Universidade Federal do Paraná

Paulo Roberto de Carvalho Junior

GPU Communication Performance Engineering for the Lattice Boltzmann Method

Curitiba PR
2016

Paulo Roberto de Carvalho Junior

GPU Communication Performance Engineering for the Lattice Boltzmann Method

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Daniel Weingaertner.

Curitiba PR
2016

C331g Carvalho Junior, Paulo Roberto de
GPU Communication Performance Engineering for the Lattice Boltzmann Method / Paulo Roberto de Carvalho Junior. – Curitiba, 2016.
62 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2016.

Orientador: Daniel Weingaertner .
Bibliografia: p. 59-62.

1. HPC (Computação). 2. Graphics Processing Unit. 3. CUDA (Computer architecture). 4. Lattice Boltzmann methods. I. Universidade Federal do Paraná. II.Weingaertner, Daniel. III. Título.

CDD: 004.068

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós Graduação em INFORMÁTICA
Código CAPES: 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **PAULO ROBERTO DE CARVALHO JUNIOR**, intitulada: "**GPU Communication Performance Engineering for the Lattice Boltzmann Method**", após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO.

Curitiba, 10 de Agosto de 2016.

Prof DANIEL WEINGAERTNER
Presidente da Banca Examinadora (UFPR)

Prof MARCO ANTONIO ZANATA ALVES
Avaliador Interno (UFPR)

Prof HARALD KÖSTLER
Avaliador Externo (FAU)

This dissertation is dedicated to my wife, Luciene, for her endless support and encouragement during this challenging work. I am truly thankful for having her in my life. This work is also dedicated to my parents, Rosangela and Paulo, for their encouragement and unconditional love.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. Dr. Daniel Weingaertner, for all opportunities he has given me and for providing me a great support. I am very grateful for his scientific guidance and for the helpful discussions we had.

I would like to thank Prof. Dr. Ulrich Rüde for giving me the opportunity to study at Friedrich-Alexander-Universität Erlangen-Nürnberg for one semester as an exchange student. The period I studied there was very important to my research.

Many thanks to Martin Bauer and Christian Godenschwager for the support with waLBerla, to José Aurimar Sepka Júnior for helping me with some experiments, to the committee members, Prof. Dr. Marco Antonio Zanata Alves and Prof. Dr. Harald Köstler, for the helpful discussion and the improvement suggestions for this work, and to everyone else who gave me technical or administrative support.

Special thanks to my former advisor, Prof. Dr. Alexandre Ibrahim Direne, who could not continue to provide me guidance due to circumstances beyond his control. The period I worked with him on another project was very productive and we had very interesting discussions.

Very special thanks to my wife, Luciene, for her endless support and encouragement during this challenging work, and to my parents, Rosangela and Paulo, for their encouragement and unconditional love.

Resumo

A crescente importância do uso de GPUs para computação de propósito geral em supercomputadores faz com que o bom suporte a GPUs seja uma característica valiosa de *frameworks* de software para computação de alto desempenho como o waLBerla. waLBerla é um *framework* de software altamente paralelo que suporta uma ampla gama de fenômenos físicos. Embora apresente um bom desempenho em CPUs, testes demonstraram que as suas soluções de comunicação para GPU têm um desempenho ruim. Neste trabalho são apresentadas soluções para melhorar o desempenho, a eficiência do uso de memória e a usabilidade do waLBerla em supercomputadores baseados em GPU. A infraestrutura de comunicação proposta para GPUs NVIDIA com suporte a CUDA mostrou-se 25 vezes mais rápida do que o mecanismo de comunicação para GPU disponíveis anteriormente no waLBerla. Nossa solução para melhorar a eficiência do uso de memória da GPU permite usar 55% da memória necessária por uma abordagem simplista, o que possibilita executar simulações com domínios maiores ou usar menos GPUs para um determinado tamanho de domínio. Adicionalmente, levando-se em consideração que o desempenho de *kernels* CUDA se mostrou altamente sensível ao modo como a memória da GPU é acessada e a detalhes de implementação, foi proposto um mecanismo de indexação flexível de domínio que permite configurar as dimensões dos blocos de *threads*. Além disso, uma aplicação do *Lattice Boltzmann Method* (LBM) foi desenvolvida com *kernels* CUDA altamente otimizados a fim de se realizar todos os experimentos e testar todas as soluções propostas para o waLBerla.

