures. These crucial points have been tackled and successfully solved within the BLIS framework for conventional cache-based CPUs.

Inspired from GotoBLAS [43], BLIS is designed with fundamental principles in dense linear algebra, including incremental and recursive construction of BLAS operations for code-reusability, data-packing and cache/register blocking for optimal locality, as well as ability to integrate platform-specific assembly code. Fig. 3.2 depicts the global algorithm of the `GEMM` operation in BLIS. The three matrices A, B and C are partitioned and traversed through five loops around a *micro-kernel*. The micro-kernel performs a

[5] This layout is similar to the row-major `GEMM_TN` format: A transposed and B non-transposed.

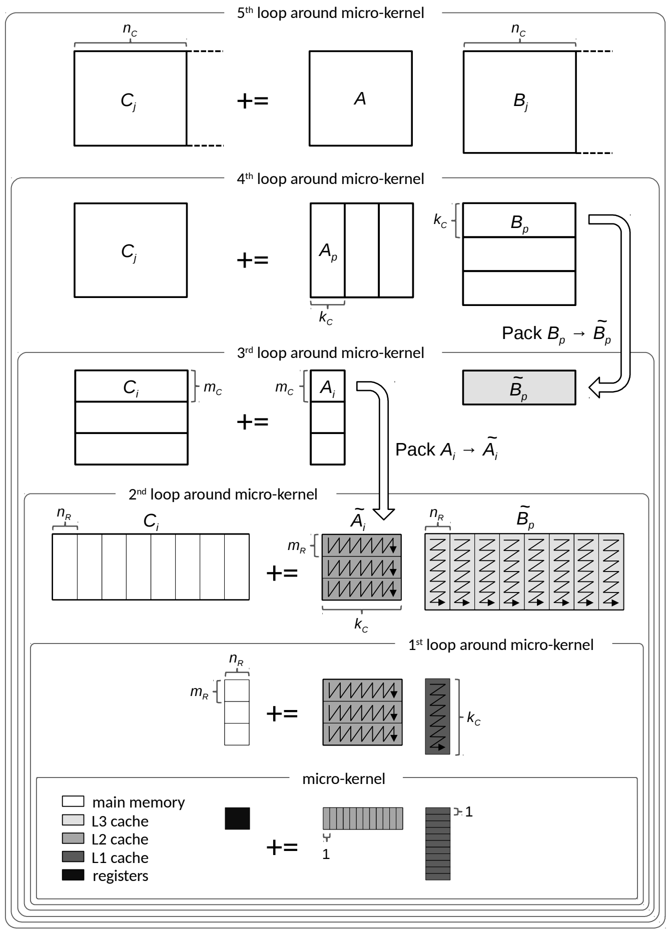

5th loop around micro-kernel
n_C
C_j
+=
A
B_j
4th loop around micro-kernel
C_j
A_p
k_C
B_p
Pack $B_p \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_C
A_i
$\tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
n_R
$\tilde{A}_i$
m_R
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

FIGURE 3.2: Cache-based layer design of BLIS.
Courtesy of: Field G. Van Zee and Robert A. Van de Geijn.

rank-k update and constitutes the sixth loop. Other level-3 operations are then built on top of `GEMM` and this partioning method.

Main advantages of BLIS can be mentioned as follow:

- BLIS introduces a reduced set of micro-kernels (*gemm*, *trsm*, *gemmtrsm*), written in the portable C99 standard by default, as a reference implementation.
- BLIS defines a reduced set of execution parameters (cache sizes, memory alignment, memory allocator), largely used by the internal algorithms, but can differ significantly from one architecture to another.
- BLIS provides a user-defined configuration header which allows arbitrary modification of these execution parameters and easy plug-and-play of ISA-specific micro-kernels to generate a nearly optimized BLAS library on any cache-based architecture, without touching the core functions of BLIS.

This abstract design enables a custom fit on any cache-based system implementing any instruction set with straightforward portability and high-performance. Cache-blocking and packing implementation techniques in BLIS have been proved to be analytically optimal [56] on multi-core and NUMA memory hierarchies, delivering competitive performance to other vendor libraries [57].

3.3 Summary

New multi- and many-core architectures keep appearing and are moving fast. On these systems, writing library and software are becoming more and more challenging. Moreover, code-portability sometimes counters performance, due to the hardware diversity. The BLIS framework has emerged as a promising solution for instantiating a light-weight and high-performance BLAS library for *cache-based* architectures.

However, both HPC design strategy and modern embedded and intelligent computing are coming up with more and more power-efficient and non-conventional architectures, on which, writing a BLAS library in pace with the hardware represents a big software challenge. They often do not have a hardware-assisted cache coherency, whereas a software-based cache protocol would suffer significant overhead. Hardware prefetcher, out-of-order execution and advanced branch prediction are commonly discarded to reduce power consumption (and fortunately avoid security vulnerabilities [6]).

On those architectures, computation-intensive parts of code are expected to perform software-managed data-prefetching, by leveraging Direct Memory Access (DMA) engines and operating on scratchpad memories, instead of the traditional cache-based load/store scheme. Support of the asynchronous programming model based on Remote-DMA (RDMA), considered the key enabler of performance on DMA-based architectures, is currently the missing point of any BLAS-like library on these later platforms.

[6]https://meltdownattack.com/

Chapter 4

Kalray Massively Parallel Processor Array (MPPA)

Any sufficiently advanced technology is indistinguishable from magic.
– Arthur C. Clarke.

4.1 Introduction

4.1.1 Company

Kalray is a fabless semiconductor company, founded in France in 2008 after a spin-off from the CEA (French Alternative Energies and Atomic Energy Commission). Kalray is specialized and pioneering in developing a new family of many-core processors, namely *Massively Parallel Processor Array* (MPPA). The MPPA architecture offers unique parallel computing capacity, low latency and low-power consumption. The Very Long Instruction Word (VLIW) core architecture, distributed non-coherent memory system and dual control-plane and data-plane Network-on-chip enable time-predictability necessary to embedded, mixed-critical and real-time systems. The MPPA massively parallel architecture is also suitable for modern and energy-efficient HPC workloads, typically in the area of image processing, computer vision and autonomous vehicles.

4.1.2 MPPA architecture overview

The second generation of Kalray MPPA-256 processor, codenamed Bostan (see Fig. 4.1) embeds 256 VLIW compute cores grouped into 16 compute clusters (CC) and 16 system cores in two unified I/O subsystems (IOS). The sixteen compute clusters are organized in a 4 x 4 grid connected by a 2D torus Network-on-Chip (NoC). The processor delivers peak performance of 634 GFLOPS in single precision and 317 GFLOPS in double precision at a frequency 600 MHz, within a power consumption of 20 W.

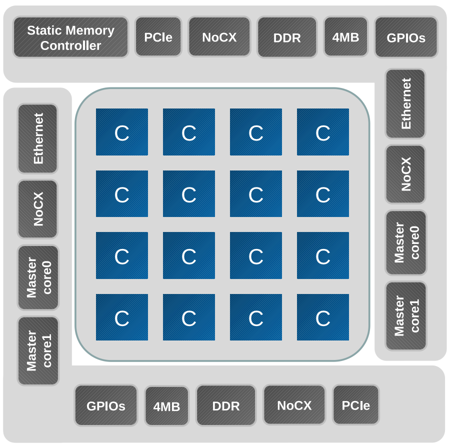


FIGURE 4.1: MPPA2-256 processor overview. (Source: Kalray).

Each compute cluster features 2 MB of local memory (SMEM) shared between 16 user cores (Processing Elements-PEs). One system core, known as Resource Manager (RM), is reserved for running operating system, resources management and performing DMA jobs. Each of these core features 8 KB of level-1 instruction cache (L1-I) and 8 KB of level-1 data cache (L1-D). DMA engines of each compute cluster and I/O subsystem provide high bandwidth and low latency transfers between SMEM-SMEM (symmetric inter-cluster) and SMEM-DDR memory (asymmetric cluster-IO).

Each I/O subsystem (North and South) contains two quad-core CPUs (also known as Master Cores) and 4 MB of SMEM. A Master Core contains four private 32 KB of L1-I per core and 128 KB of coherent L1-D (4×32 KB) and four DMA interfaces (one per CC column). Each I/O subsystem integrates an off-chip DDR3 memory, a 8-lanes Gen3 PCI-Express and 10G Ethernet interfaces, as well as an Interlaken interface to extend the NoC links across multiple MPPA-256 processors. The main goal of I/O subsystems is to deploy user applications, to provide system and software services to CCs and to act as gateways to the outside world.

4.2 Programming models

The complete Kalray Software Development Kit (SDK) features standard C/C++ and Fortran compilers, OpenMP/Pthreads support, low-level programming and communication libraries, as well as an OpenCL offloading runtime. They are divided into two main programming model:

1. A low-level distributed and inter-process communication (IPC) POSIX environment, supporting Fortran and C/C++.
2. A high-level host-based acceleration runtime based on the standard OpenCL 1.2 specification.

4.2.1 Distributed-memory POSIX-C

In the distributed POSIX programming model, an MPPA2-256 processor is exposed as a distributed multi-process system. Each compute cluster is considered as an individual computing unit with a *main* function and a proper memory space (SMEM). Each *main* function can be written in standard C/C++ or Fortran. It has direct access to the CC's SMEM in load/store model, and to the SMEM of other CCs or the DDR of IOS by explicit NoC communication, mainly based on software-triggered DMA engines to perform asynchronous transfers. Programming multiple clusters on MPPA in the POSIX model is similar to the *one-sided* message-passing model, largely used in HPC applications via the Message-Passing interface (MPI2) [9].

The MPPA asynchronous library (namely *mppa-async*) implements DMA-based asynchronous one-sided put/get, remote atomic operations, peek, poke and two-sided queues. These features enable high-throughput data-plane communication, and low-latency control-plane signal and synchronization. This programming model requires high design and development efforts, but can offer parallel applications great performance improvement over conventional platforms.

4.2.2 Host-based OpenCL acceleration

The Kalray SDK also allows programming MPPA as a compute-accelerator based on the OpenCL 1.2 Data-Parallel model [8]. A software-based cache protocol, so called *Distributed Shared Memory* (DSM), is used to implement a globally coherent cache between the 256 cores, on which OpenCL kernels are deployed.

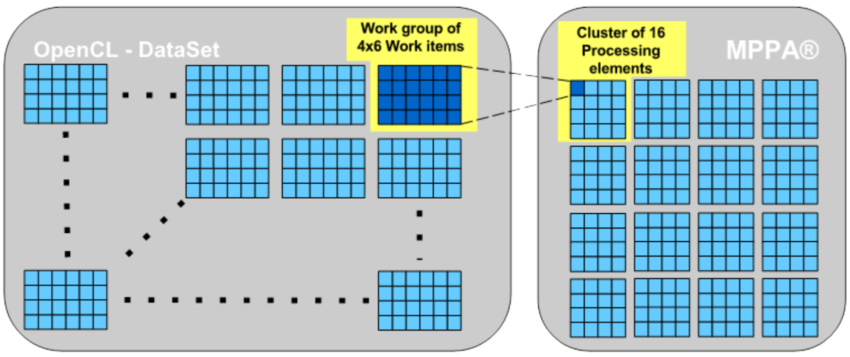

OpenCL - DataSet
Work group of 4x6 Work items
Cluster of 16 Processing elements
MPPA®

FIGURE 4.2: OpenCL Data-Parallel: execution mapping. (Source: Kalray).

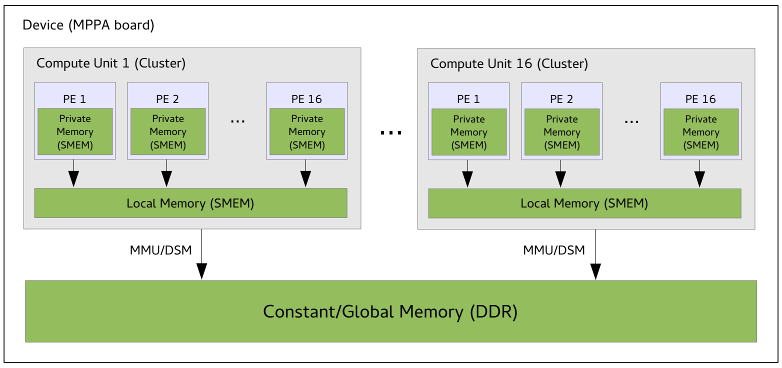


FIGURE 4.3: OpenCL Data-Parallel: memory mapping. (Source: Kalray).

Fig. 4.2 and Fig. 4.3 depict respectively the execution mapping and memory hierarchy of the OpenCL Data-Parallel model on MPPA cores. Work-items within a work-group is linearized and mapped on one PE. Collective operations like `barrier()` and `async_work_group_copy*()` are accordingly handled by the runtime compiler to comply with desired execution order. Each work-group has load/store access to the global DDR memory with coherence based on DSM, and a private local memory (`__local`), configurable in a range of 8, 16, 32 or 64 KB per PE. The work-group local memory can be used to prefetch data by asynchronous primitives (`async_work_group_copy()` and `async_work_group_strided_copy()`) defined in the OpenCL 1.2 specification.

As part of contributions of this thesis, an extended set of OpenCL asynchronous primitives, including general strided, 2D and 3D copy, was introduced and implemented in the Kalray OpenCL toolchain and is discussed in Chapter 8 on page 85.

4.3 Summary

The Kalray MPPA processor family is a highly parallel computing platform. Programming the MPPA processor requires careful segmentation of data and explicit transfers onto local memories.

This section closes the introduction part of the dissertation. In this opening segment, we have presented the thesis context, the challenges of current and future HPC trends and the potential as well as problematics of clustered many-core architectures. Application focus was given to (1) the lattice Boltzmann method as a *memory-bound* context with high bandwidth requirement, and (2) the BLAS library as a *compute-bound* scenario (level-3) with its intrinsic complexity and portability issues.

In the following chapters, we present approaches and contributions to address these two typical fields of the HPC domain, revised and shaped for the clustered DMA-based many-core architectures. We also implement, demonstrate and report analysis of each of those solutions on the Kalray MPPA2-256 processor, as well as other mainstream computing architectures whenever applicable.

Contributions

Chapter 5

Optimizing 3D LBM on Many-core Processors

However difficult life may seem, there is always something you can do and succeed at.
– Stephen Hawking.

5.1 Introduction

As discussed in Chapter 2, Section 2.1.2 (page 13), LBM implementations are memory-bound and hardly obtain high 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 prevents from using 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 [31] and [32], 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.

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 [23] 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 simple 2D/3D asynchronous copy functions on DMA-based 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 [26].

The remainder of this chapter is structured as follows. Section 5.2 presents some related works that are relevant for our contributions. Section 5.3 introduces some low-level asynchronous transfer primitives required for building 3D stencils streaming algorithm on the MPPA processor. Section 5.4 presents an overview and technical details of the new LBM streaming algorithm using these asynchronous transfers. Experimental results are presented in Section 5.5, and we conclude in Section 5.6.

5.2 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 [21]. 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 [23] 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. [58] 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 [59] 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. $12 \times 35 \times 12$) and cannot 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. [60] 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 underlying target hardware for the sake of genericity.

5.3 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. 5.1, 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, the size of each element in bytes being denoted *size*. 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, one must deal with a local stride of two pixels between each data block. 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. 5.2), 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*).

```c
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 5.1: A part of MPPA Async API for 3D transfer. Prototype of *get* and *put* are similar.

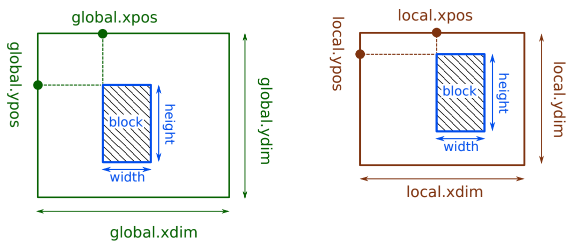


FIGURE 5.2: Illustration of `point2d_t` datatype for 2D copy. 3D copy is conceived by adding `depth` and `Z` fields.

5.4 Pipelined 3D LBM stencil on clustered many-core processors

5.4.1 Global algorithm

In the following, we take the D3Q19 LBM kernel from OPAL [23] 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, 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. 5.3), 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.

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. 5.7). On boundary subdomains (e.g. subdomains 0, 1, 2, 3 of Fig. 5.7), 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.

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

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

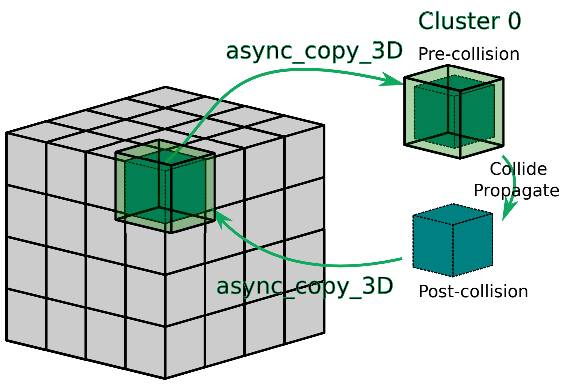


FIGURE 5.3: 3D LBM/stencil decomposition where Main-block subdomain (green) is copied with its surrounding halo layers (if exists) and one extra subdomain (blue) is needed to store post-collision state.

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. 5.3 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, from the PE0 in the Pthreads 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.

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. 5.4) 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

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();
```

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?

	Prolog	m=0	1	2	3	4	5	6	7	Epilog
		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 5.4: 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}`).

5.4.2 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. 5.3) was done by *space filling curves*, such as Morton or Hilbert curves, in [61]. These curves have been efficiently implemented by bit-interleaving in [62] or lookup-table in [63]. 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. 5.5. 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.

```
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 5.5: 3D Row-major subdomain-indexing $f : \mathbb{N} \rightarrow \mathbb{N}^3$.

5.4.3 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 calculated by the below formula and represented as in Fig. 5.6.

$$g(C_x, C_x, C_x) = \frac{(C_x + 2)^3 - C_x^3}{(C_x + 2)^3} \tag{5.1}$$

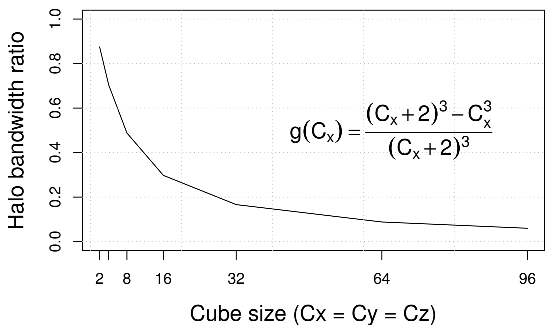


FIGURE 5.6: Halo bandwidth ratio.

Since the halo cell are needed for spatial dependency due to domain decomposition, they do not change the total number of updated cells nor the overall performance. As can be seen in Fig. 5.6, the halo bandwidth costs up to 29% of data throughput on block size $16 \times 16 \times 16$, but is enthusiastically reduced to as few as 10% on block size $64 \times 64 \times 64$. This leads to think that further many-core architectures with larger local memories can noticeably improve 3D LBM performance by enlarging the subdomain size. The best performance is so 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:

$$\begin{aligned}
\textbf{Find:}\quad & (D, C_x, C_y, C_z) \\
\textbf{Maximize:}\quad & C_x \times C_y \times C_z \quad \textit{(nodes updated per subdomain)} \\
\textbf{Minimize:}\quad & F_x \times F_y \times F_z \quad \textit{(per subdomain storage)} \\
\textbf{Minimize:}\quad & \tfrac{(F_x \times F_y \times F_z) - (C_x \times C_y \times C_z)}{F_x \times F_y \times F_z} \quad \textit{(HBW)} \\
\textbf{Subject to:}\quad & \frac{D \times ((F_x \times F_y \times F_z) + (C_x \times C_y \times C_z)) \times 19 \times 4}{1024^2} \leq 1.4 \\
& F_d = C_d + 2 \ ; \quad C_d \in \{2^n\} \ ; \quad D, n \in \mathbb{N}^+
\end{aligned} \tag{5.2}$$

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.

5.4.4 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. [26]. 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. 5.7). 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. 5.7.

$$
\begin{aligned}
const\ A &= (A_x, A_y, A_z) = (0,0,0) \\
R &= (R_x, R_y, R_z) \\
&= (m_x \times C_x,\ m_y \times C_y,\ m_z \times C_z) \\
B_{ad} &= A_d + local\_offset(m_d, M_d) \\
B_{rd} &= R_d + remote\_offset(m_d, M_d) \\
S_d &= F_d + halo\_cutoff(m_d, M_d)
\end{aligned}
\tag{5.3}
$$

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 $S_d = F_d + halo\_cutoff(m_d, M_d)$. It is read from the remote position $B_{rd} = R_d + remote\_offset(m_d, M_d)$ and written to the local position $B_{ad} = A_d + local\_offset(m_d, M_d)$. 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. 5.1.

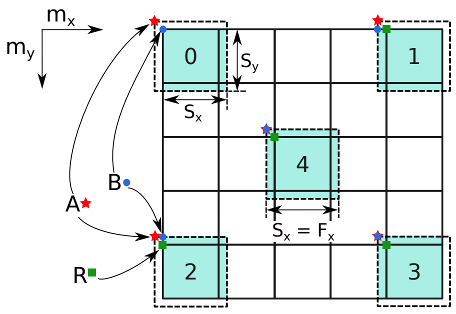


FIGURE 5.7: Local/Remote copied index in 2D (in lattice node) with A: begin of the local buffer = (0,0); R: begin of the remote main block cuboid (without halo); B: begin of the copied cuboid (S), represented by: B_a: index of S on local memory (from A) and B_r: index of S on global memory (from R).

TABLE 5.1: 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}.

5.5 Results and discussions

5.5.1 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 cannot 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 cannotice in Fig. 5.8, 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. 5.8 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. 5.8 also shows the expected 2× performance speedup by using two DDRs compared to the single-DDR mode.

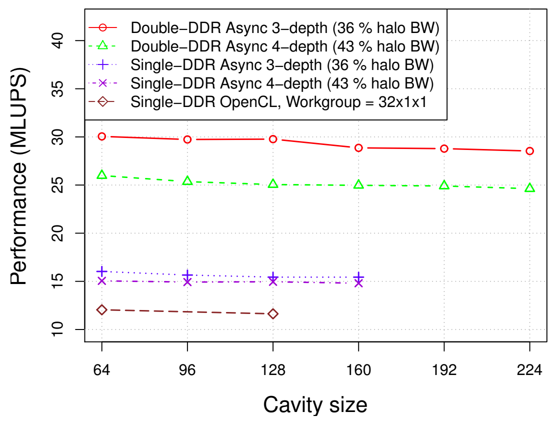


FIGURE 5.8: OPAL_async vs. OPAL OpenCL on MPPA for duration = 1000 steps. Performance in MLUPS TODO

5.5.2 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. 5.9, 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 performance line scales from 1 cluster to 8 clusters, then reaches almost a stable value of between 20 to 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 (6 GB/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 $350/(2 \times 19 \times 4) = 2.3$ flops/byte, which means that each CC spends more time working on a 3D data block. This explains in Fig. 5.9 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 22 MLUPS is to apply the performance estimation formula presented by McIntosh-Smith et al. [22]:

$$P = \frac{B \times 10^9}{19 \times 2 \times 4 \times 10^6} \text{ (MLUPS)} \tag{5.4}$$

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:

$$P_h = \frac{6.0 \times 10^9}{19 \times 2 \times 4 \times 10^6} \times \frac{8^3}{10^3} = 20.2 \text{ MLUPS} \tag{5.5}$$

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).

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

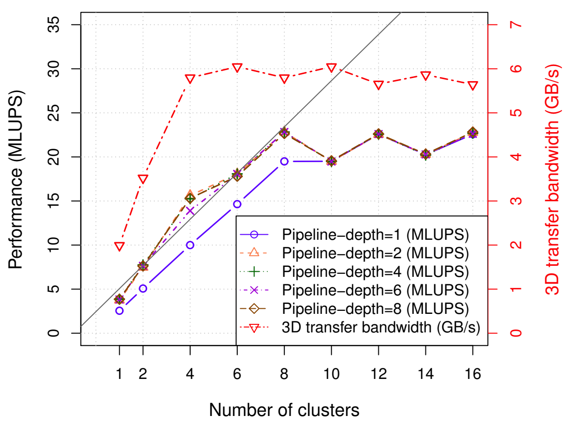


FIGURE 5.9: Performance extrapolation of OPAL_async on $8 \times 8 \times 8$ subdomains with the first eight clusters correlation represented by a gray line for 1000 timesteps and cavity size 128.

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.

5.6 Conclusions

We introduce 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 expressions are valid with or without *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 expect 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 find 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 next chapter, we 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.

Chapter 6

In-place LBM Propagation Algorithms

The biggest difference between time and space is that you can't reuse time.

– Merrick Furst.

6.1 Introduction

Besides the *two-lattice* propagation algorithm, several *one-lattice* algorithms were introduced to reduce the memory footprint (by working on only one lattice array) and also to improve data locality of the LBM. Most of them operate elaborate exchange of PDFs between neighboring lattice nodes in the parallel execution context. The fundamental challenge of any *one-lattice* algorithm is that memory accesses (read and write) must be performed carefully on the same lattice buffer to enforce the spatio-temporal dependency between nodes and time steps. More precisely, in a parallel algorithm, new PDFs of a node after collision should not be written directly into memory without special care about the old data that may have not been used yet by neighboring nodes.

On another hand, LBM boundary conditions on new physical models tend to be more and more elaborate in terms of interaction between solid and fluid cells. Typical LBM boundary conditions, such as simple bounce-back or interpolated bounce-back [30], imposes specific exchange rules of PDFs between adjacent nodes. Combining these conditions with existing *one-lattice* algorithms such as *compressed-grid*, *swap*, *AA-pattern* or *Esoteric twist* (see the next section), raises considerable complexity in implementation and code assessment, especially for 3D domains.

Our key contributions in this chapter are as follows:

1. Two novel algorithms, two-wall and three-wall, in the *one-lattice* LBM class and their implementation detail in OpenMP for shared memory context and OpenCL for heterogeneous memory systems.
2. Comparison and performance analysis of these two-/three-wall algorithms versus the AA-pattern (*one-lattice*) and the state-of-the-art two-lattice algorithm on current mainstream computing platforms: Intel Xeon NUMA CPU, Intel Xeon Phi Knights Landing MIC and NVIDIA Tesla Pascal GPU.
3. Promising results when using the proposed algorithms to implement complex LBM and CFD problems at high spatial resolution on current and future many-core processors.

The remainder of this chapter is structured as follows. Section 6.2 describes our approach, advantage and limitations of the two-wall algorithm. Section 6.3 presents the three-wall algorithm as an improvement of two-wall. Implementation details in OpenMP and OpenCL are given in Section 6.4. Experimental results and discussion on CPU, Xeon Phi KNL and GPU Pascal are given in Section 6.5 and we conclude in Section 6.6.

6.2 Two-wall propagation algorithm

In this section, we present a new parallel *one-lattice* LBM propagation algorithm, named *two-wall* algorithm, in which the lattice grid G is updated in-place. The code change for adapting this algorithm on complex boundary conditions remains simple, easy to understand and to implement.

6.2.1 Algorithm

We propose a new propagation algorithm enabling in-place update when using the *pull* scheme. Note that while we are assuming the *pull* scheme in this section, applying the *push* scheme is entirely possible as well. The main idea for the *pull* scheme is, before colliding and streaming new PDFs, outdated PDFs must be copied out and saved in a temporary buffer (small, 2D buffer). Any spatial dependency which requires reading data from a specific node in the grid G, should read in the temporary buffer instead. More in detail, let say that the grid G is allocated in the C99 multi-dimensional array convention: `float Grid[`L_z`][`L_y`][`L_x`][`Q`]`. We decompose the grid G into L_z walls. Each wall is composed of $L_y \times L_x$ nodes. The lattice grid G is updated wall-by-wall. There will be a sequential `for` loop in the z-direction to sweep through the L_z walls of G. At each z-iteration, the wall z (referenced by `Grid[`z`]`) will be updated from time step t to $t+1$ and its new PDFs are written in-place into `Grid[`z`]`. We then introduce two *wall* buffers, a `past_wall` and a `current_wall`, each of $L_y \times L_x$ nodes (`float Walls[2][`L_y`][`L_x`][`Q`]`). These two walls, swapped from one time step to another, are respectively used to store old PDFs of the wall $z-1$ and z before they were overridden by the new PDFs of the collision operation. Fig. 6.1 illustrates the two-wall algorithm on the lattice grid G of size $L_z \times L_y \times L_x$.

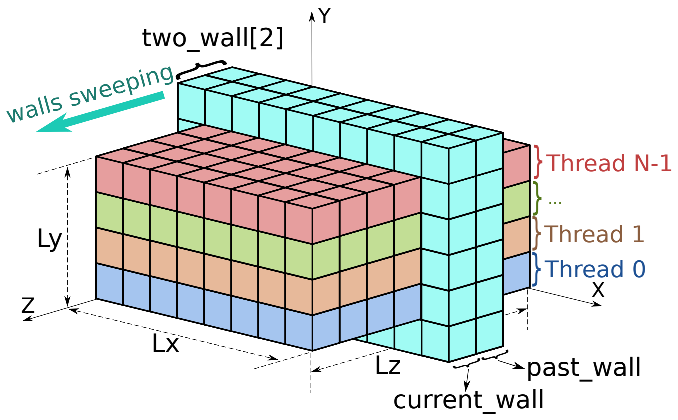


FIGURE 6.1: In-place updates on a grid with N threads using *two-wall* algorithm. Wall buffers can be allocated with or without *ghost-layer*, depending on the application.

Before updating the wall z (a 2D YX-wall) from the time step t to $t+1$, the `current_wall` is used for saving the pre-collision data of this wall before it is modified by the in-place post-collision propagation. Idem, the `past_wall` contains the pre-collision data of the back wall $z-1$, as this wall in G has been updated to the time step $t+1$ in the previous z-iteration, thus cannot be used here for the wall z at the time step t. Inside each wall, lattice nodes can be independently copied into the `current_wall`. Then a memory-fence barrier (if parallel execution) is needed to ensure that all copies are done before performing the collision and the in-place propagation on the wall z of G. At the end of each wall-update, all working threads must perform another barrier to achieve global synchronization and to avoid *data-races* between concurrent threads. Pointers to `current_wall` and `past_wall` are then swapped before advancing in the next wall in z-direction. The `current_wall` of the previous iteration $z-1$ now becomes the `past_wall` of the current iteration z. Inversely, the `past_wall` of the previous iteration $z-1$ is now the `current_wall` of the current iteration z and can be recycled for storing the pre-collision state of the current wall z, before being overwritten by the upcoming collision. Algorithm details are presented in Fig. 6.2 in which the D2Q9 lattice is used.

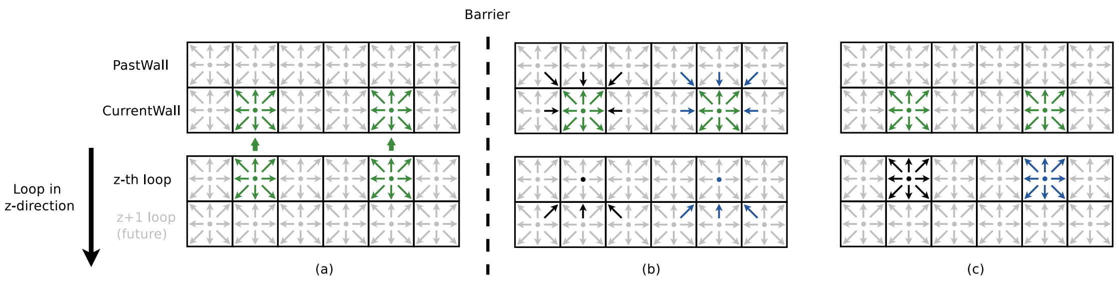


FIGURE 6.2: D2Q9 version of Two-wall in pull scheme: Copy operation (a) is represented by green cells. A barrier is needed to respect the *read-after-write* dependency between (a) and (b). Then, any node in the z-wall (black and blue cells) can be read (b), collided and stored in-place (c) independently.

From the Fig. 6.2, parallelism is therefore possible at two steps: (a) and merge of (b)+(c). These two steps are separated by a barrier to ensure the *read-after-write* requirement of the copy operation of `current_wall`, since the collision will read data from the `past_wall` and `current_wall` for neighbors in $z-1$ and z walls. This easily yields an OpenMP implementation by two `#pragma omp parallel for` statements or an OpenCL one, consisting in two kernels (one for (a) and one for (b)+(c)). They will be presented later in Section 6.4.

6.2.2 Advantages and limitations

The main advantage of the two-wall algorithm is that adapting complex boundary conditions is trivial by replacing any access to $z-1$ and z cells by `past_wall` and `current_wall`, respectively. This means that there is only one kernel code to maintain, instead of two on AA-pattern or Esoteric twist (even and odd). Adding and testing new boundary conditions will require less effort. Data locality is also a good point. Memory accesses from different cores have more chance to share and hit in cache (2D walls) than other algorithms, where PDFs are read from or written to neighbors in three dimensions, thus increasing penalty of cache-miss and cache-thrashing.

In return, there are two performance challenges of the two-wall algorithm. The first is the data movement of $4q$ (see Table 6.1) per node per time step versus $2q$ or $3q$ of two-lattice and others. This difference can be visible on a bandwidth-sensitive platform with non-persistent cache between kernel executions (like OpenCL on GPU). Secondly, updating a wall requires two barriers of all threads, resulting a total of $2L_z$ barrier count for a grid G of L_z walls. Multiplied to the number of time step T, the algorithm faces a very high number of barriers. As a matter of fact, two-wall requires fast barrier support from the hardware and/or an efficient event-driven programming language to launch and synchronize successive kernels at least cost, ideally via persistent threads and asynchronous runtime in background. Subsequently, in the next section, we introduce three-wall, an extended version of two-wall which breaks down the *read-after-write* dependency within each wall and reduces the number of barriers from $2L_z$ to L_z.

6.3 Three-wall propagation algorithm

In two-wall, the memory-fence barrier after the wall-copy is mandatory because the `current_wall` is needed in the following collision step (*read-after-write*). This leads to think that if the wall z has been copied beforehand (for example at the iteration $z-1$), the three steps (a) (b) and (c) in Fig. 6.2 can be merged into one unique kernel. This implies that the new kernel must perform the copy operation of the next wall, which we will refer to as *future wall*. Appropriately, three-wall introduces a third wall buffer, called `future_wall`. The algorithm sketch of three-wall is presented in Fig. 6.3. During execution of the wall z, the compute kernel of three-wall copies the next wall (`Grid[`$z+1$`]`) into `future_wall`. On the next iteration, the `future_wall` becomes the `current_wall`, the `current_wall` turns back to the `past_wall`, and the `past_wall` switches up to the `future_wall` and the process starts again. Now, since `current_wall` (formerly `future_wall`) has already been copied in the previous iteration (or in prolog), there is no more need to ensure ordering of cell updates of the wall of interest, thus the memory barrier is no longer required for three-wall. The total barrier count is therefore reduced to L_z, with the same quantity of data movement. Table 6.1 depicts the identical quantity of data movement between two-wall and three-wall algorithm, in either direct- and indirect-addressing, but with a different number of barrier count.

TABLE 6.1: Estimation of data requirement for updating one node with both *two-wall* and *three-wall* algorithm, and then the barrier count of each algorithm.

Direct addressing			**Indirect addressing**		
#	Type	Comment	#	Type	Comment
q	PDF	Copy: loads	q	PDF	Copy: loads
q	PDF	Copy: stores	q	PDF	Copy: stores
q	PDF	Collision: loads	q	PDF	Collision: loads
q	PDF	Collision: stores	$q-1$	IDX	Collision: lookup
			q	PDF	Collision: stores
$\Sigma = 4q$ PDFs			$\Sigma = 4q$ PDFs + $(q-1)$ IDXs		

Barrier count	
Two-wall	# barriers = $2L_z$
Three-wall	# barriers = L_z

Moreover, as the copy of `future_wall` is completely independent from the computation (see Fig. 6.3), three-wall exposes the possibility of using streaming load/store instructions to perform cache-bypassing memory operation, avoid polluting useful data in different cache levels of computation. It is also possible to perform asynchronous copy of the `future_wall` on DMA-enabled architectures, so that CPU cores only need to focus on the computation task. Note that the extra memory allocation for temporary wall buffers is considered negligible compared to the main lattice buffer. For instance, three-wall buffers represents 2.3% of additional memory on a 128^3 domain $((3 \times 128^2)/128^3)$, or 1.1% on a 256^3 domain. Furthermore, on a non-cubic or complex-geometry domain, the developer can make a choice of loop direction (either in L_z, L_y or L_x) that satisfies a specific trade-off between performance and memory consumption. Our algorithm can also be extended to more than three walls to fill the performance gap due to the barrier cost, in exchange of the additional memory footprint.

6.4 Implementations

We present in this section two short versions of our D3Q19 LBM implementation, the first one within the shared-memory OpenMP paradigm and the second in OpenCL on heterogeneous platforms including accelerators.

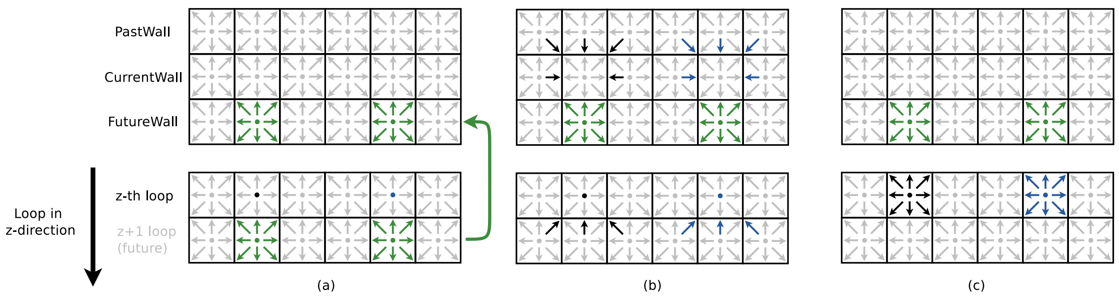


FIGURE 6.3: D2Q9 version of *three-wall* in pull scheme: Copy operation (a) is represented by green cells. The barrier is no longer needed, since `FutureWall` is only written (not read) in the current iteration. This removes the *read-after-write* dependency of *two-wall*, and exposes possibility of using streaming stores for further optimization. The black and the blue cells are then independently read (b), collided and stored in-place (c).

6.4.1 OpenMP

Both two-wall and three-wall possess a for-loop in one dimension. In this work, we choose to pin the loop on the z-direction and to allocate the grid buffer as `float Grid[`L_z`][`L_y`][`L_x`][`Q`]`. For the sake of simplicity, computation within each L_z wall is based on `#pragma omp parallel for` statements which embrace two inner-loops in L_y and L_x direction. Barrier in two-wall is implicitly done by two separate OpenMP `pragma` which involve fork/join from the master thread at each time. Further optimization can use `#pragma omp barrier` between persistent threads which can even embrace the z-loop.

Wall buffers are allocated as a circular buffer `float Walls[3][`L_y`][`L_x`][`Q`]`. On each z wall, threads (or rather the master thread) compute(s) indexes of the `past_wall`, `current_wall` and `future_wall` (if three-wall) as i_p, i_c, and i_f respectively by the modulo operation based on z:

$$\begin{aligned} i_p &= z && \mod 3 \\ i_c &= (z+1) && \mod 3 \\ i_f &= (z+2) && \mod 3 \end{aligned} \tag{6.1}$$

Here, instead of assigning $(z-1) \mod 3$ to i_p, $z \mod 3$ to i_c and $(z+1) \mod 3$ to i_f, which makes natural sense, we choose to begin at $z \mod 3$ until $(z+2) \mod 3$ for i_p, i_c, and i_f respectively. This avoids the negative first-case where $z = 0$, while i_p, i_c and i_f have distinct values ranging in $\{0, 1, 2\}$. Note that on two-wall, this range is $\{0, 1\}$ since $\mod 2$ is used instead of $\mod 3$. Implementation details of two-wall and three-wall are shown in Fig. 6.4 in pseudo-code and Fortran-like OpenMP syntax. Fig. 6.2 and

Fig. 6.3 graphically describe the code data accesses patterns. We note that on three-wall, a preliminary action (prologue) needs to be taken. It consists in performing copy of the first `future_wall` ($z = (0 + 1) \mod 3$) before entering the for-loop, so that the first iteration can directly access its `current_wall`.

```
for z in 0 .. Lz-1 do
    PastWall    = Walls[(z ) mod 2];
    CurrentWall = Walls[(z+1) mod 2];

    $OMP PARALLEL PRIVATE(y,x)
    for (y,x) in (0..Ly-1, 0..Lx-1) do
        copy cell Grid[z][y][x] to CurrentWall[y][x];
    end for

    // Implicit barrier here between two OMP pragma's

    $OMP PARALLEL PRIVATE(y,x)
    for (y,x) in (0..Ly-1, 0..Lx-1) do
        collide and inplace update Grid[z][y][x], using PastWall, CurrentWall and Grid[z+1];
    end for
end for
```

(A) Two-wall.

```
FirstWall = Walls[(0+1) mod 3];      // prolog
$OMP PARALLEL PRIVATE(y,x)
for (y,x) in (0..Ly-1, 0..Lx-1) do
    copy cell Grid[0][y][x] to FirstWall[y][x];
end for

for z in 0 .. Lz-1 do
    PastWall    = Walls[(z ) mod 3];
    CurrentWall = Walls[(z+1) mod 3];
    FutureWall  = Walls[(z+2) mod 3];

    $OMP PARALLEL PRIVATE(y,x)
    for (y,x) in (0..Ly-1, 0..Lx-1) do
        if z < Lz-1 then
            copy cell Grid[z+1][y][x] to FutureWall[y][x];
        end if
        collide and inplace update Grid[z][y][x], using PastWall, CurrentWall and Grid[z+1];
    end for
end for
```

(B) Three-wall.

FIGURE 6.4: Shared-memory OpenMP pseudo-code of one time step with Two-wall and Three-wall in D3Q19. We assume AoS (Array of Structures) storage on CPU architecture. Fortran-like `$OMP PARALLEL` pragma is optional.

6.4.2 OpenCL

In the implementation of OpenCL, the two `#pragma omp parallel` statements of the OpenMP code of two-wall is replaced by two 2D OpenCL kernels: one for wall-copy and one for wall-compute. Regarding three-wall, similarly to OpenMP, only one OpenCL kernel is needed inside each z-iteration which includes (a) copy of future wall, (b) collision and (c) in-place update. Implementation of these two kernels in OpenCL is mostly identical to ones of the OpenMP code in the previous section, with the loops in L_y and L_x broken into two-dimensional $L_y \times L_x$ global work-items. Fig. 6.5 illustrates the OpenCL implementation of two-wall and three-wall, in which we depict the code from the host side (the for-loop in L_z direction) which enqueues OpenCL kernels to the device. Regarding the synchronization between kernels, we employ the `cl_event` management provided by the OpenCL API [8]. Kernels enqueued to the device can be set to a specific execution order by their event-dependency. This allows us to explicitly enforce the FIFO order independently from any specific OpenCL driver implementation, which for a more efficient utilization of the hardware, enables the out-of-order execution by default.

```
for z in 0 .. Lz-1 do
    Set kernel argument CopyWall.z = z;
    Enqueue FIFO kernel CopyWall to device;

    Set kernel argument Compute2Wall.z = z;
    Enqueue FIFO kernel Compute2Wall to device;
end for

```

(A) Two-wall.