Palavras-chave: HPC, GPU, CUDA, Comunicação, Memória, Lattice Boltzmann Method, waLBerla.

Abstract

The increasing importance of GPUs for general-purpose computation on supercomputers makes a good GPU support by High-Performance Computing (HPC) software frameworks such as waLBerla a valuable feature. waLBerla is a massively parallel software framework that supports a wide range of physical phenomena. Although it presents good performance on CPUs, tests have shown that its available GPU communication solutions perform poorly. In this work, we present solutions for improving waLBerla's performance, memory usage efficiency and usability on GPU-based supercomputers. The proposed communication infrastructure for CUDA-enabled NVIDIA GPUs executed 25 times faster than the GPU communication mechanism previously available on waLBerla. Our solution for improving GPU memory usage efficiency allowed for using 55% of the memory required by a naive approach, which makes possible for running simulations with larger domains or using fewer GPUs for a given domain size. In addition, as CUDA kernel performance showed to be very sensitive to the way data is accessed in GPU memory and kernel implementation details, we proposed a flexible domain indexing mechanism that allows for configuring thread block sizes. Finally, a Lattice Boltzmann Method (LBM) application was developed with highly optimized CUDA kernels in order to carry out all experiments and test all proposed solutions for waLBerla.

Keywords: HPC, GPU, CUDA, Communication, Memory, Lattice Boltzmann Method, waLBerla.

Contents

List of Figures

List of Acronyms

API	Application Programming Interface
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DINF	Departamento de Informática (Department of Computer Science)
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
ECC	Error-Correcting Code
GCC	GNU Compiler Collection
GPGPU	General-Purpose Computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High-Performance Computing
LDC	Lid-Driven Cavity
LBM	Lattice Boltzmann Method
MFLUPS	Million Fluid Lattice Updates per Second
MLUPS	Million Lattice Updates per Second
MPI	Message Passing Interface
MRT	Multiple Relaxation Time
NUMA	Non-Uniform Memory Access
NVCC	NVIDIA CUDA Compiler
P2P	Peer-to-Peer
PCIe	Peripheral Component Interconnect Express
PDF	Particle Distribution Functions
RAM	Random-Access Memory
RDMA	Remote Direct Memory Access
SRT	Single Relaxation Time
UFPR	Universidade Federal do Paraná (Federal University of Paraná)
waLBerla	Widely Applicable Lattice Boltzmann from Erlangen

List of Symbols

B	Total number of blocks that decomposes a domain
B_x	Number of domain decomposition blocks in x direction
B_y	Number of domain decomposition blocks in y direction
B_z	Number of domain decomposition blocks in z direction
B_α	Number of blocks in any axis direction in a cubic decomposition
g	Thickness of the ghost layer
i_p	Index for addressing an element in *GPUField*'s data pointer
M	Total memory allocated by *GPUField*s
M_c	Total size of a cell in bytes
M_d	Memory allocated by the destination *GPUField*
M_f	Memory allocated by the flag *GPUField*
M_{gpu}	Total GPU memory
M_s	Memory allocated by the source *GPUField*
N	Total number of domain cells
N'	Total number of domain cells, including ghost layer cells
N_x	Number of domain cells in x direction
N'_x	Number of domain cells in x direction, including ghost layer cells
n_x	Number of block cells in x direction
n'_x	Number of block cells in x direction, including ghost layer cells
N_y	Number of domain cells in y direction
N'_y	Number of domain cells in y direction, including ghost layer cells
n_y	Number of block cells in y direction
n'_y	Number of block cells in y direction, including ghost layer cells
N_z	Number of domain cells in z direction
N'_z	Number of domain cells in z direction, including ghost layer cells
n_z	Number of block cells in z direction
n'_z	Number of block cells in z direction, including ghost layer cells
N_α	Number of cells in any axis direction in a cubic domain
n_α	Number of cells in any axis direction in a cubic block
s	Size in bytes of an element in a field

Chapter 1

Introduction

High-Performance Computing (HPC) has gained increasing importance in the last years as it allowed many of the recent advances in science and engineering. The field of Computational Fluid Dynamics (CFD) is among the ones that most benefit from HPC and is of great importance for the industry [14].