```
// Prolog
Set kernel argument CopyWall.z = 0;
Enqueue FIFO kernel CopyWall to device;

for z in 0 .. Lz-1 do
    Set kernel argument Compute3Wall.z = z;
    Enqueue FIFO kernel Compute3Wall to device;
end for
```

(B) Three-wall.

FIGURE 6.5: Heterogeneous memory OpenCL implementation of one time step in Two-wall and Three-wall in D3Q19. Each `$OMP PARALLEL` in Fig. 6.4 is replaced by an $L_y \times L_x$ OpenCL kernel.

6.5 Results and discussions

In this work, we implement the *lid-driven cavity* use-case based on the OPAL LBM kernel presented in [23], originally written in OpenCL-C. Multiple-relaxation-time (MRT) collision from [64] is fused with the *pull* propagation and the *half-way bounce-back* boundary condition. The collision kernel performs about 350 floating-point arithmetic operations per lattice update. Four propagation algorithms are implemented for comparison: (1) Two-lattice, (2) AA-pattern, (3) Two-wall and (4) Three-wall. Two-lattice is considered as the state-of-the-art algorithm. AA-pattern is chosen as an efficient algorithm which requires only one instance of the lattice buffer. Then two-wall and three-wall are implemented for comparison to the two former algorithms.

For comparison between the four propagation algorithms, we will use the customary performance metrics in LBM, i.e. million lattice-node updates per second (MLUPS) and the memory occupancy efficiency in terms of LUPS per byte, referred to as the *perf-mem* ratio. The main reason of comparing the perf-mem ratio is to understand the performance efficiency of algorithms on a given amount of memory, in a similar way as GFLOPS per watt account for algorithmic energy efficiency. Results of each implementation are checked for correctness against the reference OPAL code on a GPU.

6.5.1 OpenMP

The OpenMP code runs on two shared memory platforms:

- 24-core 12x2 NUMA CPU Intel Xeon Haswell E5-2680v3 at 2.5 GHz
- 64-core MIC Intel Xeon Phi Knights Landing (KNL) 7230 at 1.30 GHz

It is worth mentioning that KNL embeds 16 GB of on-chip high-bandwidth MCDRAM accessible either as cache or flat memory. In this test, we explicitly allocate lattice buffers directly into the MCDRAM via the *hbwmalloc* library [65]. [1]

For both two-lattice and AA-pattern, the three nested for-loops in z, y, x are organized and collapsed as follow:

```
#pragma omp parallel for private(x,y,z) collapse(2)
for(z = 0; z < Lz; z++)
  for(y = 0; y < Ly; y++)
    for(x = 0; x < Lx; x++)
```

This configuration leaves the possibility to the `icc` compiler of vectorizing the x-loop and gives the best performance in our tests. Regarding the computation code, the LBM kernel of OPAL is rewritten from OpenCL-C to the multi-dimensional array C99 standard. Given the similarity between OpenCL-C and C99, the porting process did not raise much difficulty, which consists mostly in copying the propagation and collision code blocks inside the three nested zyx-loops.

TABLE 6.2: Compilation flags and OpenMP context on CPU and KNL.

Arch.	**Compilation**	**OpenMP env.**

CPU	`icc -O3 -qopenmp -align`
	`-fma -ftz -finline-functions`
		`OMP_NUM_THREADS=24`
		`OMP_PROC_BIND=close`
		`OMP_PLACES=cores`

KNL	`icc -O3 -qopenmp -align`
	`-fma -ftz -finline-functions`
	`-xMIC-AVX512`
		`OMP_NUM_THREADS=64`
		`OMP_PROC_BIND=close`
		`OMP_PLACES=cores`

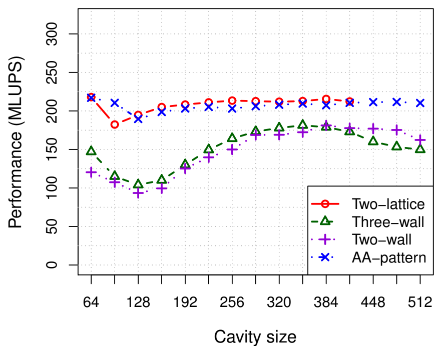
Performance in MLUPS of the four propagation algorithms on CPU is shown in Fig. 6.6a. On CPU, AA-pattern delivers similar performance to the one ot two-lattice algorithm, that shows its benefit over two-lattice in terms of memory requirement. Performance of two-wall and three-wall are between 20 to 40% lower than one of two-lattice and AA-pattern, which conform to $4q$ of data movement per lattice update. As expected, three-wall is faster than two-wall on CPU, but by only 2-9%, certainly thanks to the

[1] The *kona01* node from the PlaFRIM system which is configured with *flat* memory-mode and *quadrant* cluster-mode [66]. On both processors, the compiler version of Intel 2017_update2-knl is used. Compilation flags are given in the Tab. 6.2, by following the recommendations from [66]. Default OpenMP affinity environment variables like number of threads and thread locality are also given the Tab. 6.2. We observe that using the same number of OpenMP threads as the number of physical cores yields better performance, as it avoids additional cost related to thread-switching and cache-pollution. Thread-binding is also carefully chosen to optimize cache locality within NUMA cores on CPU or the shared common L2 cache of a tile (2 cores) on KNL.

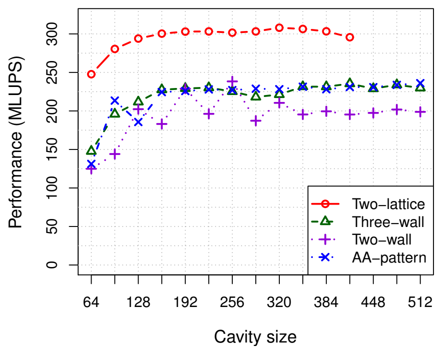
efficient OpenMP runtime on the CPU cache system. However, we observe that two-wall turns to be faster than three-wall from problem sizes larger than 384^3. This is explained by the data size of three-wall buffers that exceeds the L3 cache size of CPU (30 MB) from 384^3 domain: $3 \times (384^2 \times 19 \times 4) \approx 32$ MB. Performance of both two-wall and three-wall follows an oscillating line that fits the L2 and L3 cache, at 128^3 and 384^3 respectively. Performance efficiency of algorithms in terms of LUPS-per-byte is shown in Fig. 6.6c, whose values are normalized to the two-lattice algorithm. The AA-pattern, which consumes twice less memory than two-lattice while delivering the same performance, is two times more efficiency in terms of memory occupation. Besides, three-wall and two-wall, although they do not reach the same efficiency as the AA-pattern, are always better than two-lattice.

Moreover, we observe interesting results on KNL. As seen from Fig. 6.6b, two-lattice considerably outperforms three other algorithms (300 MLUPS versus 225 MLUPS of AA-pattern). This performance gap is justified by the fact that `icc` has succeeded at generating *non-temporal streaming stores* [67] (`movntdq`) for the two-lattice propagation. Instruction generation is confirmed by reading the executable binary of two-lattice with the `objdump` command. This instruction is introduced on Xeon Phi generations for bypassing cache levels when writing data to the memory (the second lattice buffer of two-lattice). This avoids expensive *read-for-ownership* (RFO) operations and saves memory bandwidth on the Xeon Phi architecture, which is known to be more sensitive to cache-pollution because of the write-allocate policy. Interestingly, however, performance of three-wall this time competes with AA-pattern and outperforms two-wall by 15-20% for problem sizes larger than 384^3 ($\approx$ 32 MB of last-level L2 cache). This result is promising because it shows that one can use three-wall as an alternative to AA-pattern to implement complex LBM boundary conditions on the KNL architecture, while obtaining the same performance and perf-mem ratio (see Fig. 6.6d). Performance of two-wall on KNL is not stable on small and medium problem sizes between 64^3 and 384^3 (zigzag line). There are two possible reasons to this phenomenon: (1) important number of inter-thread barriers ($2L_z$) and (2) dependence of memory alignment for two-wall buffers with respect to problem size. These two issues are likely to be amplified by the latency from the directory-based L2 coherence protocol on KNL and would require further investigation to explain the phenomenon.

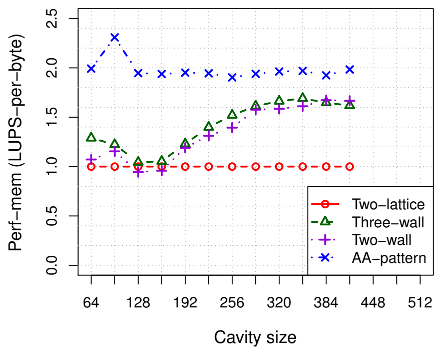
As a result, we prove that three-wall and two-wall can be implemented on a CPU system with higher memory-efficiency than two-lattice. Moreover, using three-wall on CPU does not offer significant gain over two-wall, because the cache coherence protocol can still manage to deliver satisfactory latency and bandwidth in a low-concurrency context (24 cores). However, it is promising to employ three-wall, which outperforms clearly two-wall, on a high-core count architecture like KNL (versus traditional CPU) as



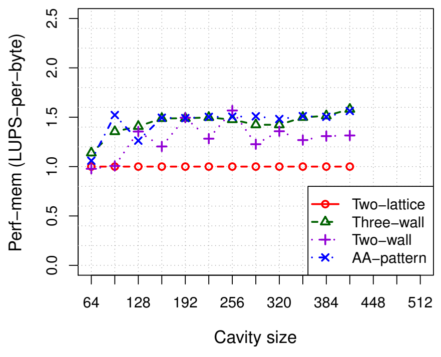
(A) Performance on CPU.



(B) Performance on KNL.



(C) `Perf-mem` ratio on CPU (normalized to two-lattice).



(D) `Perf-mem` ratio on KNL (normalized to two-lattice).

FIGURE 6.6: Comparison of different propagation algorithms in OpenMP on CPU and KNL. T = 1000 time steps.

an alternative to other *one-lattice* algorithms. The barrier cost and OpenMP overhead of three-wall is acceptable in a high-concurrency context like on KNL, with 64 cores and a high-latency cache coherence protocol. Three-wall can also be optimized on the KNL architecture by generating non-temporal streaming stores via OpenMP hints and/or compiler flags, which we expect to yield higher performance than AA-pattern.

6.5.2 OpenCL

The OpenCL code is run on two platforms:

- CPU Intel Xeon Haswell E5-2680v3 (same as the OpenMP code)
- GPU NVIDIA Tesla P100 (Pascal) PCI-E 16GB

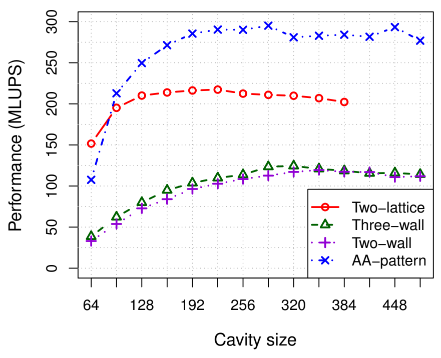
The CPU is used as an OpenCL device by setting the device type to `CL_DEVICE_TYPE_CPU` and similarly `CL_DEVICE_TYPE_GPU` for the GPU. Two-lattice has been already implemented in the original version of OPAL [23]. We took this code as a basis to write kernels for AA-pattern, two-wall and three-wall by adapting the propagation step and the post-collision in-place update of PDFs for each of these three algorithms. OPAL is implemented in a way that one can easily switch the memory layout depending on the target architecture, typically between AoS (favorable for CPU) and SoA (favorable for GPU). OpenCL libraries used in our tests are summarized in Tab. 6.3, in which global and local work-items configuration of kernels are also given. Similar to OpenMP, in OpenCL we can have either 2D or 3D kernels corresponding to the four propagation algorithms (2D for three-wall/two-wall and 3D for two-lattice and AA-pattern). We study performances using 32 as the problem size increment. It should be mentioned that setting OpenCL work-group size to $32 \times 1 \times 1$ already gives satisfactory performance on target platforms. Incrementing work-group size to for instance $64 \times 1 \times 1$ does not yield important gains, while constraints the problem size to be multiple of 64.

Table 6.3: OpenCL drivers and configuration on CPU and GPU.

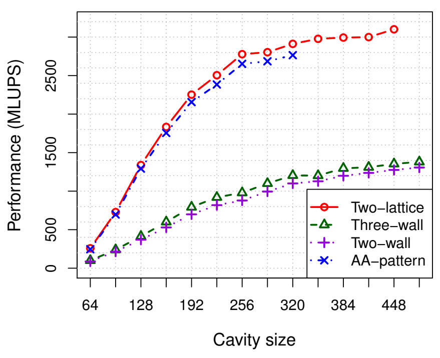
Arch.	**OpenCL driver**	**OpenCL config. in 2D (or 3D)**
CPU	Intel OpenCL 14.2	GlobalSize $L_x \times L_y$ $(\times L_z)$ LocalSize 32×1 $(\times 1)$
GPU	CUDA 7.5.18	GlobalSize $L_x \times L_y$ $(\times L_z)$ LocalSize 32×1 $(\times 1)$

Surprisingly in Fig. 6.7a, despite using the same CPU, the AA-pattern OpenCL code delivers significantly higher performance than the OpenMP version (290 MLUPS versus 210 MLUPS). This represents up to 2.75 times higher in perf-mem efficiency with respect to the normalized two-lattice ratio (see Fig. 6.7c) (versus $\times 2$ in OpenMP). This leads us to think that the Intel OpenCL SDK, by some advanced SIMD optimizations, performs much better than the `icc` compiler on OpenMP `pragma`'s of the AA-pattern propagation. The Intel OpenCL runtime might have detected that AA-pattern always writes data back to the same location of earlier reads and generates the corresponding optimized code. More fine-tuned compilation options for `icc` could be the missing point in our tests. However, determining the corresponding flags or using advanced OpenMP hints and extensions is not the main goal of this work. For the sake of portability and proof-of-correctness of algorithms, we do not include any platform-specific code annotation and only use default compilation flags as recommended in [66]. Moreover, we see that implementing two-/three-wall in OpenCL on CPU is less favorable (125 MLUPS) than directly using OpenMP (180 MLUPS), with respect to the performance and perf-mem ratio. The oscillating line, seen in the OpenMP code, is replaced by a normal lower performance line, probably due to the OpenCL runtime overhead and cache flush after each kernel completion.

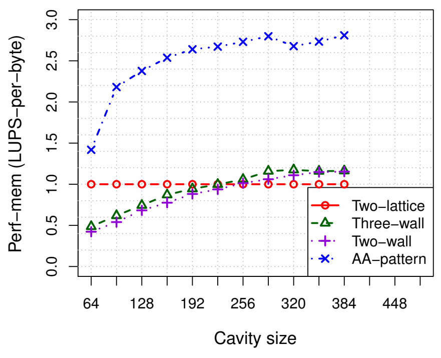
Furthermore, the latest Pascal GPU, with its HBM2 memory, delivers gratifying performance with more than 3000 MLUPS on two-lattice and 2700 MLUPS on AA-pattern, which is 10 times higher than on the CPU. Besides, we obtain about 1300 MLUPS for two-wall and 1380 MLUPS for three-wall, which is 6% better than two-wall (see Fig. 6.7b). For unknown reason, the AA-pattern implementation fails to execute on problem sizes larger than 352^3 and returns an `XID 31` error. According to the NVIDIA documentation, this error is due to an illegal address access. It is possibly caused by the application code, but there could also be driver bugs or hardware bugs. As the issue does not appear on problem sizes smaller than 352^3 with the same code, we suspect a pathological corner-case of the AA-pattern algorithm on the target GPU architecture and/or the current driver version. Nonetheless, the original source of the error remains unknown for the moment. Regarding the LUPS per byte efficiency on GPU, in Fig. 6.7d, the perf-mem ratio of AA-pattern is two times higher than the one of two-lattice (same as in OpenMP). Furthermore, we see that two-wall and three-wall algorithms are less efficient than two-lattice, by their perf-mem ratio between 0.6 and 0.9, normalized to two-lattice.



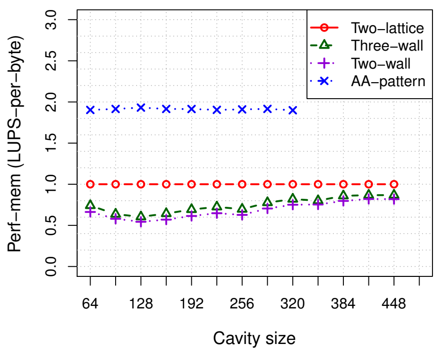
(A) Performance on CPU.



(B) Performance on GPU.



(C) `Perf-mem` ratio on CPU (normalized to two-lattice).



(D) `Perf-mem` ratio on GPU (normalized to two-lattice).

Figure 6.7: Comparison of different propagation algorithms implemented in OpenCL on CPU and GPU. T = 1000 time steps. Memory layout AoS is used on CPU and SoA is used on GPU.

From these results, we conclude that two-/three-wall can also be easily implemented from an existing two-lattice kernel in OpenCL. While running correctly on CPU or GPU in OpenCL, two-/three-wall do not offer significant performance gain nor memory-efficiency compared to two-lattice and AA-pattern. In our testing environment, we notice execution failure of the AA-pattern algorithm for problem sizes larger than 352^3 on the Tesla P100 GPU, which can be a drawback in applying AA-pattern on GPU-based platforms.

6.6 Conclusions

In this chapter, we introduce two-wall and three-wall, two novel algorithms of the LBM *one-lattice* algorithm class which use a single lattice instance. Two-wall and three-wall are easy to implement and can be used on complex boundary conditions.

Our algorithms do not deliver the highest performance on CPU and GPU compared to other algorithms, but offer better memory efficiency (LUPS per byte) than the state-of-the-art two-lattice algorithm. Especially on a high-core-count shared-memory architecture like KNL, three-wall gives competitive performance and memory efficiency compared to AA-pattern, one of the most efficient propagation algorithms in literature. Employing three-wall on future many-core architectures which embed more local memory and private cache as well as the non-temporal streaming store instruction could lead to further performance improvements. Furthermore, hardware architecture in near future is likely to favor large on-chip coherent memory associated to many-core designs. For such evolutions, our approach will become mandatory to increase performance with a straightforward OpenMP implementation. Using a single lattice instance also allows to increase spatial resolution and thus to obtain more reliable solutions of the physical problem under consideration.

Chapter 7

Message Passing Interface (MPI) on Many-core Processors

The single biggest problem in communication is the illusion that it has taken place.
– George Bernard Shaw.

7.1 Introduction

In this chapter, we propose the design of an MPI Message-Passing library [9] for the intra communication on many-core processors, using the vendor support library (MP-PAIPC [68]) as the transfer-fabric to build MPI protocols from scratch, while porting any of existing MPI implementations such as MPICH or OpenMPI would not be possible due to limited on-chip memory of most recent many-core processors.

Based on studied MPPA hardware specifications presented in [31, 68], this chapter does a brief hardware summary and focuses on an MPI design over (but not limited to) the MPPA architecture, with detailed implementation algorithms and formulated models following vendor-hardware characteristics (K, h) and different optimizing approaches (Lazy, Eager). These studies are generic enough to be compared/ported to other very-similar architectures, such as Tilera [69], STHORM [70] or Neo chip [71], on which doing/optimizing MPI communication over Network-on-chip is still a challenging or never-posed question.

The remainder of this chapter is organized as follows: Related MPI-oriented works on other many-core platforms are compared in section 7.2. Section 7.3 describes our MPI architecture design. Section 7.4 resumes our MPI implementation in pseudo-codes of blocking and non-blocking communication (`MPI_Send` and `MPI_Isend`). Some optimization ideas are then proposed and developed in this section such as (1) synchronization-free *eager send* and (2) implicit local-buffered *lazy send* for short and medium sized messages respectively. A throughput estimation model based on the data transmission time is also introduced in section 7.5 to evaluate the communication performance. Section 7.6 presents our results for the ping-pong test following two scenarios, either symmetric ranks (MPI compute node - MPI compute node) or asymmetric ranks (MPI compute node - MPI I/O), corresponding on MPPA to CC-CC and CC-I/O subsystem respectively. Different optimization approaches are also tested and compared.

Using the MPPA-MPI library, the HPL benchmark [38, 39] was ported on MPPA with the support of the standard BLAS-Netlib [72, 73] (mono-threaded) and OpenBLAS [74], an OpenMP optimized implementation. These benchmark results are summarized in section 7.7 and conclusions are given in section 7.8.

7.2 Related works

Our design is similar to the *co-processor-only* MPI model on the Intel Xeon-Phi platform [75], with support of OpenMP for hybrid multi-threaded programming. Besides

MPI ranks running on CC, we introduce an MPI I/O rank running on an I/O subsystem of the chip as bridge to communicate with the host through the PCI-e interface, while there are no direct communication link between the host and the compute clusters on MPPA. Along the way, some collective MPI functions were also implemented (`MPI_Comm_split`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce` and `MPI_Barrier`).

Our message-trigger handling mechanism using the RM core was inspired by the similar work of Prylli and Tourancheau [76] [77] implementing the BIP protocol for an optimized MPI implementation over the Myrinet network, taking advantage of its dedicated hardware, an extra core like the MPPA RM core.

Today, there exist other multi-/many-core processors similar to the MPPA. Some of them has an MPI implementation, others do not. This section reviews MPI-oriented libraries on other many-core architectures and their performance related to our work.

Raw processor [78], designed by the Computer Science Laboratory at MIT, combines 16 identical compute units, called tile. The 16 tiles are connected by one static NoC and two dynamic NoC. The static network is used for predefined memory access pattern at compile time, the dynamic ones are used for communication scheme at runtime. Psota and James [79] propose rMPI, the first MPI library over the Raw achitecture by inheriting some design aspects from MPICH and LAM/MPI also other specific implementation belonging to the Raw hardware. The highest throughput obtained on the ping-pong test of the Raw processor is about 150 MB/s with buffer size of 3.2 MB (100K words) [80].

Tilera processors [69] are mainly used in high performance embedded systems such as networking and multimedia. The TilePro64 processor defines a flat 2D-mesh with 64 identical VLIW cores connected through the Tilera iMeshTM network-on-chip. Cache coherence on TilePro64 is guaranteed by a hardware mechanism called Dynamic Distributed Cache (DDC) [69]. Kang et al. [81] propose an MPI implementation on Tile64 processor which delivers up to 250 MB/s on `MPI_Send/MPI_Recv` communication, with the largest message size of 256 KB due to the limited memory per core. At this buffer size, our MPI implementation on MPPA delivers 400 MB/s on `MPI_Send` or up to 1 GB/s using DMA.

Intel Single-Chip Cloud (SCC) is a prototype aimed to promote many-core processor. Its 48 cores are organized in 24 dual-core clusters with access to off-chip DRAM shared/private region for all/each core through a look-up table (LUT), also a dedicated shared on-chip Message Passing Buffer (MPB). This memory architecture gives extra advantage for implementing quick message sending based on shared buffers. However, the use of the dynamic NoC routing on Intel SCC (instead of static NoC in MPPA) makes it difficult to evaluate the maximum communication latency [82] also incurs unordered packets, hence inappropriate for hard real-time applications.

The SCC-specific MPI-like native communication library (RCCE) delivers peak throughput of 55 MB/s on the ping-pong test [83]. By the same test, Clauss et al. [84] presented iRCCE (an improved RCCE version) and SCC-MPICH (an MPICH-based implementation over iRCCE) that reach respectively 150 MB/s and 120 MB/s of throughput. RCKMPI, an Intel MPI implementation for SCC is also bounded by the performance of the iRCCE layer. Our MPI library on MPPA was built from scratch over the MPPAIPC library, without any TCP/UDP layer, while an MPICH-based solution would not fit the cluster private memory space (2 MB). A such MPICH implementation on MPPA can be extrapolated to 1.0 GB/s by adding the same overhead of 20% of SCC-MPICH over to iRCCE.

Intel Many Integrated Core (MIC), known as Intel Xeon Phi co-processor family, is a x86-based many-core architecture with native support of Linux operating system and standard software stack. The first Intel MIC generation, namely Knights Corner (KNC), proposes three MPI programming models [75] which are (1) offload (host-only), (2) co-processor-only and (3) symmetric (both host and co-processor). The MPI communication in the intra-MPPA context corresponds to the co-processor-only intra-MIC case. Potluri et al. [85] studied the communication throughput of the MVAPICH2 library on KNC and their results show that a MIC-optimized MVAPICH2 library can delivers more than 9 GB/s of uni-direction throughput for messages up to 1 MB.

7.3 MPPA-MPI design

In the MPPA context, each CC is referred to as an MPI rank. Thus, the MPPA-256 processor supports up to 16 MPI ranks. Each MPI rank owns a private memory space of 2MB. Moreover, a hybrid MPI I/O rank is introduced running on the North IOS and manages the off-chip DDR memory. This MPI I/O rank is started from the host via the `k1-mpirun` command and is responsible for spawning MPI compute ranks on CCs subsequently. To keep the portability of any MPI legacy code, this extra MPI I/O rank is not listed in the `MPI_COMM_WORLD`. Any communication with this rank can be achieved through a *local communicator* (`MPI_COMM_LOCAL`) that groups all MPI ranks within an MPPA processor (i.e. 17 ranks). The MPPA-MPI architecture on each rank is then divided in two layers:

7.3.1 MPI-inter-process Control (MPIC)

Each MPI transaction begins by exchanging control messages at the MPIC layer between MPI ranks. Control messages are used for:

- information exchange about MPI transaction type (send/receive, communicator split, etc.).
- synchronization point in case of rendez-vous protocol.

We implement an RQueue-based active message server [86] on each MPI rank (CC and IOS) to handle incoming control messages from all other ranks (including itself on loop-back). Upon control message arrival, a callback function is executed on the RM, consisting typically on saving it into an internal buffer which later will be read by MPI calls from the `main` function (PE0).

Control messages exchanged in the MPIC layer contain either one of the structures defined in Fig. 7.1.

```
/* Message sent by Tx to Rx (Request-To-Send) */
typedef struct send_post_s {
  mppa_pid_t    sender_id;       /* ID of Tx process */
  int           mpi_tag;         /* MPI message tag */
  ...
} send_post_t;

/* Message sent by Rx to Tx (Clear-To-Send) */
typedef struct recv_post_s {
  int           dnoc_tag;        /* DNoc allocated on Rx */
  mppa_pid_t    reader_id;       /* ID of Rx process */
  int           mpi_tag;         /* MPI message tag */
  ...
} recv_post_t;
```

FIGURE 7.1: Control message structures

In an MPI send/receive, The Tx rank posts a *Request-To-Send* (`send_post_t`) to the Rx rank; idem, the Rx rank sends back a *Clear-To-Send* (`recv_post_t`) containing its allocated `dnoc_tag`, to which the Tx rank will send data. Beforehand, this `dnoc_tag` needs to be configured and linked to the receive buffer to enable remote writing.

7.3.2 MPI-inter-process Data-Transfer (MPIDT)

MPIDT is a light-weight wrapper of MPPAIPC *Portal* primitives. Once the Tx rank has got a matching control message, it configures a data transfer using received information (e.g. `dnoc_tag`). Data can then be sent in either blocking or non-blocking mode dependent on the calling MPI function, using appropriate *Portal* primitives (see Tab. 7.1 for detailed function mapping).

TABLE 7.1: MPI send/receive implementation in MPIDT level

MPPA-MPI	**MPPAIPC Portal**
`MPI_Send`, `MPI_Ssend`	`mppa_pwrite`, `mppa_pwrites`
`MPI_Isend`, `MPI_Issend`	`mppa_aio_write`
`MPI_Recv`, `MPI_Irecv`	`mppa_aio_read`
`MPI_Wait`	`mppa_aio_wait`

Fig. 7.2 illustrates the structure of our MPPA-MPI implementation. Each rank emits control-message to its involved partner at each MPI call. The active server runs on the RM core and processes incoming control-messages. Furthermore, depending on MPI transactions and their status at runtime, the server can decide whether to perform a data send if this has not been or could not be done by the `main` thread, especially in case of a pending `MPI_Isend` request or a matching registered lazy message.

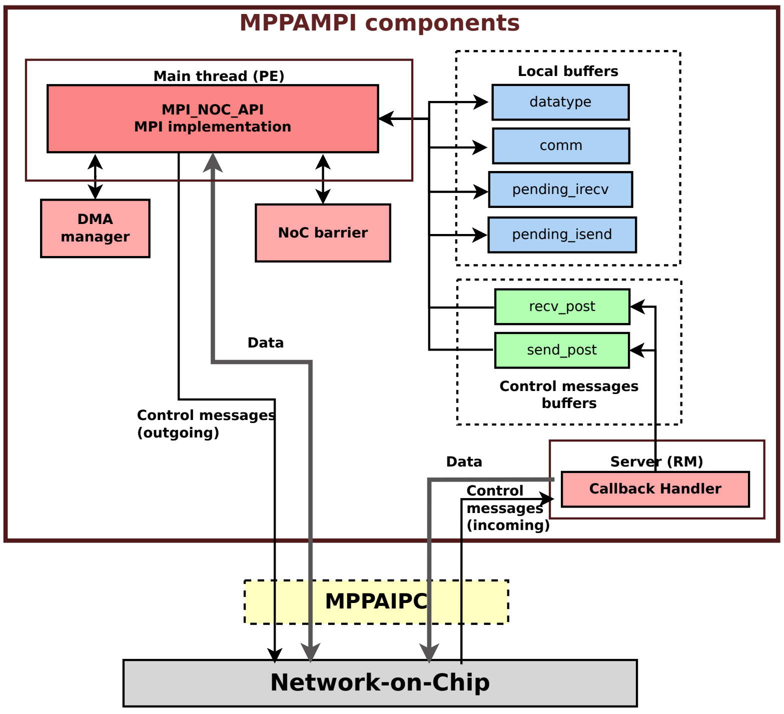


FIGURE 7.2: MPPA-MPI components and interaction with Network-on-chip through MPPAIPC.

7.4 MPPA-MPI implementation

As mentioned above, on-flight control messages carry essential information depending on their purpose. We present now their usage as well as algorithms of the two communication scenarios in our work:
(1) synchronous blocking send (`MPI_Send`) and
(2) asynchronous non-blocking send (`MPI_Isend`)

7.4.1 MPI_Send - MPI_Recv

Most well-known and optimized MPI libraries contain many (combined) techniques to perform the `MPI_Send` call. In the first time, we chose to implement this function with rendez-vous blocking behavior, in order to avoid extra buffer space and minimize memory usage. This choice certainly adds more synchronization cost but does not change the functionality of the send/receive transaction. Some optimization approaches will be presented in the coming sections. Algorithms 2 and 3 summarize the implementation of `MPI_Send` and `MPI_Recv`.

Algorithm 2 MPI_Recv(buf, count, datatype, source, tag, comm, status)

```
1:  my_rank ← get_rank(comm);
2:  dnoc_tag ← allocate_dnoc_tag();
3:  /* configure to receive data on this dnoc_tag */
4:  aio_request ← configure_aio_read(buf, dnoc_tag, ...);
5:  if source == MPI_ANY_SOURCE then
6:      send_post ← find_send_post(count, datatype, tag, ...);
7:      real_source ← send_post.source;
8:  else
9:      real_source ← source;
10: end if
11: /* send recv_post to real_source (MPIC layer) */
12: send_recv_post(my_rank, dnoc_tag, real_source, tag, comm, ...);
13: /* wait data (MPIDT layer) */
14: mppa_aio_wait(aio_request);
15: return MPI_SUCCESS;
```

The implementation of `MPI_Irecv` is the same as the one of `MPI_Recv`, except that the function returns right after having posted the receive to the sender, and the completion of reading (`mppa_aio_wait`) is done in `MPI_Wait`.

Algorithm 3 MPI_Send(buf, count, datatype, dest, tag, comm)

```
1: my_rank ← get_rank(comm);
2: /* send send_post to dest */
3: send_send_post(my_rank, dest, tag, comm, ...);
4: /* wait for matching recv_post from Rx (MPIC layer) */
5: repeat
6:     recv_post ← find_recv_post(count, datatype, tag, ...);
7: until recv_post ≠ NULL
8: /* send data (MPIDT layer), using mppa_pwrite(s) */
9: send_data(buf, count, datatype, recv_post.dnoc_tag);
10: return MPI_SUCCESS;
```

7.4.2 MPI_Isend - MPI_Recv

The implementation of `MPI_Isend` uses non-blocking *Portal* primitives on both PE and RM on the Tx side. When the Tx rank (PE0) reaches `MPI_Isend` in its execution *without* having received any matching `recv_post`, it creates a *non-started* `pending_isend` request containing related information (buffer pointer, dest, count, tag etc.) and returns. On arrival of the matching `recv_post`, the RM core (callback handler) reads the previous `pending_isend` request and triggers a non-blocking data send (to the `recv_post.dnoc_tag` of the Rx rank). The request is then set to started state to be distinguished from other non-started requests.

On the other hand, when the `recv_post` arrives before `MPI_Isend`, the RM core saves it into the internal buffer. The PE core executing `MPI_Isend` later reads this `recv_post`, performs a non-blocking send and marks the `pending_isend` request as started. This propriety ensures that the transfer is performed only once for each transaction, either by the PE core (in `MPI_Isend`) or by the RM core (in callback handler). At the end, started requests will be finished and cleaned by `MPI_Wait`. Algorithms 4 and 5 present in more details the implementation of `MPI_Isend` and of the callback handler.

7.4.3 Optimization

7.4.3.1 Eager send optimization

The idea is to pack any MPI message which can fit into a 120-byte space, as a control-message and send it directly to the Rx active server. In reality, the maximum data payload is about 96 bytes (24 bytes is used for control header). An `eager_buffer` needs to be allocated on each MPI rank and can be defined by the `EAGER_BUFFER_LENGTH` macro in `main.c`. This approach is synchronization-free when the `MPI_Send` call can

Algorithm 4 MPI_Isend(buf, count, datatype, dest, tag, comm, request)

```
1: my_rank ← get_rank(comm);
2: /* send send_post to dest */
3: send_send_post(my_rank, dest, tag, comm, ...);
4: req ← new_request(buf, count, ..., PENDING_ISEND);
5: /* look for a matching recv_post (MPIC layer) */
6: recv_post ← find_recv_post(count, datatype, tag, ...);
7: if recv_post ≠ NULL then
8:     /* configure/start a non-blocking write (MPIDT layer) */
9:     aio_request ← configure_aio_write (buf,  recv_post.dnoc_tag, ...);
10:    req→status := STARTED;
11:    req→aio_request := aio_request;
12: else
13:    /* Do nothing (request initialized NON_STARTED) */
14: end if
15: request ← req;
16: return  MPI_SUCCESS;
```

Algorithm 5 callback_recv_post(recv_post)

```
1: /* look for a matching pending_isend */
2: req ← find_pending_isend(recv_post);
3: if req ≠ NULL then
4:     /* configure/start a non-blocking write (MPIDT layer) */
5:     aio_request ← configure_aio_write (req→buf,  recv_post.dnoc_tag, ...);
6:     req→status := STARTED;
7:     req→aio_request := aio_request;
8: else
9:     save_recv_post(recv_post);
10: end if
11: return  ;
```

return before a matching receive is posted (non-local). It also leads to an improvement of about 6 to 10% in performance for the HPL benchmark on MPPA (see Section 7.7).

For longer messages, using several eager sends introduces segmentation and reassembly costs. A test case is implemented with messages split into eager packets in order to determine the best communication trade-off in Fig. 7.4. Such segmentation however consumes buffering memory and therefore increases RM workload.

7.4.3.2 Lazy send optimization

Lazy send consists in copying medium-size message into a local buffer and returns. The RM is then responsible for sending it to the destination. Unlike `eager_buffer` on the Rx side, `lazy_buffer` is allocated on the Tx side and can be tuned via some macros (`LAZY_THRESHOLD`, `LAZY_BUFFER_LENGTH`).

This approach must be used with care because bad communication scheduling may lead to buffer wasting and lazy messages remaining for too long. Inversely, a dense communication scheme should neither be set to lazy mode in order to be able to send data directly rather than spending time doing `memcpy` in local memory.

7.4.3.3 DMA thread usage

`MPI_Isend` uses *Portal* non-blocking primitive to configure a Tx DMA thread for data sending. The DMA engine implements a `fetch` instruction that loads the next cache line while pushing the current line into the NoC. This `fetch` is nowadays not available on PE cores, meaning that outbound throughput using PE is 4 times lower than using DMA engine (1 B/cycle vs. 4 B/cycle). Thus, tuning to use non-blocking DMA on `MPI_Send` for messages of size greater or equal to `DMA_THRESHOLD` will maximize the transfer performance.

7.5 MPPA-MPI Throughput modeling

The MPPA-256 Network-on-chip [87] is designed so that any path linking two CCs always contains less than eight hops (including two local hops - one at sender and one at receiver). The average switching time on a NoC router is 7 cycles, then it takes the packet at most 8 cycles to reach the next hop. In the worst case, the link distance (time a packet spends on NoC to reach its destination) is 112 cycles ($7\times8+8\times(8-1)$). However, the necessary time to send a buffer (transmission time - t) is about $\mathcal{O}(N)$ cycles [88] (where N the buffer size in bytes), which is much longer than the link distance [89].

As a result, we describe the transmission time t as a function of the buffer size N, a constant transfer ratio K and a default overhead h (aka. the cost of sending an empty buffer). This default overhead presents the initial cost of MPI implementation management (ID mapping, metadata setup, synchronization, error checking ...) and/or configuring the peripherals (cache, DMA) to prepare for data sending. This cost is paid on each MPI call and is independant to the subsequent data-sending process (which is presented by a data-transfer factor K). The ping-pong round-trip time (RTT) is approximately the sum of the transmission time on both sides, as the propagation time is negligible.

$$TransmissionTime: \quad t = K \times N + h \; (cycles) \tag{7.1}$$

$$RTT \approx 2 \times t = 2 \times (K \times N + h) \; (cycles) \tag{7.2}$$

$$Throughput : T = \frac{2 \times N}{RTT} \approx \frac{N}{K \times N + h} = \frac{1}{K + \frac{h}{N}} \quad (bytes/cycle) \tag{7.3}$$

$$\begin{aligned} \lim_{N \to \infty} T \approx \lim_{N \to \infty} \frac{1}{K + \frac{h}{N}} &= \frac{1}{K} \ (bytes/cycle) \\ &= 400 \times K^{-1} \ (MB/s) \\ &(at\ frequency\ 400\ MHz) \end{aligned} \tag{7.4}$$

The constant K is a value specific to each send function with its own underlying transport primitive. For example, the `MPI_Isend` which uses the DMA engine with peak throughput of 4 B/cycle, would have its transfer ratio K of about 0.25. The `MPI_Send`, with default peak throughput of 1 B/cycle (no DMA engine), should obtain a transfer ratio K around 1.

7.6 Results and Discussion

Using the MPPA Developer platform [90] with the first-generation MPPA processor, we set up ping-pong tests between:
(1) MPI rank 0 (CC_0) - MPI rank 15 (CC_15) and
(2) MPI I/O 128 (IOS_128) - MPI rank 15 (CC_15).
All MPI cluster ranks run at the same clock frequency of 400 MHz. The North IOS running the MPI I/O rank is configured to use the DDR controller at the default frequency of 600 MHz.

In each case, the same MPI send function is used on both sides (`MPI_Send` or `MPI_Isend`). At the first time, all tests are run without any optimization in order to calibrate the proper throughput of each context (Fig. 7.3). At the second time, we enable all optimization on the `MPI_Send` test and compare our optimization approaches in terms of latency, throughput and messages sent per second (Fig. 7.4).

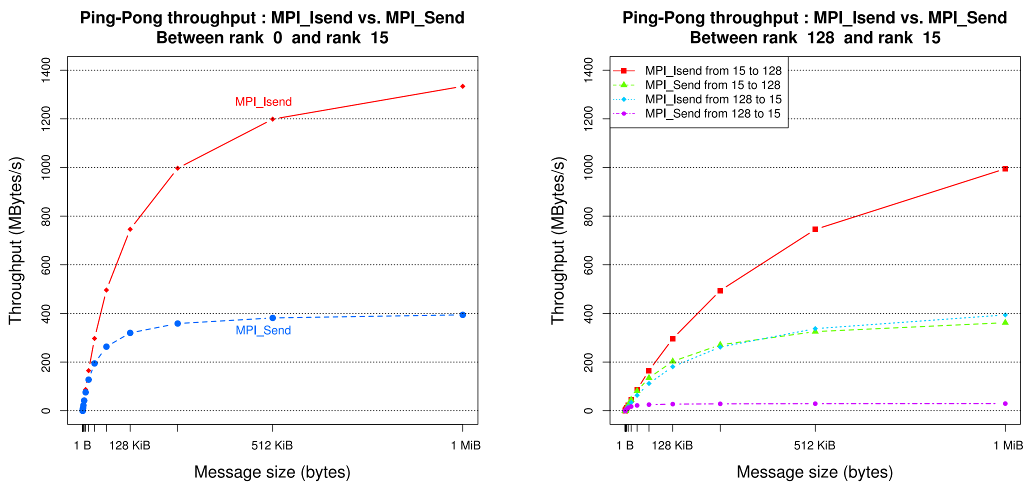
Each ping-pong is repeated 50 times. We assume that there is no waiting time inside the MPI send function, since all ranks start at the same time and run at the same clock speed. Hence, the duration of the MPI send function can be considered as the transmission time. Depending on the send context, the measured transmission time is fitted into a linear correlation $K \times N + h$ presented in Tab. 7.2. The standard deviation from all obtained results is always less than 0.2%.

From	**To**	**MPI_Send**	**MPI_Isend**
CC_0	CC_15	$t = 0.98 \times N + 31430$	$t = 0.27 \times N + 33690$
CC_15	CC_0	$t = 0.98 \times N + 30240$	$t = 0.27 \times N + 32850$
IOS_128	CC_15	$t = 13.52 \times N + 159544$	$t = 0.84 \times N + 181300$
CC_15	IOS_128	$t = 0.98 \times N + 129200$	$t = 0.26 \times N + 144500$

Table 7.2: Transmission time (cycles).

7.6.1 Inter-CC communication

Communication links between CCs are bi-directionally symmetric. According to our model and the K values from Tab. 7.2, the estimated maximum throughput (given by $400 \times K^{-1}$ MB/s) should be around 408 MB/s and 1481 MB/s for `MPI_Send` and `MPI_Isend` respectively. The ratio h/N can be ignored in this case. Fig. 7.3a shows obtained results that match with our estimation model.



(A) Symmetric : Between ranks 0 and 15 (B) Asymmetric : Between ranks 128 and 15

Figure 7.3: Ping-pong throughput `MPI_Send` (PE core) vs. `MPI_Isend` (DMA).

7.6.2 CC-IOS communication

Contrary to the symmetric communication performance between CCs, the transmission rate on I/O subsystem relies on the DDR bandwidth, which is much lower than the on-chip memory on CCs. We observe higher K values and much more considerable overhead h on the IOS_128, showing that the communication link from IOS to CCs might be the bottleneck on the MPPA. It is then difficult ignoring $\frac{h}{N}$ in this case. Keeping on our throughput estimation by $400 \times \left(K + \frac{h}{N}\right)^{-1}$ now matches with experiment results on Fig. 7.3b, where the performance gap between the CC_15 and the IOS_128 is also illustrated.

7.6.3 Optimization comparison

We focus now on finding, on a given message size, the best send method among the four (Normal, Eager, Lazy and DMA) to use on `MPI_Send`, in order to obtain lowest latency (round-trip-time) and/or highest ping-pong throughput, by enabling all optimizations and re-running our experiments between CCs. We also evaluate the number of messages sent per second in each approach by dividing the clock frequency (400 MHz) by the duration of the `MPI_Send` call (in cycles). As the message will now be eagerly sent or lazily buffered and `MPI_Send` returns right afterward, this duration on Eager(-splitting) or Lazy could no longer be evaluated as the transmission time in the Tab. 7.2, but respectively by :

$$E \times \left(\left\lfloor \frac{N}{96} \right\rfloor + 1 \right) \ (cycles, E \approx 3800) \tag{7.5}$$

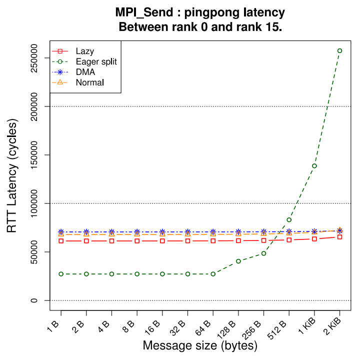
$$\mathcal{O}_{memcpy}(N) = 1.28 \times N + 5300 \ (cycles) \tag{7.6}$$

where E is the constant necessary cost to send 1 eager-split and O_{memcpy} is a linear function of `memcpy` cost. Note that in the Lazy approach, the message is sent in background by the RM.

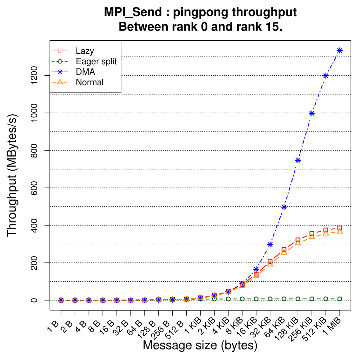
Hence, Eager and Lazy methods provide lower latency and higher message rate on short buffers, since they were designed to get rid of two-sided synchronization and the buffer size is still small enough not to be outperformed by the DMA high-throughput capacity.

Fig. 7.4a shows that the ping-pong latency from 1 to 256 bytes using eager-splitting is reduced by half compared to DMA or Normal. Otherwise, this latency increases radically as soon as its transmission time, despite being smaller at the beginning, getting repeated as many times as split segmentation $\left(\left\lfloor \frac{N}{96} \right\rfloor + 1 \right)$. On the other hand, using DMA on large buffers optimizes bandwidth utilization compared to Normal (using PE) or Lazy (using RM) methods. (Fig. 7.4b).

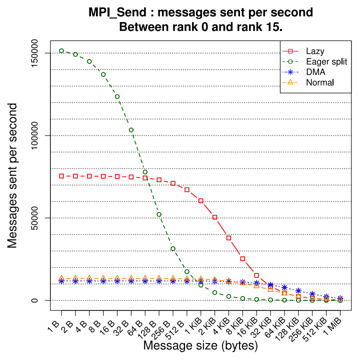
Fig. 7.4c illustrates the message-rate of the four send methods. Not only this kind of measure gives user a high-level point of view about the implementation capacity to support communication load, but it shows interesting advantages of Eager and Lazy methods in tuning MPI applications, thanks to their fast sending time for short messages and synchronization-free algorithm.



(A) Ping-pong latency (RTT) on short buffers



(B) Ping-pong throughput

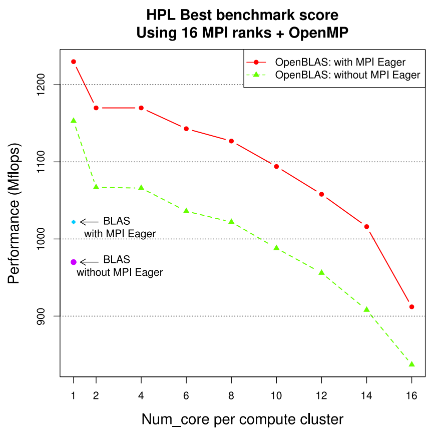


(C) Number of messages sent per second

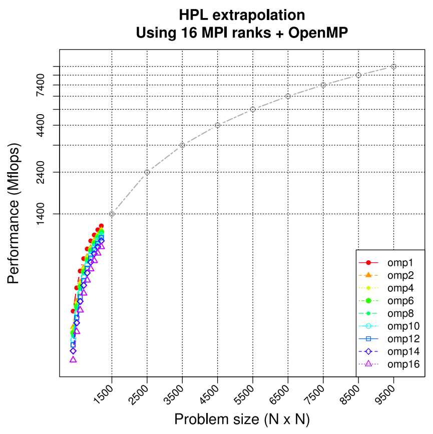
FIGURE 7.4: Optimization approaches comparison.

7.7 High Performance Linpack (HPL) on MPPA-256

HPL benchmark was ported on MPPA-256 using our MPI implementation and cross-compiling of BLAS-Netlib and OpenBLAS. Each MPI compute rank, assigned to a compute cluster, only owns 2 MB of memory, which amounts to a total on-chip memory of 32 MB, enable to store up to 4 million double precision floating-point numbers or a 2000×2000 matrix. Operating system space and user code (BLAS, MPI, HPL) must be taken into account as well. In practice, the HPL can run on the MPPA-256 with 1250×1250 matrix, which is a very small problem size for this kind of benchmark. As a result, communication, local indexing etc. has a significant cost with respect to the number of floating point operations ($\mathcal{O}(N^3)$). Fig. 7.5b. does an estimation on further problem sizes on future MPPA generations with more on-chip memory.



(A) HPL benchmark on MPPA-256 against number of cores: BLAS vs. OpenBLAS.



(B) HPL score extrapolation with increasing problem size.

FIGURE 7.5: HPL current performance (a) and extrapolation (b)

Fig. 7.5a shows the HPL result on MPPA-256 using BLAS and OpenBLAS. Note that 70 GFLOPS of annouced theoretical performance is for all the 256 cores, while the best benchmark score (R_{max} = 1.2 GFLOPS) was achieved using only one core per CC (i.e. 16 cores in total) and MPI eager send. Also, we have seen no performance change by enabling MPI lazy optimization. This can be explained by well-scheduled HPL overlapping [39] in which, either MPI processes arrive to the communication step at the same time, or all heavy sends are done asynchronously by `MPI_Isend`, while lazy sending only shows its advantage in bad-scheduled `MPI_Send`. Furthermore, multi-threading on MPI compute ranks (OpenMP on CCs) did never give better HPL result, because of the small working set and the OpenMP overhead.

7.8 Limitations and conclusions

In this chapter, we have introduced the design and performance issues of an MPI implementation on the Kalray MPPA-256. The MPPA-MPI library provides 1.2 GB/s of throughput for any inter compute-cluster point-to-point communication and this performance depends on the underlying MPPAIPC library. Optimization ideas such as eager send and lazy message are proposed, implemented and compared to determine the best approach based on message size. A synthetic model is also presented for each approach to evaluate their communication latency and throughput. We also learn that supporting MPI programming model is not an easy task on recent many-core processors, including MPPA, since MPI has become a large API with high-level abstractions and many-core hardware is taking more diversity and complexity. Thus, optimizing an MPI implementation on each of these platforms is even more not trivial.

Despite the fact that the HPL benchmark was also successfully ported on MPPA as a validation test of our MPI library, we think that mapping an MPI rank on a single compute cluster of MPPA is not a long-term solution. This topology mapping can provide some ease of programming in the short-term, but will encounter a scalability issue in the future when we will want to connect hundreds or thousands of MPPAs together. The granularity will be too fine and such mapping will face a significant communication overhead. A more coarse-grained mapping, for instance, one MPI rank per MPPA would be a versatile solution, with the larger DDR space, PCIe and Ethernet interfaces. Each MPI rank then can be accelerated by the 16 compute clusters via OpenMP 4 or OpenCL in the MPI+X hybrid model.

Chapter 8

General Matrix Multiplication (GEMM) on Many-core Processors

The best performance improvement is the transition from the non-working state to the working state.
– John Ousterhout.

8.1 Introduction

The largest difference of a DMA-based many-core architecture from other conventional platforms is the lack of a data cache system. On a DMA-based architecture, all data must be accessed and copied from the off-chip global memory to the on-chip local memory. This not only improves data locality and latency, but also allows computation-communication overlap by asynchronous transfers.

Our objective in the present chapter is to implement a fast GEMM on DMA-based many-core processors, typically the MPPA2-256. Consequently, we are focusing on asynchronous algorithms to reduce memory latency and an MPPA assembly micro-kernel to obtain the highest performance. Therefore, we do not plan to support a full range of GEMM parameters from the BLAS API in this chapter.

8.2 GEMM in POSIX-C

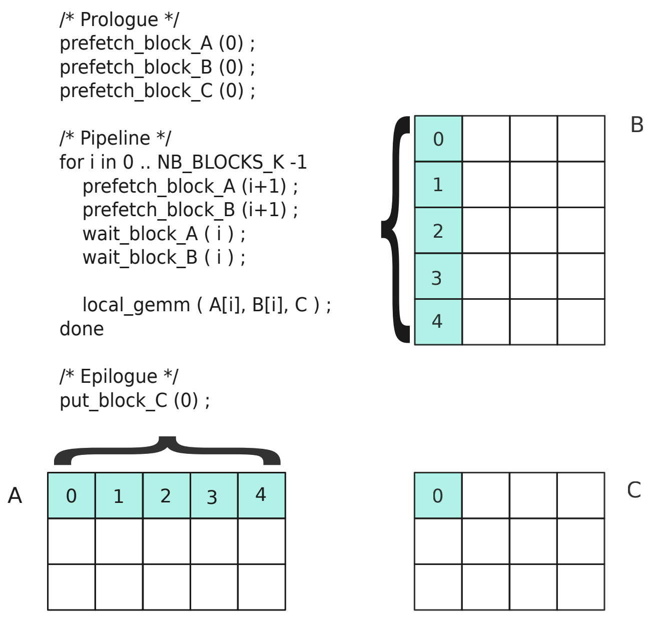
8.2.1 Algorithm

In this work, we use blocksize naming convention from Matsumoto et al. [91], which implement an auto-tuning blocked `GEMM` implementation on OpenCL devices, based on configurable runtime parameters. The matrix C is divided in blocks of size $M_{wg} \times N_{wg}$ (C_i). Blocks of A are partitioned as $M_{wg} \times K_{wg}$ (A_i), and blocks of B as $K_{wg} \times N_{wg}$ (B_i).

On MPPA2, the two-layer memory configuration matches. A large and slow DDR memory versus a small and fast scratchpad memory on each Compute Cluster (CC). Computing a block C_i requires multiplying a block-row of A_i to a block-column of B_i (see Fig. 8.1). Each block-row and block-column contains a same number of $\left\lceil \frac{K}{K_{wg}} \right\rceil$ A_i or B_i blocks.[1] Our DMA-based GEMM algorithm, called GEMM-async, is given in Fig. 8.1. Each block C_i is computed by one CC, equipped with 1.5 MiB of local memory (scratchpad) and 16 PEs. There are totally $L = \left\lceil \frac{M}{M_{wg}} \right\rceil \times \left\lceil \frac{N}{N_{wg}} \right\rceil$ blocks C to be distributed over 16 CCs. Each CC is responsible for at least $\left\lfloor \frac{L}{16} \right\rfloor$ blocks C. Whenever L is not multiple of 16, certain clusters must process one more block to fulfill remaining blocks ($L \mod 16$).

To compute a block C_i within each cluster, blocks A_i and B_i are streamed from the DDR to scratchpad in the *double-buffering* overlapping scheme. At any time, there is always

[1] The `ceil()` function rounds up to the next integer to deal with trailing matrix elements after decomposition.