The top end HPC systems are commonly referred to as supercomputers and the 500 most powerful of them that are commercially available are listed on TOP500 [44]. The fastest supercomputers in the world are based on cluster architecture, which relies on splitting the application processing among a set of computing nodes that are connected through fast local area networks, e.g. InfiniBand and variations of Gigabit Ethernet.

As the price, performance and energy efficiency of Graphics Processor Units (GPUs) have improved, their adoption in HPC for general purpose computation is increasing significantly in order to fulfill the demands for floating point operations [43]. According to TOP500, as of June 2016, 2 of the top 10 supercomputers use NVIDIA GPUs to accelerate computation: the number 3 system, Titan, and the number 8 system, Piz Daint [45].

Although there are many advantages of using GPUs for general purpose computation on supercomputers, achieving their optimal performance is challenging. Many GPU communication approaches require using CPUs to intermediate the communication steps, resulting in processing overhead, additional latency and intra-node bandwidth consumption. Besides, different communication technologies are used for intra-node and inter-node data transfer, each one with distinct bandwidth and latency. Hence, improving communication performance and balancing processing and communication are still subject of study and discussion.

The Lattice Boltzmann Method (LBM) is a computational fluid dynamics method widely accepted in academia and industry for solving incompressible flows [9]. It has gained increasingly importance as it takes the best advantage of massively parallel supercomputers.

Software frameworks such as waLBerla [6] are available to help the development of LBM applications by providing programming patterns, data structures and routines of common use in HPC. Although waLBerla supports a wide range of physical phenomena and runs on some of the top 10 supercomputers in the world, it currently does not support efficient GPU communication.

This dissertation presents a new communication infrastructure and its associated programing interface for the waLBerla framework in order to improve its performance and usability on GPU-based supercomputers. In addition, an LBM application making use of the developed solutions is also presented.

1.1 Objectives

The general objectives of this dissertation are:

- Evaluate different GPU communication strategies, taking into consideration inter-node, intra-node and intra-GPU scenarios.
- Analyze different mechanisms for improving GPU memory management and efficiency.
- Improve waLBerla's performance, memory usage efficiency and usability on GPU-based supercomputers with state-of-the-art solutions.
- Develop a distributed parallel LBM application in order to carry out experiments and test all proposed solutions.

1.2 Contributions

The main contributions of this dissertation for the HPC field are:

- A GPU-to-GPU communication infrastructure for stencil-based applications transparently embedded into an HPC framework.
- An efficient GPU memory management mechanism for 3D stencil computation easily accessible by application programmers.
- A flexible mechanism for indexing 4D data on GPU memory and support for contiguous memory.
- Significant speedups in GPU-based LBM simulations using waLBerla.
- Proposal of communication experiments for assessing the distributed GPU communication overhead and bottlenecks.

Chapter 2

Background

This chapter provides appropriate background for the solutions proposed in this dissertation. CUDA and related technologies such as GPUDirect and CUDA-aware MPI are introduced. The Lattice Boltzmann Method (LBM) as well as the Lid-Driven Cavity (LDC) problem, are briefly presented. Finally, the waLBerla framework and some of relevant details of its architecture are described.

2.1 CUDA

The general-purpose computing on GPU (GPGPU) is currently of great importance for HPC as price, performance and energy efficiency of GPU are improving. NVIDIA GPUs are among the most popular coprocessors on top end HPC systems [44].

Basically, an NVIDIA GPU consists of multiple streaming multiprocessors, each one consisting of multiple processor cores that share a directly connected DRAM as the global memory. The programming model provided by NVIDIA for that architecture is called CUDA (Compute Unified Device Architecture) [24].

CUDA exposes to the programmer a set of language extensions for C, C++ and Fortran. In CUDA programming model, part of the program is executed by the host, i.e. the CPU, and part is executed by the device, i.e. GPU. The part of the program executed by the host is usually a sequential program. On the other hand, programs executed by the device are highly parallel. Typically, the host request execution of programs in device by launching kernels. When launching a kernel, the host defines how many threads will execute it. Each thread is executed by a different processor core handling different data. As depicted in Figure 2.1, CUDA threads are grouped in thread blocks and blocks are grouped in grids. Every thread has an index to address it inside a block. Similarly, every block has an index to address it inside a grid. Both indexes are 3-component vectors that can be used for indexing threads and blocks in 1D, 2D or 3D fashion.

The CUDA programming model usually assumes that both the host and the device maintain their own separate memory spaces. However, the CUDA Runtime API [25] provides functions such as *cudaMemCpy()* and *cudaMemCpy3D()* for moving data between device and host.

With CUDA Toolkit v6 [26], however, the Unified Memory feature was introduced providing a memory model that simplifies memory management for the application programmer by hiding details of host and device memory spaces. The programmer can use a single managed pointer for accessing data from host or device address space. Data is transparently transfered from an address space to the other. The main drawback of this approach is that kernel synchronization

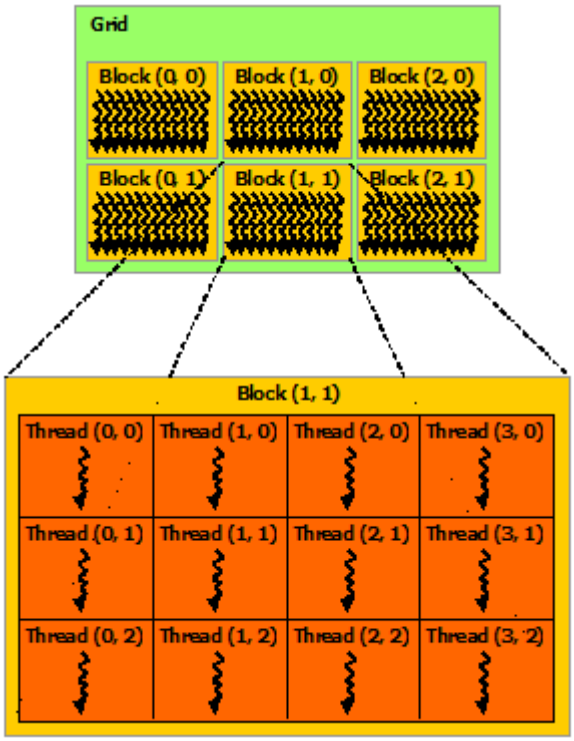


Figure 2.1: Grid of thread blocks [24].

must always be performed with *cudaDeviceSynchronize()* after every kernel launch, which harms some optimization strategies.

CUDA Toolkit v7 introduced the concept of independent streams for allowing concurrent tasks to be executed on GPUs [15]. Commands within a stream execute sequentially, but different streams execute concurrently. Commands that support independent streams include kernel launches and memory copies.

2.1.1 GPUDirect

GPUDirect is a set of CUDA features that allows for efficient data movement between GPUs and network adapters as well as between different GPUs, as long as all devices are connected to the same PCIe (Peripheral Component Interconnect Express) bus [27, 36].

Before the release of the GPUDirect, transferring data from a GPU to a network adapter always required host intervention and at least three buffer copies. First, GPU copies data to a pinned memory buffer on host. Then, host copies data from the pinned buffer to a buffer that is accessible by the network adapter. Finally, the network adapter copies data to its own memory. The GPUDirect 1.0, released in 2010, allows GPUs and network adapters to share the same pinned memory region on host, which eliminates the intermediate memory copy made by the host.