```
/* Prologue */
prefetch_block_A (0) ;
prefetch_block_B (0) ;
prefetch_block_C (0) ;

/* Pipeline */
for i in 0 .. NB_BLOCKS_K -1
    prefetch_block_A (i+1) ;
    prefetch_block_B (i+1) ;
    wait_block_A ( i ) ;
    wait_block_B ( i ) ;

    local_gemm ( A[i], B[i], C ) ;
done

/* Epilogue */
put_block_C (0) ;
```

FIGURE 8.1: Streamed-tiled GEMM algorithm.

two blocks in prefetch by DMA (one next A_i and one next B_i), and other three blocks in computation (current A_i, B_i and C_i). Blocksize parameters (M_{wg}, N_{wg} and K_{wg}) are consequently set to $M_{wg} = N_{wg} = K_{wg} = 256$ in FP32 and $M_{wg} = N_{wg} = 256$, $K_{wg} = 128$ in FP64, so that the scratchpad memory can hold at least five blocks: two A_i, two B_i and one C_i. We note that, in our algorithm, both single-buffering and double-buffering were implemented on blocks C_i. The communication cost on C_i (read at the beginning and write at the end), even being on the critical path with the single-buffering scheme (not overlapped), can be amortized by flops computation from blocks A_i and B_i when K is large ($\left\lfloor \frac{L}{16} \right\rfloor \times 2$ versus $\left\lfloor \frac{L}{16} \right\rfloor \times \left\lceil \frac{K}{K_{wg}} \right\rceil \times 2$).

On edge blocks [2], since the matrix data does not entirely fulfill the block, we perform *zero-padding* on the remaining part (right and bottom) of the edge block, so that the block-computation kernel can be reused without change or polluting results.

In order to facilitate development process and to focus on algorithmic aspects, we designed and implemented a set of two- and three-dimensional asynchronous copy functions and integrated them into the Kalray low-level asynchronous communication library, so-called `mppa_async`. Conception details of these 2D and 3D functions are presented in Chapter 5, Section 5.3 (page 38)

[2]An edge block is the block added by the `ceil()` function, often at the right and bottom of the decomposition grid, to cover trailing matrix elements.

8.2.2 Assembly-level GEMM micro-kernel

In the above section, we have described the asynchronous GEMM algorithm between DDR and scratchpad memory. Once blocks A_i, B_i and C_i are copied into scratchpad and zero-padded if necessary, the local computation can proceed.

For the sake of simplicity, in this work, we consider the row-major `GEMM_TN` configuration, in which the block C_i are decomposed into tiles C_r of size $m_r \times n_r$ to fit into the register file. This technique is known as *register-blocking*, where m_r and n_r are usually small (4, 8 and less than 16).

In row-major `GEMM_TN`, C_r is computed by a panel A_p of size $m_r \times k_c$ and a panel B_p of size $n_r \times k_c$ ($k_c = K_{wg}$, $A_p \in A_i$, $B_p \in B_i$). C_r is updated as a sequence of k_c rank-1 updates between m_r elements of A_p and n_r elements of B_p (see Fig. 8.2). This computation can be done by a BLIS *gemm* micro-kernel (see Chapter 3, Section 3.2 (page 23)), and setting the panel column-stride (cs) of A_p to M_{wg} and the panel row-stride (rs) of B_p to N_{wg}, instead of respectively m_r and n_r as defined in BLIS. [3] The MPPA ISA provides a *streaming-load* instruction which performs prefetching from the scratchpad memory into registers, bypassing the cache L1. Using this prefetch instruction allows another finer-granularity overlapping level: in-core computation and register-scratchpad communication, versus the DDR-scratchpad overlapping described in the previous section. Combining these two overlapping schemes enables parallel and highly efficient `GEMM` computation on a DMA-based, cache-free architecture like MPPA. Implementation of the corresponding micro-kernel is summarized in Algorithm 6.

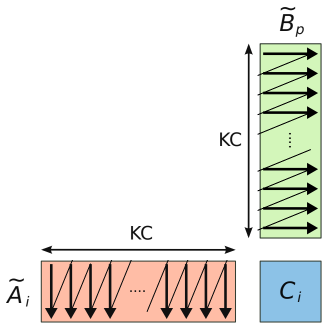


FIGURE 8.2: BLIS micro-kernel.

8.3 GEMM in OpenCL Data-Parallel

In OpenCL, blocks C_i are distributed onto work-groups. Each work-group computes one block C_i. Since `GEMM` is inherently parallel and a work-group is mapped on one PE on MPPA2, we set the local work-group size to 1×1 and global work-items count to 256×1, so that there are exactly 256 work-groups scheduled on 256 PEs during execution. The asynchronous `GEMM` pipeline on each work-group will operate the longest path, necessary to amortize the sequential cost of prolog and epilog.

[3]In BLIS, a *packing* step is carried out on A_p and B_p to pack these panels into other temporary contiguous ones ($\widetilde{A}$ and $\widetilde{B}$). These packed panels reduce the memory spatial distance between each rank-1 update (as small as m_r and n_r), thus improve cache and TLB performance.

Algorithm 6 BLIS *gemm* micro-kernel on MPPA2 with in-core computation and register prefetching.

```
/* Prolog */
Prefetch first column $A_p$ [0*cs] $\rightarrow$ registers $A_r[m_r]$
Prefetch first row $B_p$ [0*rs] $\rightarrow$ registers $B_r[n_r]$
Set registers acc$[m_r][n_r]$ to zero

/* $k_c$-loop */
for $k$ in 0 ... $k_c - 2$ do
    Prefetch ($k$ +1)-th column $A_p$ [($k$ +1)*cs] $\rightarrow$ registers $A'_r[m_r]$
    Prefetch ($k$ +1)-th row $B_p$ [($k$ +1)*rs] $\rightarrow$ registers $B'_r[n_r]$
    acc$[m_r][n_r]$ += $A_r[m_r] \times B_r[n_r]$                // $m_r \times n_r$ FMAs
    $A_r \leftarrow A'_r$
    $B_r \leftarrow B'_r$
end for

/* Epilog */
acc$[m_r][n_r]$ += $A_r[m_r] \times B_r[n_r]$                    // $m_r \times n_r$ FMAs

/* alpha-beta post-processing */
if (beta != zero) then
    Load $C_i$ $[m_r][n_r]$ $\rightarrow$ registers $C_r$ $[m_r][n_r]$
    $C_r$ $[m_r][n_r]$ *= beta
else
    Set registers $C_r$ $[m_r][n_r]$ to zero
end if
acc$[m_r][n_r]$ = alpha $\times$ acc$[m_r][n_r]$ + $C_r$ $[m_r][n_r]$    // $m_r \times n_r$ FMAs
Store acc$[m_r][n_r]$ to $C_i$ $[m_r][n_r]$ scratchpad
```

In the Kalray OpenCL data-parallel mode, an independent *compute-unit* (work-group) possesses a maximal scratchpad of 64 KB (`__local` memory). We note that according to the OpenCL definition, scheduling of work-groups and their execution binding is controlled by the OpenCL runtime. It is not possible to synchronize work-groups or explicitly bind them on the same cluster to somehow build up and share a larger common `__local` memory. Compared to the POSIX programming model, in which a compute-unit is mapped on a cluster with 1.5 MB of scratchpad memory (see Chapter 4, Section 4.2.1 (page 29)), the OpenCL data-parallel mode has much less local memory available per compute-unit for DMA transfers. The same overlapping algorithm was ported from POSIX-C to OpenCL-C without any difficulty. However, blocksizes are reduced to M_{wg} = N_{wg} = 64, K_{wg} = 48 for FP32, and $M_{wg} = N_{wg} = K_{wg}$ for FP64, due to the `__local` memory constraint. Since the performance of the tiled algorithm relies strongly on the blocksize of the fast memory layer (see Chapter 3, Section 3.1.2.2 (page 22)), this memory constraint would suffer I/O bottleneck and deliver lower performance than the POSIX-C programming model. The BLIS micro-kernel was not usable either, since linking an assembly code or a user-compiled object to an OpenCL kernel was not supported at the time of our work.

Regarding the two- and three-dimensional asynchronous copy functions, we implement a set of extended asynchronous primitives of the ones defined in the OpenCL specification. They are: (1) a general-strided copy function which supports remote and local strides, respectively in global and local memory (the original one does not support stride on `__local` memory), (2) 2D asynchronous copy and (3) 3D asynchronous copy. These two later primitives are mapped one-to-one over the underlying 2D and 3D ones from the `mppa-async` library. Similarly, these new OpenCL primitives facilitate the writing of the GEMM-async algorithm in OpenCL.

8.4 GEMM in OpenCL POSIX-like (Task-Parallel)

In the above section, we have identified the two following performance limitations of GEMM-async in OpenCL data-parallel, compared to the POSIX-C model:

- Small scratchpad memory per compute-unit. This execution and memory mapping reduces DMA blocksizes, results to I/O bound and performance loss.
- Lack of support for assembly code or pre-compiled objects. A carefully hand-tuned BLIS micro-kernel could deliver higher efficiency than the built-in OpenCL compiler, especially on the MPPA VLIW architecture.

We propose in this section a proof-of-concept of an extension to the current OpenCL environment, so-called OpenCL POSIX-like or Task-Parallel. This extension combines (1) the execution and large scratchpad memory mapping of the POSIX model, with (2) the deployment facility and expressibility of the OpenCL API, in order to produce a high performance and portable `GEMM` code on DMA-based many-core architectures.

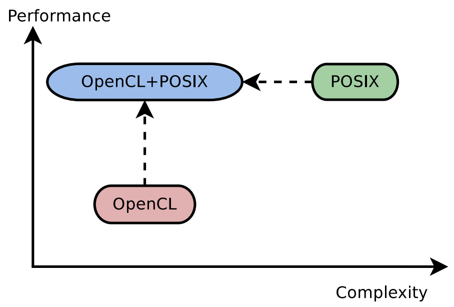


FIGURE 8.3: OpenCL POSIX-like: motivation.

8.4.1 Execution and memory mapping

In the POSIX-like mode, an OpenCL work-group is mapped on one compute cluster (CC) and operates 1024 KB of `__local` memory. When a work-group is scheduled, its kernel is booted on the PE0 of the associated CC. The kernel, running on the PE0, then relies on the Pthreads API to spawn the remaining 15 PEs on the CC and share the common scratchpad for co-working (see Fig. 8.4). This large shared memory reduces data redundancy (replication of A_i, B_i and C_i), thus notably increases the tiling blocksize.

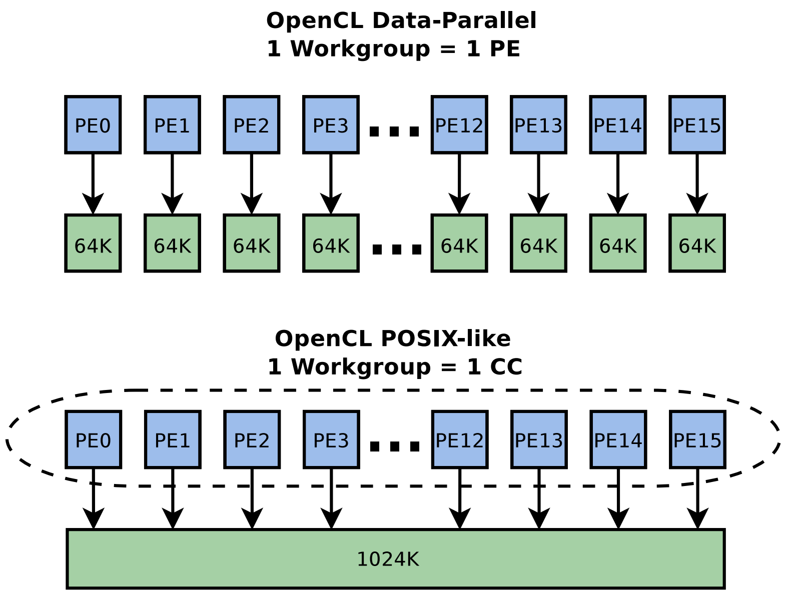
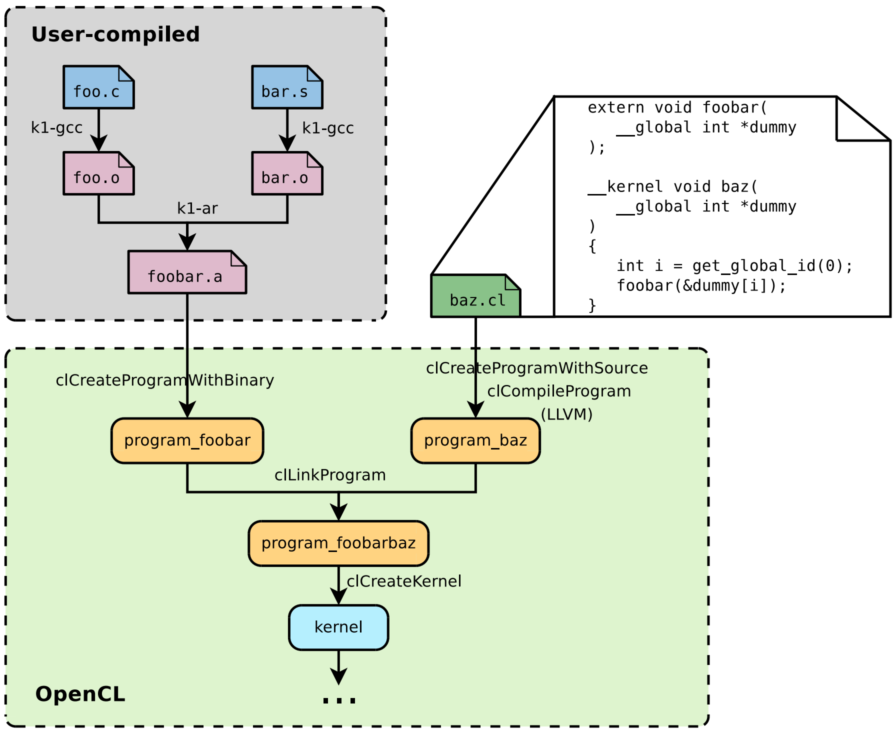


FIGURE 8.4: OpenCL POSIX-like: execution and memory (also known as OpenCL Task-Parallel).

From the ease-of-coding point of view, the OpenCL POSIX-like mode can be considered as an alternative deployment runtime to the POSIX programming model. The OpenCL kernel, once booted on the PE0, is identical to a `main` function in the POSIX-C model. Developers no longer need to take care of writing compilation *Makefile*, nor launching scripts from the host processor onto the MPPA processor. Instead, they can directly provide a kernel wrapper to the OpenCL program and get it compiled and linked to an optimized object or library implementing the crucial part of performance (BLIS micro-kernel). The executable is then sent, scheduled and started on the MPPA (operations that are considered as non-productive work) by the OpenCL driver. The on-site runtime only need to provide essential features of the `libc`, such as `malloc/free`, `printf` and `pthread_*`, as well as access to some vendor low-level tools like the `mppa-async` communication library or profiling APIs, for the kernel to be fully operational.

8.4.2 Integration of object code or library

For on-the-flight linking of the kernel wrapper and the pre-compiled object or library, the OpenCL API defines the `clLinkProgram()` function that performs the same action. The linking process within a regular OpenCL program is depicted in Fig. 8.5. We combine the handle of an object code given by the `clCreateProgramWithBinary()` with one from the sequence of `clCreateProgramWithSource()` and `clCompileProgram()`. This produces two OpenCL `cl_program`'s which are not self-standing. By performing a linking step with `clLinkProgram()`, the two binaries merge in a common and self-sufficient program, ready to be used to create an OpenCL `cl_kernel` for the device.