With CUDA Toolkit v4.0, the GPUDirect Peer-to-Peer (P2P) technology was introduced. It allows for high-speed Direct Memory Access (DMA) transfers between GPUs connected on the same PCIe bus. In addition, it allows P2P NUMA-style memory access between GPUs from within CUDA kernels.

CUDA Toolkit v5.0 introduced the GPUDirect RDMA (Remote Direct Memory Access), which completely bypass copy to host memory, allowing RDMA transfers between GPUs and other PCIe devices. As a consequence, CPU bandwidth and latency bottlenecks can be eliminated.

2.1.2 CUDA-Aware MPI

MPI (Message Passing Interface) is a standardized API for communicating data between distributed processes via message passing. It is frequently used to build HPC applications that can scale on computer clusters [28].

Regular MPI implementations can only handle pointers to host memory. Therefore, in order to transmit data stored in a GPU memory using functions such as *MPI_Send()*, it is required to copy the data into the host memory via *cudaMemcpy()*. Similarly, when data is received via *MPI_Recv()*, *cudaMemcpy()* must be used again to copy data from the host memory into the GPU memory.

On the other hand, CUDA-aware MPI libraries make use of GPUDirect to send and receive GPU buffers directly, without staging them first in host memory. Besides being easier to use, removing this additional memory copy can improve communication performance and decrease latency. In addition, CUDA-aware MPI libraries usually encapsulate other technologies and strategies to improve communication efficiency, e.g. GPUDirect P2P, GPUDirect RDMA and message transfer pipelining.

Some CUDA-aware MPI libraries currently available for HPC applications are: MVAPICH2 [18], Open MPI [46] and IBM Platform MPI [16].

2.2 Lattice Boltzmann Method

The Lattice Boltzmann Method (LBM) is a Computational Fluid Dynamics (CFD) method widely used in academia and industry. It emerges as an alternative to Navier-Stokes equations that can scale much better in massively parallel supercomputers [20].

The domain of a LBM simulation is discretized as a lattice, whose nodes represent cells in a cellular automaton. Every cell is a Particle Distribution Function (PDF) that represents fluid propagation in determined directions. The exact representation of a PDF depends on the lattice model being considered for the simulation. In this work, the lattice model considered is the D3Q19, which represents a 3D PDF with 19 propagation directions, as depicted in Figures 2.2. The value of every PDF direction can represented in a cell as a float point. Hence, a D3Q19 domain can be represented as a 4D regular grid with (x, y, z, f) coordinates, where the first three dimensions correspond to 3D space and the fourth dimension represents the 19 PDF values. In addition to the 4D grid, the lattice model also defines a weight for each direction.

As a cellular automaton, a LBM simulation consists of a loop where every iteration is a simulation step. In every simulation step, the PDFs are calculated according to the corresponding lattice model whose directions define a stencil. There are basically three major routines involved in a simulation step:

1. **Stream**: In this routine, PDF values propagate to neighboring cells, which are defined by the stencil directions. The D3Q19 stencil, for instance, defines that a fluid cell has 18 neighboring cells to propagate values.

2. **Collide**: This routine defines a particle collision operation for every cell. Two common collision operator are the SRT (Single Relaxation Time) and MRT (Multiple Relaxation Time) [20, 42].