```
extern void foobar(
    __global int *dummy
);

__kernel void baz(
    __global int *dummy
)
{
    int i = get_global_id(0);
    foobar(&dummy[i]);
}
```

FIGURE 8.5: `clLinkProgram()`: linking a user-compiled object to an OpenCL kernel.

This procedure is similar to the traditional compiling and linking process of a regular C program. In fact, it follows the same principles under the hood (compile, link, run), but translated into function calls of a portable programming API. With this OpenCL POSIX-like extension, the C code of GEMM-async and the BLIS micro-kernel were almost entirely reused to compile a custom binary object. The OpenCL kernel wrapper is implemented as a simple call to the custom C function (with the BLIS micro-kernel inside). The host program is rewritten to use the `clLinkProgram()` function. The adaptation cost of GEMM-async from the OpenCL data-parallel to this new mode was not more than 100 lines of code.

8.5 Results

The three GEMM-async implementations are run on the MPPA2-256 processor operating at clock frequency of 400 MHz and DDR3 of 1066 MHz. At that frequency, the peak `GEMM` performance is 400 GFLOPS in FP32 and 200 GFLOPS in FP64. In this work, we present only performance of the FP32 precision to be in pace with an existing Kalray implementation of `SGEMM`, based on the name of another communication library

`MPPAIPC`, considered as the baseline of our comparisons. The POSIX-C code is run in two configurations: (1) single-DDR with the three matrices A and B and C on one DDR, and (2) double-DRR with the matrices A and C on the DDR North and the matrix B on the DDR South. This configuration provides twice more DDR bandwidth to the A-B streaming algorithm. The OpenCL runtime presently enables the use of only one DDR for global memory, either in data-parallel or task-parallel mode. In the OpenCL task-parallel mode, we link the kernel to two BLIS micro-kernels, one using normal load through the L1 cache and one using *streaming-load* to prefetch data into registers. The both BLIS micro-kernels are written in assembly and compiled by `gcc`.

TABLE 8.1: GEMM-async performance on MPPA2-256, FP32, frequency 400 MHz.

	MPPAIPC (baseline)	**POSIX**	**POSIX**	**OpenCL Data-Parallel**	**OpenCL Task-Parallel**
Matrix size	4096 × 4096	4096 × 4096	4096 × 4096	4096 × 4096	4096 × 4096
Compiler	gcc (-O3)	gcc (-O3)	gcc (-O3)	clang (-O3)	clang + gcc
#DDR	2	1	**2**	1	**1**
Local memory	1.5 MiB	1.5 MiB	**1.5 MiB**	64 KiB	**1024 KiB**
C standard	N/A	75 GFLOPS	76 GFLOPS	65 GFLOPS	N/A
Assembly: ffmawp, cached-load	200 GFLOPS	227 GFLOPS	228 GFLOPS	N/A	192 GFLOPS
Assembly: ffmawp, streaming-load	N/A	290 GFLOPS	**350 GFLOPS**	N/A	**207 GFLOPS**

Tab. 8.1 summarizes `GEMM` performance in various programming models and configurations on the MPPA2 many-core processor. We highlight the R_{max} performance of 350 GFLOPS, (corresponding to 87% of R_{peak}) of the double-DDR POSIX-C implementation, combined with the BLIS micro-kernel that asynchronously reads data to registers by *streaming-load* and performs the 64-bit vectorized `ffmawp` instruction, the most intensive in the MPPA2 VLIW ISA. It yields 1.75x speedup over the baseline score of MPPAIPC, and 1.2x over the single-DDR POSIX-C configuration.

The OpenCL data-parallel mode delivers 65 GFLOPS, due to the blocksize constraint as discussed in section 8.3. The OpenCL POSIX-like extension, with a large common scratchpad memory and capability of linking to a pre-compiled code, yields 207 GFLOPS which is three times better than the data-parallel mode and getting closer to the POSIX-C performance. Note that the current extension does not expose the same hardware resources as the POSIX-C model (one DDR versus two DDRs, 1024 KiB versus 1.5 MiB of local memory). An identical configuration will allows our OpenCL extension to reach the same `GEMM` performance of the POSIX model.

8.6 Conclusions

In this chapter, we introduce a set of `GEMM` implementations over existing programming models, as well as new proposed extensions on the MPPA2 processor. The best performance is obtained on the POSIX-C model with large development efforts ($\approx$ 4000 lines of code). The default OpenCL data-parallel mode, despite reducing the programming cost, encounters the scratchpad memory limitation due to the execution mapping. The new OpenCL POSIX-like extension combines assets of each one of the two above models and provides an elegant and high performance programming mode for the MPPA2 processor. Within a reduced development cost of about 2500 lines of code, the OpenCL POSIX-like yields a performance three times higher than the OpenCL data-parallel mode and 59% compared to the POSIX-C one, with rooms for improvement by using two DDRs and the largest local memory space.

Nevertheless, those implementations are only for benchmark and study purpose. They are not usable in system-wide software applications or production frameworks which, for the sake of portability, commonly invoke only standard APIs and libraries. For instance, convolution algorithms in deep learning or simulation solvers in physics, biology and chemistry frequently bind their compute kernels on dense and sparse linear algebra operations. Providing an optimized sparse and dense BLAS library has always been a decisive condition to any processor architecture. However, achieving that task is not always manageable.

Chapter 9

Portable and Optimized BLAS Library on Many-core Processors

...I would be panicked if I were in industry. Now I'm forced into an approach that I haven't laid the groundwork for, it requires a lot more software leverage ..., and the microprocessor manufacturers don't control the software business.

– John L. Hennessy, Stanford University.

9.1 Introduction

Besides traditional CPU and GPU platforms, recent initiatives and strategies for energy-efficient HPC [92] [93] also look at low-power system-on-chip (SoC) processors as the building blocks. These SoC processors integrate a reduced instruction set CPU accelerated with a high number of in-order DSP, VLIW or vector cores operating on scratchpad memories. To strip off power-consuming sources, accelerator cores do not implement out-of-order execution, hardware prefetcher, or globally coherent caches. Applications are expected to focus on local memory to overcome the high latency of the main memory and to leverage Direct Memory Access (DMA) engines to overlap communication and computation. Integrating DMA capabilities into a BLAS library represents a design and development challenge. This combines all the complexity of the BLAS library development, known to be elaborate, expertise-requiring and time-consuming, with the asynchronous aspects of DMA transfers, the multiplicity of memory spaces and the high execution concurrency. Failure to design a portable DMA-based BLAS will result in repeating the same time-consuming process on each new architecture. Our key contributions are as follows:

1. Asynchronous implementation of the level-3 BLAS in the BLIS framework as a flexible and fully-compatible module to leverage DMA capabilities on embedded platforms with minimal and deterministic memory footprint. [1]

2. Definition of a generic and user-friendly DMA back-end interface of six functions, inspired by the classic asynchronous one-sided *put/get* RDMA operations for interconnection networks, firstly proposed in the SHMEM paradigm [94]. The purpose of the back-end interface is to unify low-level communication libraries from different platforms and to provide a coherent mapping of BLIS primitives on any DMA-based architecture. Contributed code is released under the same BSD license as BLIS.[2]

3. Validation of the principles on the Kalray MPPA2-256 many-core processor, with mapping of the back-end DMA interface over the Kalray DMA library as an evidence of portability. The implementation delivers 75% of peak performance on a single-core execution with a memory footprint of 480 KB.

4. Implementation of a reference DMA back-end with the *memcpy* function and Pthreads. This reference implementation passed out of the box the BLIS test suite [3] on CPU as another evidence of portability and correctness.

[1]We refer to memory footprint as the scratchpad memory space needed for DMA and packing buffers. This footprint depends on the block size parameters, the overlapping scheme of each matrix as well as the parallelization of different loops. These aspects will be discussed later in the chapter.

[2]https://github.com/hominhquan/blis/tree/rdma

[3]https://travis-ci.org/hominhquan/blis

The remainder of this chapter is structured as follows. Section 9.2 presents some related work about using DMA transfers to optimize BLAS on different architectures. Section 9.3 introduces a design of a portable DMA back-end for the BLIS framework, and motivates the key technical decisions which respect to embedded memory constraints. Section 9.4 shows obtained performance on the Kalray MPPA2-256 processor with detailed observation and explanation of the results. We conclude and outline future work in Section 9.5.

9.2 Related work

Exploiting DMA transfers has been the primary option when developing applications on Digital Signal Processors (DSP), Field Programmable Gate Array (FPGA) and other accelerator architectures fitted with local memories. Early works on developing DLA library were presented for FPGA [95], DSP [96] and the CELL processor [97] [98]. Lin et al. [99] optimized the `DGEMM` operation on the Sunway many-core architecture. They achieved 88.7% peak efficiency by combining both asynchronous DMA between different memories and asynchronous RLC (Register-Level Communication) between cores within the same group. Tasende [100] introduced a BLIS-instantiated Epiphany-accelerated `SGEMM` on the Parallela board, by offloading the micro-panel computation from the host processor to the Epiphany co-processor and overlapping the host-device communication. Each of these works comes with specific solutions suited to the architecture in terms of hardware topology, DMA characteristics and memory alignment, but the fundamental technique remains the same. While the main idea has always been using DMA engines to overlap communication and computation by streaming matrix sub-blocks to the local memory, the question of generalizing algorithms to a software API and a hardware abstraction layer is seldom brought up and tackled.

Previous and comparable works to this chapter were on implementing the level-3 BLAS with DMA capabilities on Texas Instruments DSP by Igual et al. [96] and Ali et al. [101]. Later on, these techniques were applied on the BLIS framework [102] [103] and released within the MCSDK HPC toolchain [104]. Likewise, Szydzik et al. [105] presented a level-3 BLIS port with DMA transfers on the Movidius Myriad, a cutting-edge low-power processor for computer vision, that appears quite similar to previous work on TI DSP.

The aforementioned works illustrate the costly but fundamental need of providing an optimized BLAS library on novel and low-power many-core architectures. They leverage efficient DMA asynchronous communication between large (external, slow) and small (on-chip, fast) memories, which matches our first objective. However, the second objective of having a generic and portable DMA support for a BLAS library (e.g. BLIS framework) is not satisfied. Despite the fact that the TI's MCSDK HPC source code is public, development was done specifically for the C66x DSP without an abstraction layer. To the best of our knowledge, source code of the work on the Myriad architecture has not been released nor made open-source, as of today. None of other works proposes an open-source solution generic enough to be reused on another architecture. Finally, BLIS has been actively evolving since the last two years with significant changes and simplification in order to reach its current excellent maintainability. This provides an ideal opportunity for the design of a generic and portable DMA support in BLIS, so that developers of a new DMA-based architecture can quickly port a highly-optimized BLIS on their platform with minimal efforts.

9.3 Portable DMA support for level-3 BLIS

9.3.1 Algorithm overview

To reduce the communication cost, DMA-based architectures feature one or several local memories close to computing cores. Each Synergistic Processing Elements (SPE) core of the IBM CELL platform possesses 256 KB of *Local Store* (LS) memory [98]. The TI KeyStone-II processor embeds 6 MB of *Multicore Shared Memory* (MSM) [101] between eight DSP cores and the host processor. The Movidius Myriad-2 processor implements 2 MB of shared *Connection MatriX* (CMX) memory between twelve SHAVE cores [105]. On the Kalray MPPA2-256 processor, there are 32 MB of *Local Memory* distributed across sixteen compute-clusters [31]. In the remainder of this chapter, we will refer to these local memories in general as *scratchpad memory* (SMEM). A scratchpad memory is defined as a user-allocatable and DMA-accessible memory space, close to the cores in the memory hierarchy and thus accessible at lower latency and higher bandwidth than global memory.

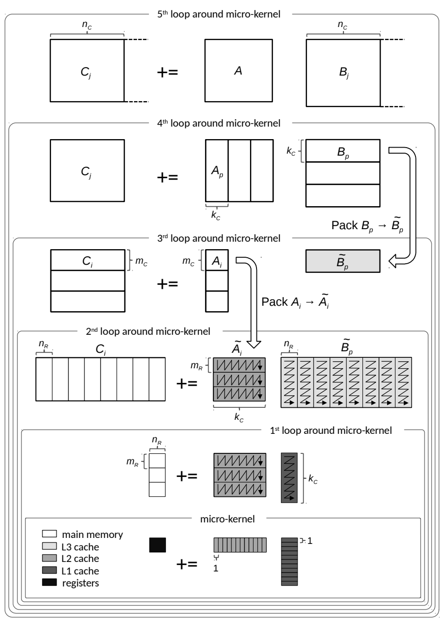
On a specific architecture, the scratchpad memory may reside in L1/L2/L3 locked cache partitions (if accessible to DMA) or in a physically distinct memory region. It should be mentioned that in the case of the TI DSP, different cache levels can be configured as multiple levels of SMEM. However, at the present work, we consider only one level of SMEM. The main reason to that is that we do not know of any other hardware in the market with memory characteristics similar to the TI DSP. Another reason is that supporting multi-level SMEMs would add considerable complexity in the BLIS-RDMA control trees [48], which we strive to keep as simple as possible in the first instance, in order to be used as a reference point for any further platform-specific customization.

Fig. 9.1b depicts the global vision of the DMA support and how it is integrated into the layered design of the cache-based version (Fig. 9.1a) of BLIS. The three matrices A, B and C are partitioned and traversed through five loops around a micro-kernel, itself is a rank-k update constituting the sixth loop. Sub-partitions of A, B and C, before being used for computation by cores, are continuously streamed by DMA from the main memory to the SMEM. Level-3 in BLIS is currently built on top of `GEMM` (and `TRSM`) following approaches proposed in [43]. Algorithm of each of these two routines is implemented as a control tree following the block-panel-based loop organization described in [106]. In order to develop a comprehensive DMA support with as few modification as possible, we re-apply the block-panel-based algorithm by instantiating two new DMA control trees for `GEMM` and `TRSM` respectively.

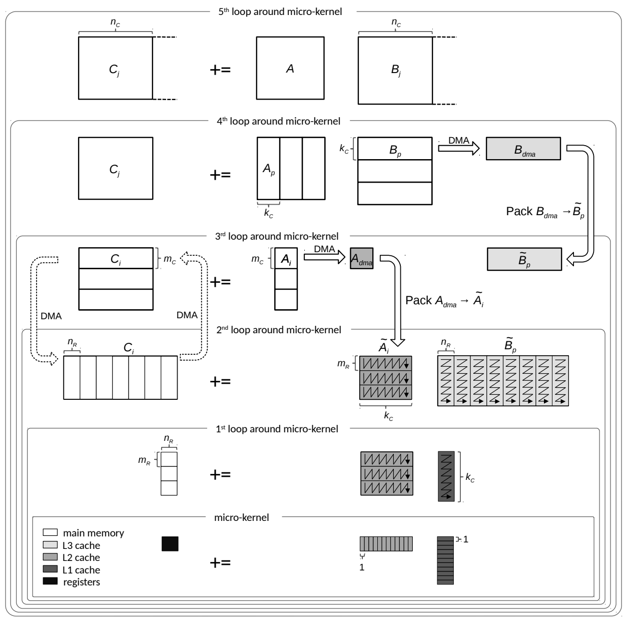
Pseudo-code to integrate DMA transfers into the control tree is summarized in Fig. 9.2, where code modifications for the DMA extension are highlighted using colored text. Sub-partitions of A, B and C are copied by DMA from the main memory into SMEM DMA buffers, then packed *in situ* in an appropriate contiguous memory layout [44] ($\widetilde{A}$ and $\widetilde{B}$). Packing facilities are used unchanged. Computation is then performed on these SMEM packed buffers $\widetilde{A}$, $\widetilde{B}$ and C_{dma}. C_{dma} is then copied back to the main memory as described in Fig. 9.2. In `GEMM`, as matrices A and B are *read-only*, double-buffering is used for copying blocks/panels of A and B into SMEM. Matrix C, which is used for both input and output, is traversed under a triple-buffering scheme: one buffer in computation, one buffer in prefetch (*get*) and one buffer in writing back (*put*).

9.3.2 Memory management

Control trees of BLIS-RDMA require SMEM dynamic allocation to perform on-demand asynchronous copies between the main memory and the SMEM. Conveniently, BLIS comes with a memory broker managing a pool of memory blocks, mainly used for the packing process. However, those blocks are often inflated to match some given algorithms or a pre-defined alignment that are needed by the underlying macro- and micro-kernel. As the DMA copy occurs before the packing and does not interface directly with either macro- or micro-kernel (see Fig. 9.1b), the DMA buffer can be allocated as a dense and contiguous block in order to reduce the SMEM footprint. We thus added the ability of using a *scratchpad allocator* into BLIS-RDMA so that the control trees can allocate exactly the needed amount of SMEM for DMA transfers. This allocator is mapped on `malloc` by default and can be re-defined by user. Recent architectures tend to embed


5th loop around micro-kernel
n_c
C_j
+=
A
B_j
4th loop around micro-kernel
C_j
A_p
k_c
B_p
Pack $B_p \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_c
A_i
$\tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
n_R
$\tilde{A}_i$
$\tilde{B}_p$
m_R
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

(A) Cache-based BLIS layers.


5th loop around micro-kernel
n_c
C_j
+=
A
B_j
4th loop around micro-kernel
C_j
A_p
k_c
B_p
DMA
B_{dma}
Pack $B_{dma} \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_c
A_i
DMA
A_{dma}
$\tilde{B}_p$
Pack $A_{dma} \rightarrow \tilde{A}_i$
DMA
DMA
2nd loop around micro-kernel
n_R
$\tilde{A}_i$
$\tilde{B}_p$
m_R
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

(B) DMA-based BLIS layers.

FIGURE 9.1: Cache-based and DMA-based layer design of BLIS. Image used and modified with permission.

```
for $j_c$ in $0, ..., n-1$ steps $n_c$
  DMA block $k_c \times n_c$ from $B[0][j_c] \rightarrow B_{dma}$
  for $p_c$ in $0, ..., k-1$ steps $k_c$
    DMA next block $k_c \times n_c$ from $B[p_c + k_c][j_c] \rightarrow B'_{dma}$
    Wait $B_{dma}$ and pack $B_{dma}$ to $\widetilde{B}$
    DMA block $m_c \times k_c$ from $A[0][p_c] \rightarrow A_{dma}$
    DMA block $m_c \times n_c$ from $C[0][j_c] \rightarrow C_{dma}$
    for $i_c$ in $0, ..., m-1$ steps $m_c$
      DMA next block $m_c \times k_c$ from $A[i_c + m_c][p_c] \rightarrow A'_{dma}$
      Wait $A_{dma}$ and pack $A_{dma}$ to $\widetilde{A}$
      DMA next block $m_c \times n_c$ from $C[i_c + m_c][j_c] \rightarrow C'_{dma}$
      Wait $C_{dma}$
      ------------------------------------------------
      for $j_r$ in $0, ..., n_c - 1$ steps $n_r$          // Macro-kernel
        for $i_r$ in $0, ..., m_c - 1$ steps $m_r$
        ----------------------------------------------
          // $m_r \times n_r$ rank-$k_c$ micro-kernel :
          gemm_ukr($\widetilde{A}$ $[i_r][0]$, $\widetilde{B}$ $[0][j_r]$, $C_{dma}$ $[i_r][j_r]$, ...)
        ----------------------------------------------
        end for
      end for
      ------------------------------------------------
      DMA block $m_c \times n_c$ $C_{dma}$ back to main memory
    end for
  end for
end for
```

FIGURE 9.2: Pseudo-code of BLIS-RDMA through the five layers. Green is used for matrix B. Blue is used for matrix A. Red is used for matrix C. Swap of $A_{dma}/B_{dma}/C_{dma}$ buffers within each iteration is not presented here.

fast on-chip memory with a dedicated allocator, such as the MCDRAM on Intel KNL and the *hbwmalloc* library [65], which match our requirement of having a dedicated SMEM allocator.

By instantiating control trees from the default ones, BLIS-RDMA inherits the same multi-threading mechanism as was enabled in the cache-based version, through five thread-count variables (e.g. `BLIS_JC_NT`, `BLIS_IC_NT` and so on) corresponding to the five loops around the micro-kernel (see Fig. 9.2). Upon available SMEM capacity, one can enable or disable multi-threading in some or a specific layer to fit the hardware constraints. For instance, setting `BLIS_JC_NT` = 2 will trigger two simultaneous DMA control flows in the outermost j_c loop and double the SMEM footprint when each DMA flow is traversing the four underlying loops. A good parallelization scheme of BLIS-RDMA on embedded platforms would be to enable multi-threading in the inner loops to expose data-sharing. Typically, threads within the i_c loop will share the same B_{dma} panels (hence same $\widetilde{B}$). More deeply, threads in the j_r loop will share both B_{dma} and A_{dma} (hence same $\widetilde{B}$ and $\widetilde{A}$ respectively) as well as the C_{dma} blocks.

Detailed memory footprint calculation of a top-down single-threaded DMA flow is given in Tab. 9.1. To take into account per-loop parallelization, let f_i and t_i denote respectively the unitary memory footprint and the thread count (`BLIS_*_NT`) of the i-th loop. The total memory footprint F of a DMA control tree is determined by the following expression:

$$F = t_5(f_5 + t_4(f_4 + t_3(f_3 + t_2(f_2 + t_1 f_1)))) + f_c \qquad (9.1)$$

We note f_c the footprint of BLIS internal data and control trees ($\leq$ 80 KiB) and take them in account in the calculation as these structures may be allocated in the SMEM for the execution performance as well. They are used only by the control code and are not duplicated like other user buffers (DMA, packing). The current DMA implementation has f_5, f_2 and f_1 equal to zero. There is further possibility to implement fine-grained multi-level SMEMs by enabling the f_2 and f_1 footprints to fit the L2 and L1 cache for instance. Regarding f_5, a simple multi-threading via the `BLIS_JC_NT` variable would be enough to enable coarse-grained parallelization.

9.3.3 Asynchronous DMA back-end interface

We present in this section a new back-end interface providing DMA support to perform asynchronous copy operations. The main idea behind this interface is inspired from the abstract design of BLIS, enabling the use of assembly-tuned micro-kernels written for a given architecture in order to reach near-peak performance. Likewise, Fig. 9.4 introduces a vendor-defined data structure and six functions of the back-end DMA interface. The

TABLE 9.1: Single-threaded level-3 BLIS-RDMA SMEM footprint calculation.

Layer	**SMEM footprint** (floating-point numbers, except f_c)	**Description**
Control code (f_c)	$\leq$ 80 KiB	Internal data & control trees
5th loop (f_5)	0	Outermost n_c partitioning
4th loop (f_4)	$2\times\ k_c \times n_c$	B_{dma} double-buffering
	$< (k_c + m_r) \times n_c$	$\widetilde{B}$ with m_r-zero-padding
3rd loop (f_3)	$2\times\ m_c \times k_c$	A_{dma} double-buffering
	$< m_c \times (k_c + n_r)$	$\widetilde{A}$ with n_r-zero-padding
	$3\times\ m_c \times n_c$	C_{dma} triple-buffering
2nd loop (f_2)	0	Macro-kernel
1st loop (f_1)	0	Macro-kernel (cont.)
Total (upper bound)	$3\times\ ((k_c \times n_c) + (m_c \times k_c) + (m_c \times n_c)) + (m_r \times n_c) + (m_c \times n_r)$ + 80 KiB	

`dma_event_t` datatype is defined as a data structure passed to the underlying platform-specific primitives to perform asynchronous transfers (`get`/`put`) and data synchronization (`wait`). The `bli_dma_backend_init()` and `bli_dma_backend_finalize()` functions, respectively called within the default functions `bli_init()` and `bli_finalize()`, are for general purpose which leaves room for setting up any DMA-related hardware resources at the beginning, as well as de-allocating them at the end of execution. Depending on the architecture, they may contain specific initializations or just be left empty.

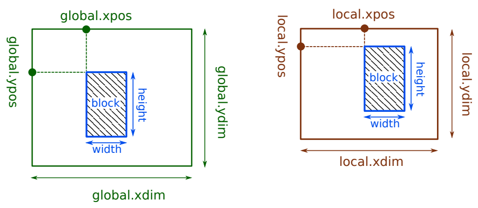


FIGURE 9.3: Illustration of a generic 2D copy. The receiving local buffer (right hand side) must be equal to (or can be larger than) the extent block (A_{dma} or B_{dma}).

The `bli_dma_backend_get2D()` and `bli_dma_backend_put2D()` are the two most important functions in the back-end interface. They respectively perform asynchronous copy of 2D blocks corresponding to matrix sub-partitions from the main memory to the SMEM and vice-versa. Their arguments provide the information necessary to describe a generic 2D copy and the pointer to an `event` parameter. Any *put/get* function is supposed to return immediately and data are transfered in the background by the hardware