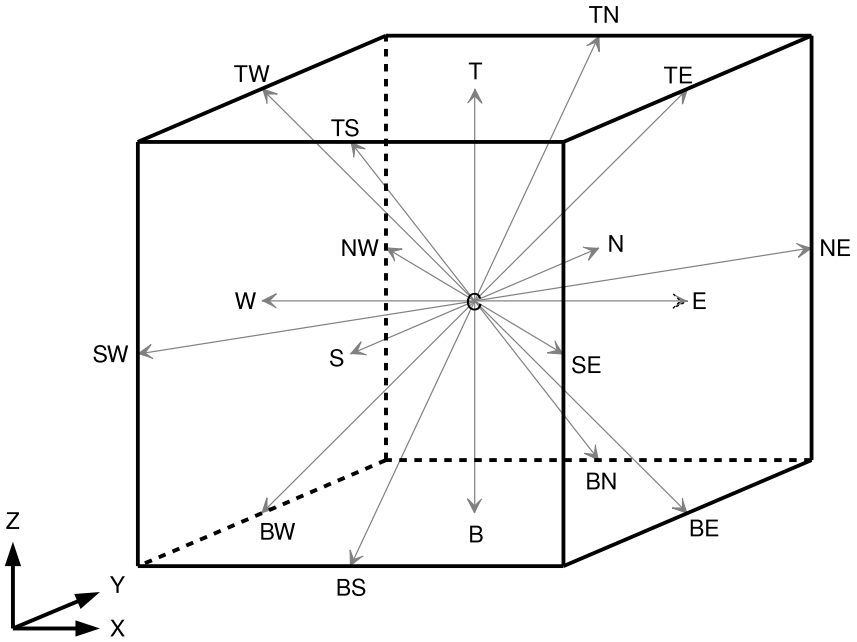


Figure 2.2: The D3Q19 lattice model [31].

3. **Boundary handling**: The fluid boundaries require special handling as their cells do not have neighboring cells in all stencil directions necessary for calculating the stream routine. Two common boundary handling strategies are the *noSlip* and the *freeSlip*.

The key LBM characteristic that makes it scale very well on massively parallel architectures is the local dependency of data when calculating a simulation step for a given cell. As only the own cell data and data from its neighbors are required for cell calculation, when decomposing a simulation domain into uniform blocks, i.e. axis-aligned blocks with the same dimensions, in order to split the domain processing among a set of connected computing nodes, only cells in the block boundary require communication of data from neighboring blocks. The block topology follows the same neighboring pattern as cells do, i.e. the neighbors of a block is also defined by the stencil.

2.3 Lid-Driven Cavity

The lid-driven cavity (LDC) is a typical benchmark problem for CFD applications, including LBM [20]. It is a classic idealization of recirculation flows with numerous environmental, geophysical and industrial applications [38].

The 3D LDC problem consists of a steady fluid in a cubic cavity, where all boundaries are stationary except for the upper boundary, where a constant velocity is imposed in x direction, as depicted in Figure 2.3.

When running an LDC simulation, as more time steps are executed, the top velocity creates a flow towards the east (right-hand) boundary, then fluid starts to flow towards the bottom boundary and so on. After a large amount of time steps, eventually a central vortex can be identified in the cavity as depicted in Figure 2.4.

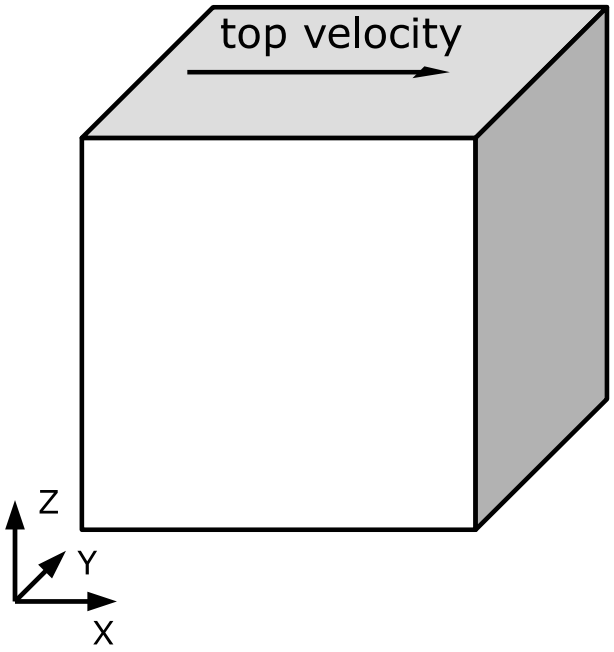


Figure 2.3: Representation of the 3D LDC problem.