```c
typedef struct point2d_s {
  int xpos, ypos, xdim, ydim;
} point2d_t;

typedef <vendor_dma_event_t> dma_event_t;

void bli_dma_backend_init(void);
void bli_dma_backend_finalize(void);
void bli_dma_backend_get2D(
      const void* global, void* local,
      point2d_t* global_point, point2d_t* local_point,
      size_t elem_size, int width, int height, dma_event_t *event);

void bli_dma_backend_put2D(
      const void* local, void* global,
      point2d_t* global_point, point2d_t* local_point,
      size_t elem_size, int width, int height, dma_event_t *event);

void bli_dma_backend_event_wait(dma_event_t *event);
int  bli_dma_backend_addr_in_global_mem(const void* addr);
```

FIGURE 9.4: BLIS-RDMA back-end interface, to be implemented by the hardware vendor in order to provide the generic 2D copy described in Fig. 9.3. The dimension of the *src* (global) and *dst* (local) 2D buffer (xdim, ydim), the dimension of the extent block (width, height, element size) and its localization (xpos, ypos) within both *src* and *dst* buffer. The position offset, from which data is read/written, is then calculated from the start address of each *src* and *dst* 2D buffer.

DMA engines. Each transfer is registered as an `event` on which computing threads (or rather the master thread) can later come back and wait for completion, by calling the `bli_dma_backend_event_wait()` function. For further details on these data structures, reader is invited to refer to previous works by Ho et al.[3] and Hascoët et al.[2].

The last function, `bli_dma_backend_addr_in_global_mem()` is not used yet and is left for future development. The purpose of this boolean function is to detect if a matrix or a sub-partition has already been allocated in the SMEM, or conversely in the main memory, so that the control tree can decide whether to trigger the DMA transfer or leave the buffer as-is. For example, one can imagine that the matrix C, for any reason (size, latency or reusability), has been allocated directly into the SMEM by the user, who later calls the `GEMM` operation between this C and two large matrices A and B from the main memory. In such case, BLIS-RDMA should work on and update the instance of C resident in SMEM and only trigger DMA transfers on A and B.

Once the `dma_event_t` datatype and the back-end interface is mapped and implemented over the platform-specific DMA library, the BLIS-RDMA control trees will then appropriately perform asynchronous copies to overlap data transfer to computation by calling the back-end functions. This shows the advantage of the BLIS-RDMA interface in terms of portability and code-reuse. Developers of an existing or a new DMA-based architecture only need to declare the scratchpad allocator, implement the six functions on top of their DMA library with usually as few as 100 lines of code (see Appendix A.2 and Appendix A.1), plug them into BLIS in the same way as the micro-kernels and finally obtain a highly-optimized BLIS-RDMA with communication-computation overlapping.

9.3.4 Special cases handling

We discuss in this section the technical solutions that were implemented in BLIS-RDMA in order to comply with the limited local memory available on DMA-based platforms. These restrictions may or may not apply on other (conventional) architectures where hardware resource is less critical. In any case, these operations can be disabled in the source code.

For `TRMM` (Triangular matrix multiply) and `TRSM` (Triangular equation system solving) operations, the long dimension of the micro-panel (k_c) is, if necessary, rounded up to be multiple of m_r or n_r (upon `sidea`) [48] so that the packing facility can manage edge cases by zero-padding.[4] Despite the fact that only a few elements are added to the DMA panel, this slightly increased k_c size can exceed the available SMEM space reserved to the library, which will likely fail at execution. In BLIS-RDMA, k_c will be rounded down to the lower nearest multiple of m_r (or n_r), instead of the upper nearest value. As m_r and n_r are often small (4, 8 and less than 32) and k_c is in the order of hundreds, this rounding-down mostly does not affect the global performance.

For `SYMM` (Symmetric matrix multiply) operations, let us denote A the symmetric matrix and consider the left-side lower-part non-transpose case with k_c larger than m_c as shown in Fig. 9.5. The packing process of a diagonal-intersected panel must perform a *symmetrization* - a mirror-copy of the lower part and symmetrically write to the upper part of the panel so that the *gemm* micro-kernel can be reused for computation. Since the SMEM address space is different from the main memory and the DMA panel of size $m_c \times k_c$ is physically detached and copied from the main memory to the SMEM, the packing routine will try to read data from an invalid region outside the DMA panel (the gray part in Fig. 9.5), which will probably cause a segmentation fault. There are three options to manage this case: (1) virtually increase the m_c dimension of the DMA panel to be equal to k_c (without changing the real m_c step parameter) to cover the gray missing

[4]Edge case occurs when a panel intersects the diagonal and thus is divided into two parts: a *gemm*-like part which is computation-ready and a *trmm*- or *trsm*-like part in which a small triangle needs to be zeroed out before the computation. The first part can be used as is as input of the *gemm* micro-kernel. Whereas on the second part, it is preferable to have k_c multiple of m_r (or n_r) so that the trailing triangular sub-matrix is compatible with the *trsm* micro-kernel. The two parts can also be computed at once by the fused *gemmtrsm* micro-kernel as well.

region or (2) reduce k_c to be equal to m_c so that the symmetrization is totally covered by the default panel, or (3) increase m_c and reduce k_c both at the same time to reach a common value. The two first options involve either adding m_c or subtracting k_c by $abs(k_c - m_c)$, that we respectively call *upper-squarization* and *lower-squarization*. The third option, which is a combination of the two first ones, is then called *mid-squarization*.

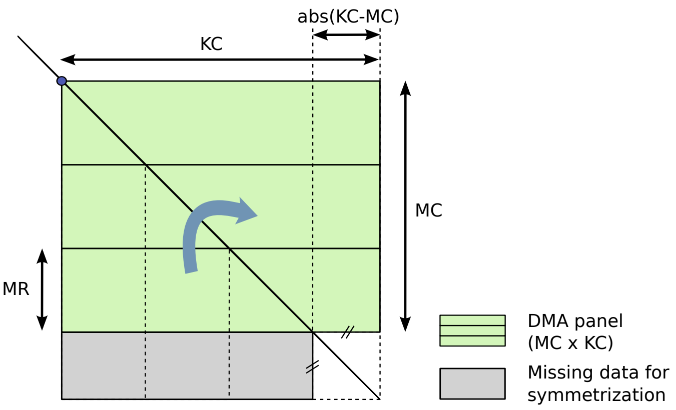


FIGURE 9.5: Illustration of DMA panel extension requirement for symmetrization in case of `SYMM` (left-side/lower/non-transpose A).

Upper-squarization, while ensuring the intended performance by keeping the same k_c block size, leads to excessive SMEM allocation that can quickly exceed the capacity of the embedded hardware. On the contrary, lower-squarization results in under-utilization of the reserved SMEM by lowering the default k_c block size, hence is exposed to performance degradation, especially on low-bandwidth memory systems. Mid-squarization, despite being more complex to implement, appears to be the most relevant option as it can yield acceptable performance while being reasonable in memory consumption, by maintaining a trade-off between computation and memory footprint. In the current state of our work, lower-squarization is employed for the sake of simplicity and proof of concept, and affects only the performance of `SYMM`. Implementing the mid-squarization is part of our roadmap.

9.4 Experimental results

9.4.1 Hardware configuration

We use the Kalray MPPA2-256 many-core processor as the target platform to validate our development. The cluster local memory of 2 MB is equivalent to the SMEM in BLIS-RDMA terminology. Multi-threading within a cluster is enabled using Pthreads. Default clock frequency of cores is set to 500 MHz with the peak performance of 2 GFLOPS/-core in `SGEMM` and 1 GFLOPS/core in `DGEMM`. The DDR3 main memory is configured at

1333 MHz with a theoretical bandwidth of 10.4 GB/s. For the time being, BLIS-RDMA is run on a single compute cluster and multi-threaded over eight cores. The eight remaining cores are not used, due to local memory capacity constraints, and will be enabled for BLIS when a planned more memory-efficient software code cache is released.

The back-end DMA interface is implemented on top of the MPPA2 asynchronous library in less than 70 lines of code (see Appendix A.2). In order to maximize data-sharing and minimize SMEM footprint, multi-threading is enabled only in the j_r loop by setting the `BLIS_JR_NT` environment variable to the thread count. This is considered as the most modest mode in BLIS-RDMA, as all threads share the same A_{dma}, B_{dma}, $\widetilde{A}$, $\widetilde{B}$ and C_{dma} buffers. Blocksize parameters m_c, n_c, k_c and register block size m_r, n_r used on MPPA2 are given in Tab. 9.2, on which we also calculate the SMEM footprint for compilation. This estimated SMEM footprint is calculated from formulas given in Tab. 9.1 and Eq. 9.1. This size bounds closely the real execution footprint, without the need of beforehand iterative run. Micro-kernels of *gemm* in single and double precision are written in the MPPA2 VLIW assembly language to maximize flops count, which we estimate to reach 97% efficiency. The fused *gemmtrsm* micro-kernels are combined from *gemm* micro-kernels and reference C99 *trsm* ones.

In the current state of work, the level-3 BLAS is run only in single and double precision on the MPPA2 processor, hence without c and z operations and Hermitian routines. With on-going software developments, we expect to be able to run the full BLIS testsuite on the MPPA2 processor in a near future. In the meantime, BLIS-RDMA has already passed the testsuite on CPU with the back-end DMA interface emulated by `memcpy` and Pthreads, which we believe to be a pertinent cross-validation. Comparing to the cache-based implementation on MPPA2, the DMA-based approach delivers between 5$\times$ to 10$\times$ speedup on small matrices (≤ 512). On larger matrices, the cache-based solution suffers severe performance degradation (up to 1000$\times$ slowdown), due to the increasing overhead of matrix-packing, caused by large strided-accesses and cache-thrashing on the software cache.

TABLE 9.2: Level-3 BLIS-RDMA configuration on MPPA2-256 for a well balance between performance and scratchpad.

	m_c	n_c	k_c	m_r	n_r	SMEM footprint
SGEMM	64	128	130	4	8	480 KiB
DGEMM	48	64	130	4	4	480 KiB
Level-3 BLAS						480 KiB

Computation results from the MPPA2 processor are checked for correctness against the OpenBLAS library on an x86 CPU. Every figure in this section is drawn with the top border line representing the theoretical peak performance of the target routine (e.g

SGEMM or DGEMM, multi-threaded or not). Any figure in which both SGEMM and DGEMM is presented will have the top border equal to the SGEMM peak and another lower line precising the DGEMM peak.

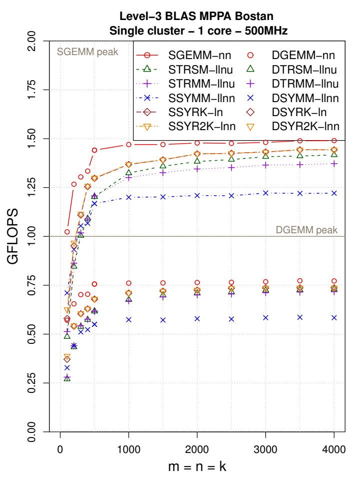
9.4.2 Multi-core level-3 BLAS

Fig. 9.6 reports the performance of level-3 BLAS on a single MPPA2 compute cluster with different core counts. As can be seen from Fig. 9.6a, most single-core operations (except SYMM) obtain between 68% and 75% of peak in both single-precision (FP32) and double-precision (FP64). The left-side lower-part non-transpose case for SYMM was intentionally chosen in our benchmark to measure the performance loss of the *lower-squarization* handling as previously discussed in section 9.3.4. Typically, single-core SSYMM and DSYMM achieve respectively 61% and 58% of peak. This represents approximately 13 to 16% of performance degradation between SYMM compared to other level-3 routines. However, we note that this degradation also depends on the block size configuration. It is significant on the MPPA2 processor because the default k_c is twice larger than m_c. Lower-squarization will thus divide the k_c block size by two, hence double the required data bandwidth as well as the DMA control-flow overhead, which explains the performance loss on the MPPA2 processor. This leads to think that larger (and square) block sizes will be less exposed (or even not at all) to this issue when they are large enough to cross the memory bar of the system and pass to the compute-bound side. Nevertheless, it is not seemingly the case for embedded platforms where local (on-chip) memory is often the most constrained resource.

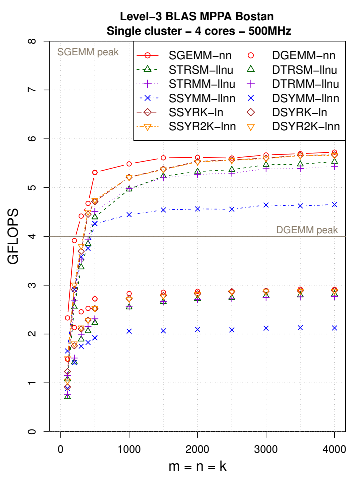
Multi-threaded performance using 4 cores and 8 cores is presented in Fig. 9.6b and Fig. 9.6c. We observe that the efficiency decreases with increasing core count. We identify the following factors to the multi-threaded performance:

- Small block size due to limited local memory capacity. This constraint not only reduces the strong scalability of the parallelized j_r loop, but also increases the I/O cost.
- Local memory bank conflicts. Micro-kernels written in MPPA2 assembly language employ a cache-bypass *streaming load* instruction between SMEM and core registers. This instruction delivers higher single-core throughput, but is sensitive to the local memory bank conflicts (local memory is composed of 16 banks interleaved every 64 bytes on the MPPA2 processor).

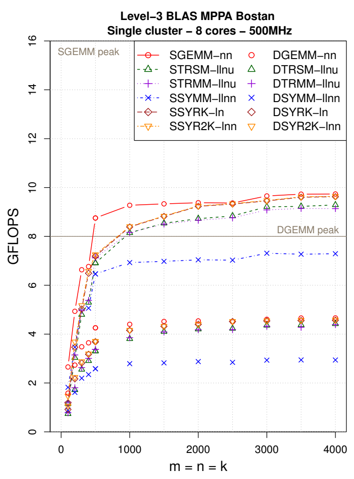
- Barrier and DMA synchronization. Thread barriers incur a significant overhead in multi-core context, especially in BLIS-RDMA where threads must synchronize between each other and wait for termination of DMA transfers.



(A) 1 core.



(B) 4 cores.



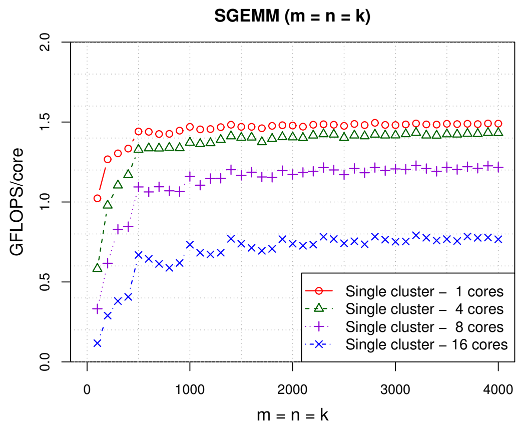
(C) 8 cores.

FIGURE 9.6: Performance of multi-threaded level-3 BLAS on MPPA2-256, single compute-cluster, 500MHz.

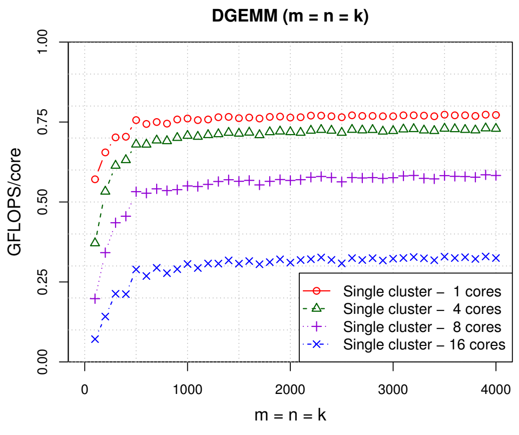
9.4.3 Multi-core xGEMM in different shapes

We present in this section performance of `GEMM` in single and double precision in three matrix configurations: (1) square matrices ($m = n = k$), (2) various $m = n$ and $k = k_c$ and (3) various k with constant $m = n$. Each configuration is run with 1, 4 and 8 cores and performance is normalized in terms of GFLOPS/core. Satisfactory strong scalability is achieved when the GFLOPS/core ratio does not depend from the number of cores used. In general, the multi-threaded performance on MPPA tends to yield lower GFLOPS/core ratio, for the reasons mentioned in the previous section. The only possible improvement in our opinion is to increase the SMEM footprint to enlarge the DMA panel block size, which is constrained by hardware limitations.

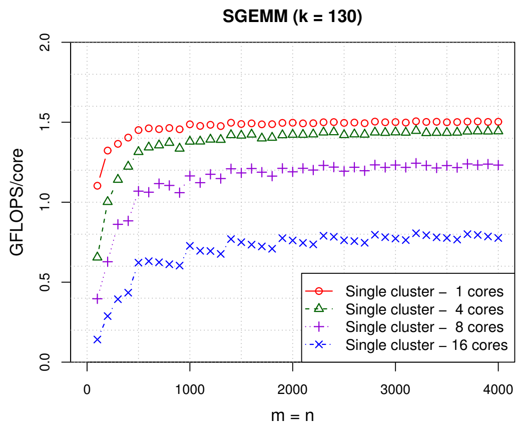
Performance efficiency and scalability of configuration (1) and (2) are reported in Fig. 9.7a-9.7b and Fig. 9.7c-9.7d, in FP32 and FP64, respectively. We observe slightly higher and more stable performance of the $k = k_c$ configuration compared to the square ($m = n = k$) configuration. When k is not multiple of k_c, the p_c loop (4th loop around the micro-kernel) must perform an extra iteration to process the trailing elements in the k-direction. This implies an additional cost in re-triggering the DMA transfers and overlapping mechanism in the inner loops to update a smaller number of elements. Fortunately, this overhead can be compensated by the computational load in the case of large k (Fig. 9.7a-9.7b).



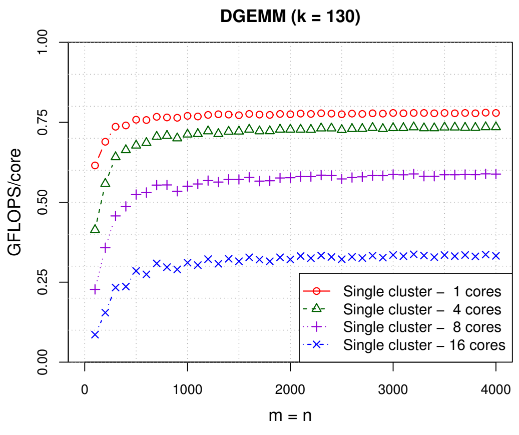
(A) SGEMM ($m = n = k$).



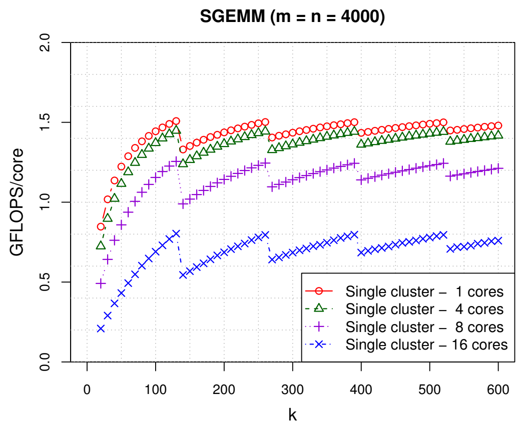
(B) DGEMM ($m = n = k$).



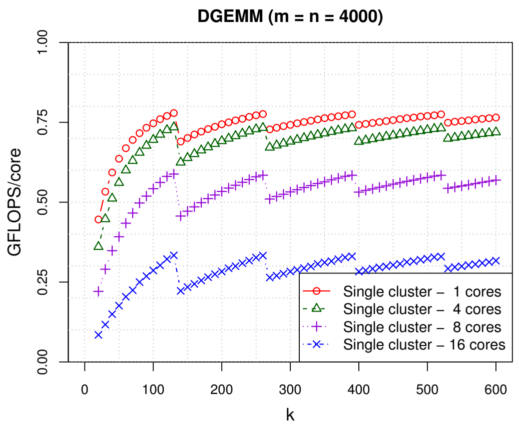
(C) SGEMM ($k = k_c = 130$).



(D) DGEMM ($k = k_c = 130$).



(E) SGEMM ($m = n = 4000$).



(F) DGEMM ($m = n = 4000$).

FIGURE 9.7: Scalability of multi-threaded xGEMM in different shapes on MPPA2-256, single compute-cluster, 500MHz.

In the case of small k however, as depicted in Fig. 9.7e-9.7f, when k is just slightly larger than k_c, the performance drops by 8% on the first k_c slump (130) and by 3% on the subsequent slumps ($N \times k_c$) This phenomenon, called *divot*, has been observed and cured by Smith et al. [107] in the cache-based version of BLIS on the Xeon Phi processor, by increasing the effective k_c by a certain size at each k_c-multiple point to create a "bridge" passing over the divot value. Interestingly enough, contrarily to the previous situation, this *performance-throttling* (or divot) on the MPPA2 processor cannot be quickly recovered to its previous sustained performance, but rather climb slowly up and re-reach the maximum performance exactly at the next k_c-multiple point, and get throttled down again (like a TCP congestion line). We believe this is a representative case of the difference of overhead between a software DMA-based communication versus a hardware cache-based implementation, where a cache-miss takes often less than 50 cycles, while a "DMA-miss" is in the order of thousands of cycles.

9.5 Conclusions

Pursuing the previous chapter work on optimizing blocked GEMM on DMA-based many-core processor, in this chapter, we present BLIS-RDMA, a generic support for, not limited to, the level-3 BLAS in the BLIS framework. We achieve three objectives: (1) standard programmability by supporting the BLAS API, with (2) portability through a back-end interface, and (3) computation performance in leveraging the cumbersome DMA capabilities on an increasing range of embedded and non-conventional many-core architectures. The work are demonstrated on the MPPA2 many-core processor with satisfying performance (75% peak on single-core). The implementation, despite being presently ported only on MPPA and CPU, is designed having in mind to support any DMA-based architecture and to minimize the porting effort through a back-end interface of six functions, aimed to be easily implemented by hardware vendors. We observe that, even with a DMA engine that enables efficient overlap between communication and computation, it is still hard to obtain near-peak performance on embedded platforms that are constrained by the local memory capacity.

Conclusions

Home isn't where you're from, it's where you find light when all grows dark.
– Pierce Brown, Golden Son.

A few steps back

In this thesis, we walk through memory-bound and compute-bound applications on increasingly popular conventional and non-conventional many-core architectures. Main contributions of this thesis are summarized as follows.

We first propose approaches for improving the data bottleneck of the 3D lattice Boltzmann method on many-cores processors by using scratchpad memory and asynchronous DMA transfer. We achieve 33 % performance gain on the MPPA architecture by actively streaming data from and to local memory, unlike in the passive OpenCL programming model.

We then tackle the memory consumption of the LBM by proposing two novel algorithms which require only one lattice array. The proposed algorithms are implemented in OpenMP and OpenCL and offer more development facility than other algorithms among the one-lattice class. Results show the adequacy of using our algorithms on a large set of cores with a sophisticated cache and memory system, by obtaining the same performance as the AA-pattern algorithm on the Xeon Phi Knights Landing processor, and 1.5 times higher memory-efficiency than the state-of-the-art two-lattice algorithm.

We present a DMA-based matrix multiplication (GEMM) algorithm. We identify necessary key features of DMA-based programming models to obtain high performance and portability. We implement those approaches on MPPA in POSIX-C and OpenCL with a hand-tuned optimized VLIW assembly kernel. We achieve up to 350 GFLOPS of SGEMM (87 % peak performance).

We present BLIS-RDMA, a generic DMA support for the BLIS framework. BLIS-RDMA enables high-performance BLAS computations on DMA-based many-core processors. BLIS-RDMA proposes a deterministic data footprint model for a controlled scratchpad allocation. BLIS-RDMA implements a portable asynchronous one-sided communication interface which allows porting of highly optimized BLAS library on any DMA-based architecture in less than 100 lines of code. Our implementation is validated on the MPPA processor, delivering over 75 % peak performance within a memory footprint of 480 KB.

These contributions are built around and for the DMA capabilities of non-conventional many-core processors in general, and the Kalray MPPA in particular. Providing a coherent application mapping on hardware and partitioning the dataset onto parallel compute-units with asynchronism and concurrency is the most challenging task.

Perspectives

Next step in our work plan is porting and improving proposed LBM in-place propagation algorithms with DMA engines to prefetch and copy *wall* data on the MPPA processor. Extending them to a 2.5D approach and complex geometries is also a plausible direction.

Regarding BLIS-RDMA, we want to study the adaptation cost to run it on the whole MPPA processor, as well as the portability on other DMA-based platforms or high-end many-core processors. There will be implementation of the *mid-squarization* approach for optimal `SYMM` performance, as well as integration of the DMA capabilities to the lower (and easier) level-1 and level-2 in order to provide a full-blown BLIS-RDMA. We also plan to support more elaborate control trees for multi-level SMEMs upon hardware availability. We will also look at extending the DMA back-end to perform DMA-copy-and-pack and remove the CPU-based packing step (at least for `GEMM` operation).

We believe future high performance systems will combine multiple computing architectures. Applications (and their sub-modules), upon their arithmetic intensity range, will be deployed and run on the most suitable or specific-designed architecture. Such a system will be heterogeneous and non-uniform in terms of memory technology, computing power and programming models. Abstraction layers and domain-specific frameworks are important to end-users who are not familiar to the hardware system. All the complexity is kept in the underlying level, where system developers will strive to deliver the highest performance.

The energy-efficient design of embedded many-core architectures is attractive to theoretical HPC power budget, but in reality may sacrifice ease of programming and sustainability of performance due to the reduced hardware feature set and memory constraints. Opening minds to adopt new architectures, identifying their strength and weakness, designing portable libraries and programming models are steps to be taken to keep the HPC community moving forward. With this final message we conclude this thesis.

Appendix A

BLIS RDMA backend: reference and MPPA implementation

Listing A.1: BLIS-RDMA backend: memcpy-based reference implementation.

```c
#include "blis.h"

void bli_dma_backend_init_ref()
{
  // Empty
}

void bli_dma_backend_finalize_ref()
{
  // Empty
}

void bli_dma_backend_trigger_get2D_ref(
  void* global,              // begin address of global buffer
  void* local,               // begin address of local buffer
  size_t size,               // size of an element in byte
  int width,                 // block width in element
  int height,                // block height in element
  point2d_t *global_point,   // global_point
  point2d_t *local_point,    // local_point
  dma_event_t *event         // DMA event. If NULL: blocking,
                             // else return immediate and must later call
                             // wait() on this event
)
{
  int i;
  char* local_ptr  = ((char*) local)  +
    (((local_point->ypos * local_point->xdim) + local_point->xpos) * size
   );
  char* global_ptr = ((char*) global) +
    (((global_point->ypos * global_point->xdim) + global_point->xpos) *
   size);

  for(i = 0; i < height; i++)
  {
    memcpy(local_ptr, global_ptr, width*size);
    local_ptr  += (local_point->xdim * size);
    global_ptr += (global_point->xdim * size);
  }
}
```

```

void bli_dma_backend_trigger_put2D_ref(
  void* local,              // begin address of local buffer
  void* global,             // begin address of global buffer
  size_t size,              // size of an element in byte
  int width,                // block width in element
  int height,               // block height in element
  point2d_t *local_point,   // local_point
  point2d_t *global_point,  // global_point
  dma_event_t *event        // DMA event. If NULL: blocking,
                            // else return immediate and must later call
                            // wait() on this event
)
{
  int i;
  char* local_ptr  = ((char*) local)  +
    (((local_point->ypos * local_point->xdim) + local_point->xpos) * size
   );
  char* global_ptr = ((char*) global) +
    (((global_point->ypos * global_point->xdim) + global_point->xpos) *
   size);

  for(i = 0; i < height; i++)
  {
    memcpy(global_ptr, local_ptr, width*size);
    local_ptr  += (local_point->xdim * size);
    global_ptr += (global_point->xdim * size);
  }
}

void bli_dma_backend_event_wait_ref(dma_event_t *event)
{
  // Nothing to wait here
}

int bli_dma_backend_addr_in_global_mem_ref(void* addr)
{
  return 1;
}
```

Listing A.2: BLIS-RDMA backend: MPPA implementation.

```c
#include "blis.h"

void bli_dma_backend_init()
{
  // Empty
}

void bli_dma_backend_finalize()
{
  // Empty
}

void bli_dma_backend_trigger_get2D(
  void* global,               // begin address of global buffer
  void* local,                // begin address of local buffer
  size_t size,                // size of an element in byte
  int width,                  // block width in element
  int height,                 // block height in element
  point2d_t *global_point,    // global_point
  point2d_t *local_point,     // local_point
  dma_event_t *event          // DMA event. If NULL: blocking,
                              // else return immediate and must later call
                              // wait() on this event
)
{
  mppa_async_sget_block2d(
    (void*) local,                                  // local
    MPPA_ASYNC_DDR_0,                               // NOTE : DDR0 hardcoded
    (uintptr_t)global-(uintptr_t)&DDR_START,        // offset
    (size_t) size,                                  // size
    (int) width,                                    // width
    (int) height,                                   // height
    (const mppa_async_point2d_t*) local_point,      // local_point
    (const mppa_async_point2d_t*) global_point,     // remote_point
    (mppa_async_event_t*) event                     // event
  );
  mppa_async_event_setdinval((mppa_async_event_t*) event, 0);
}

void bli_dma_backend_trigger_put2D(
  void* local,                // begin address of local buffer
  void* global,               // begin address of global buffer
  size_t size,                // size of an element in byte
  int width,                  // block width in element
  int height,                 // block height in element
  point2d_t *local_point,     // local_point
  point2d_t *global_point,    // global_point
  dma_event_t *event          // DMA event. If NULL: blocking,
                              // else return immediate and must later call
                              // wait() on this event
)
{
  mppa_async_sput_block2d(
    (const void*) local,                            // local
    MPPA_ASYNC_DDR_0,                               // NOTE : DDR0 hardcoded
    (uintptr_t)global-(uintptr_t)&DDR_START,        // offset
    (size_t) size,                                  // size
    (int) width,                                    // width
    (int) height,                                   // height
    (const mppa_async_point2d_t*) local_point,      // local_point
    (const mppa_async_point2d_t*) global_point,     // remote_point
    (mppa_async_event_t*) event                     // event
  );
  mppa_async_event_setdinval((mppa_async_event_t*) event, 0);
}

void bli_dma_backend_event_wait(dma_event_t *event)
{
  mppa_async_event_wait((mppa_async_event_t*)event);
}

int bli_dma_backend_addr_in_global_mem(void* addr)
{
  return (((uintptr_t)addr >= (uintptr_t)&DDR_START) ? 1 : 0);
}
```

Bibliography

[1] Minh Quan Ho, Bernard Tourancheau, Christian Obrecht, Benoît Dupont de Dinechin, and Jérôme Reybert. MPI communication on MPPA many-core NoC: design, modeling and performance issues. In Gerhard R. Joubert, Hugh Leather, Mark Parsons, Frans J. Peters, and Mark Sawyer, editors, *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK*, volume 27 of *Advances in Parallel Computing*, pages 113–122. IOS Press, 2015.

[2] Julien Hascoët, Benoît Dupont de Dinechin, Pierre Guironnet de Massas, and Minh Quan Ho. Asynchronous one-sided communications and synchronizations for a clustered manycore processor. In *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia, ESTImedia 2017, Seoul, Republic of Korea, October 15 - 20, 2017*, pages 51–60, 2017.

[3] Minh Quan Ho, Christian Obrecht, Bernard Tourancheau, Benoit Dupont de Dinechin, and Julien Hascoet. Improving 3D lattice Boltzmann method stencil with asynchronous transfers on many-core processors. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC) (IPCCC 2017)*, San Diego, USA, December 2017.

[4] Minh Quan Ho, Christian Obrecht, and Bernard Tourancheau. New parallel in-place update algorithm for better memory usage in 3D lattice Boltzmann algorithm. In submission, 2017.

[5] Minh Quan Ho, Benoit Dupont de Dinechin, Bernard Tourancheau, and Christian Obrecht. BLIS-RDMA: A portable and high performance level-3 BLAS for DMA-based many-core architectures. In submission, 2017.

[6] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[7] OpenACC: More Science Less Programming, 2018.

[8] Khronos OpenCL Working Group. Editor : Aaftab Munshi. The OpenCL Specification. Version 1.2, 2012.

[9] William D Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT Press, 1999.

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[11] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos SYCL for OpenCL: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, page 24. ACM, 2015.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[13] Zizhong Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Notices*, volume 48, pages 167–176. ACM, 2013.

[14] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *ACM SIGPLAN Notices*, 47(8):225–234, 2012.

[15] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[16] Hans-Joachim Wunderlich, Claus Braun, and Sebastian Halder. Efficacy and efficiency of algorithm-based fault-tolerance on GPUs. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 240–243. IEEE, 2013.

[17] P Rech, C Aguiar, C Frost, and L Carro. An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on GPUs. *IEEE Transactions on Nuclear Science*, 60(4):2797–2804, 2013.

[18] Uriel Frisch, Brosl Hasslacher, and Yves Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical review letters*, 56(14):1505, 1986.

[19] Guy R McNamara and Gianluigi Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical review letters*, 61(20):2332, 1988.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Peter Bailey, Joe Myre, Stuart DC Walsh, David J Lilja, and Martin O Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 550–557. IEEE, 2009.

[29] Martin Geier and Martin Schönherr. Esoteric Twist: An Efficient in-Place Streaming Algorithmus for the Lattice Boltzmann Method on Massively Parallel Hardware. *Computation*, 5(2):19, 2017.

[30] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. Efficient GPU implementation of the linearly interpolated bounce-back boundary condition. *Computers & Mathematics with Applications*, 65(6):936–944, 2013.

[31] 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.

[32] 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.

[33] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[34] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[35] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammerling, James Demmel, C Bischof, and Danny Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

[36] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[37] John D McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, 1995.

[38] Antoine Petitet, Jack Dongarra, et al. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, September 2008.

[39] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[40] Intel. Intel Math Kernel Library, Accessed 2017.

[41] Advanced Micro Devices (AMD). AMD Core Math Library, Accessed 2017.

[42] IBM. Engineering and Scientific Subroutine Library, Accessed 2017.

[43] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4, 2008.

[44] Kazushige Goto and Robert A Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[45] Zhang Xianyi, Wang Qian, and Werner Saar. OpenBLAS, Accessed 2017.

[46] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[47] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the plasma numerical library to the openmp standard. *International Journal of Parallel Programming*, 45(3):612–633, 2017.

[48] Field G Van Zee and Robert A Van De Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):14, 2015.

[49] NVIDIA. GPU-accelerated standard BLAS library, Accessed 2017.

[50] Advanced Micro Devices (AMD). A software library containing BLAS functions written in OpenCL, Accessed 2017.

[51] Advanced Micro Devices (AMD). Next generation BLAS implementation for ROCm platform, Accessed 2017.

[52] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[53] Cedric Nugteren. CLBlast: A Tuned OpenCL BLAS Library. *CoRR*, abs/1705.05249, 2017.

[54] Ahmad Abdelfattah, David E. Keyes, and Hatem Ltaief. KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators. *CoRR*, abs/1410.1726, 2014.

[55] Field Van Zee, Ernie Chan, Robert van de Geijn, Enrique Quintana, and Gregorio Quintana-Orti. Introducing: The libflame library for dense matrix computations. *Computing in science & engineering*, 2009.

[56] Tze Meng Low, Francisco D Igual, Tyler M Smith, and Enrique S Quintana-Orti. Analytical modeling is enough for high-performance BLIS. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):12, 2016.

[57] Field G Van Zee, Tyler M Smith, Bryan Marker, Tze Meng Low, Robert A Geijn, Francisco D Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, et al. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software (TOMS)*, 42(2):12, 2016.

[58] Márcio Castro, Fabrice Dupros, Emilio Francesquini, Jean-François Méhaut, 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.

[59] Sebastian Raase and Tomas Nordström. On the use of a many-core processor for computational fluid dynamics simulations. *Procedia Computer Science*, 51:1403–1412, 2015.

[60] Prateek Nagar, Fengguang Song, Luoding Zhu, and Lan 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*, pages 51–60. IEEE, 2015.

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

[62] 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.

[63] 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.

[64] 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.

[65] Avinash Sodani. Knights Landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–24. IEEE, 2015.

[66] Vali Codreanu and Jorge Rodríguez. Best Practice Guide - Knights Landing, January 2017 (accessed Juin 12, 2017).

[67] Ronald W Green. Memory movement and initialization: Optimization and control, September 2014 (accessed October 12, 2017).

[68] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[69] Tilera Corporation. Tile processor architecture overview for the Tilepro series, February 2013.

[70] Julien Mottin, Mickael Cartron, and Giulio Urlini. The STHORM Platform. In *Smart Multicore Embedded Systems*, pages 35–43. Springer, 2014.

[71] Nicole Hemsoth. The Tiny Chip That Could Disrupt Exascale Computing, March 2015. http://www.theplatform.net/2015/03/12/the-little-chip-that-could-disrupt-exascale-computing.

[72] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[73] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[74] Z Xianyi, W Qian, and Z Chothia. OpenBLAS, version 0.2. 8. *URL http://www. openblas. net/. Fe tched*, pages 09–13, 2013.

[75] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[76] Loic Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on myrinet. In *Parallel and Distributed Processing*, pages 472–485. Springer, 1998.

[77] Loïc Prylli, Bernard Tourancheau, and Roland Westrelin. Modeling of a high speed network to maximize throughput performance: the experience of BIP over Myrinet. *Parallel and Distributed Processing Techniques and Applications-PDPTA*, 2:341–349, 1998.

[78] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, 2002.

[79] James Ryan Psota. *rMPI: An MPI-compliant message passing library for tiled architectures*. PhD thesis, Massachusetts Institute of Technology, 2005.

[80] James Psota and Anant Agarwal. rMPI: message passing on multicore processors with on-chip interconnect. In *High Performance Embedded Architectures and Compilers*, pages 22–37. Springer, 2008.

[81] Mikyung Kang, Eunhui Park, Minkyoung Cho, Jinwoo Suh, D Kang, and Stephen P Crago. MPI performance analysis and optimization on tile64/maestro. In *Proceedings of Workshop on Multicore Processors for Space—Opportunities and Challenges Held in conjunction with SMC-IT*, pages 19–23, 2009.

[82] Bruno d'Ausbourg, Marc Boyer, Eric Noulard, and Claire Pagetti. Deterministic Execution on Many-Core Platforms: application to the SCC. In *4th Many-core Applications Research Community (MARC) Symposium*, page 43, 2012.

[83] Timothy G Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, et al. The 48-core SCC processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High*

Performance Computing, Networking, Storage and Analysis, pages 1–11. IEEE Computer Society, 2010.

[84] Carsten Clauss, Stefan Lankes, Pablo Reble, and Thomas Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525–532. IEEE, 2011.

[85] Sreeram Potluri, Khaled Hamidouche, Devendar Bureddy, and Dhabaleswar K DK Panda. MVAPICH2-MIC: A High Performance MPI Library for Xeon Phi Clusters with InfiniBand.

[86] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. *Active messages: a mechanism for integrated communication and computation*, volume 20. ACM, 1992.

[87] Kalray Inc. MPPA-256 Cluster and I/O Subsystem Architecture, 2015. Specification documentation.

[88] Kalray Inc. MPPAIPC Performance, 2013. Benchmark report.

[89] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Öberg, Kari Tiensyrjä, and Ahmed Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112. IEEE, 2002.

[90] Kalray Inc. Kalray Platforms and Boards. Accessed March 30, 2015.

[91] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G Sedukhin. Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 396–405. IEEE, 2012.

[92] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2013.

[93] Michael Feldman. Mont-Blanc 2020 Project Sets Sights on Exascale Processor, December 2017 (accessed December 30, 2017).

[94] Scott Pakin, Vijay Karamcheti, and Andrew A Chien. Fast Messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE concurrency*, 5(2):60–72, 1997.

[95] Yong Dou, Stamatis Vassiliadis, Georgi Krasimirov Kuzmanov, and Georgi Nedeltchev Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 86–95. ACM, 2005.

[96] Francisco D Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, and Robert A van de Geijn. Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 26. IEEE Computer Society Press, 2012.

[97] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1175–1186, 2008.

[98] Vaibhav Saxena, Prashant Agrawal, Yogish Sabharwal, Vijay K Garg, Vimitha A Kuruvilla, and John A Gunnels. Optimization of BLAS on the Cell Processor. In *HiPC*, volume 5374, pages 18–29. Springer, 2008.

[99] James Lin, Zhigeng Xu, Akira Nukada, Naoya Maruyama, and Satoshi Matsuoka. Optimizations of Two Compute-Bound Scientific Kernels on the SW26010 Many-Core Processor. In *Parallel Processing (ICPP), 2017 46th International Conference on*, pages 432–441. IEEE, 2017.

[100] Miguel Tasende. Generation of the Single Precision BLAS library for the Parallella platform, with Epiphany co-processor acceleration, using the BLIS framework. In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/-DataCom/CyberSciTech), 2016 IEEE 14th Intl C*, pages 894–897. IEEE, 2016.

[101] Murtaza Ali, Eric Stotzer, Francisco D Igual, and Robert A van de Geijn. Level-3 BLAS on the TI C6678 multi-core DSP. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 179–186. IEEE, 2012.

[102] Devangi Parikh, Francisco D Igual, and Murtaza Ali. Implementation of Linear Algebra Libraries for Embedded Architectures Using BLIS, Accessed 2017.

[103] Devangi Parikh and Will Leven. An Implementation of GEMM for DMA-enabled Architectures, Accessed 2017.

[104] Texas Instruments (TI)). MCSDK HPC 3.x Linear Algebra Library, Accessed 2017.

[105] Tomasz Szydzik, Marius Farcas, Valeriu Ohan, and David Moloney. Level-3 BLAS on Myriad multi-core media-processor SoC. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–1. IEEE, 2014.

[106] Field G Van Zee and Tyler M Smith. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Transactions on Mathematical Software. Under review*, 2017.

[107] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1049–1059. IEEE, 2014.