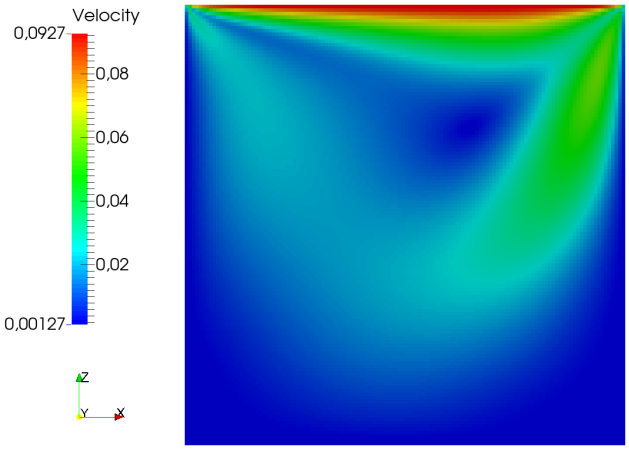


Figure 2.4: Resulting velocity in the y midplane of a LDC simulation.

2.4 waLBerla Framework

waLBerla (Widely Applicable Lattice Boltzmann from Erlangen) is a massively parallel software framework created and maintained by the Department of System Simulation of the Friedrich-Alexander-Universität Erlangen-Nürnberg [5]. It supports a wide range of physical phenomena [7] and runs on some of the top 10 HPC clusters in the world [11].

The waLBerla framework has already been refactored two times during its development. A previous refactoring of the framework, developed in 2012, was specialized in GPU clusters [13], however it is not being maintained anymore because its architecture did not suit the

requirements for the new features planned for the framework. This work is based on the active architectural refactoring of the framework, which referred in this text for simplicity as the base version.

waLBerla is mainly centered around the LBM, but its applicability is not limited to this family of algorithms [8]. Although it provides classes to help implementing algorithms based on LBM, its design is sufficiently generic to make it suitable for implementing a variety of stencil based applications. In order to split the simulation computation among the cluster nodes, it employs a block partitioning of the simulation domain.

The framework is programmed in C++11 and implements algorithms that take advantage of HPC clusters and hardware specific features. It makes extensive use of C++ templates in order to handle user defined data types while keeping an optimal performance.

2.4.1 Application Structure

As a framework, waLBerla is designed so that users, i.e. application programmers, can develop applications by extending the framework's basic functionalities. HPC applications developed using waLBerla are usually structured as following (detailed information about the mentioned classes can be found in Section 2.4.2):

1. **Creation and initialization**: The domain data structure is created and initialized. Typically, the domain is decomposed into axis-aligned blocks represented by the *IBlock* class. Blocks themselves are containers for fields, the actual simulation data. Fields can be represented by an user defined data structure or by classes provided by the framework, such as *Field*, *GhostLayerField* or *GPUField*.

2. **Operation definition**: The user is responsible to define a sequence of operations that will be executed on the domain data in every step of the simulation. The user can implement some of the operations, but the framework already provides a set of convenient operations that the user can also select. In a typical application, the operations are classified as following:

   - **Communication**: Communicate data between neighboring blocks. Communication can occur locally, i.e. in the same process, or remotely, i.e. between different process, although most of the complexity involving different types of communication are transparent to the user. Remote communication usually involves MPI. A set of communication schemes with varied purposes is provided by the framework, such as *UniformDirectScheme* and *GhostLayerField* copy.
   - **Boundary handling**: Operate on lattice cells at the domain boundary. The framework provides some common boundary handling operations for LBM, such as *no slip* and *free slip*.
   - **Sweep**: Operate on lattice cells other than boundary cells. It is typically implemented by the user via a functor class referred in this work as *BlockSweep*.

3. **Simulation loop**: Execute the defined operations in a loop for a given number of steps, i.e. iterations. The infrastructure behind the simulation loop is provided by the framework and manages the sequence of operations among all simulation processes. The framework is capable of allocating blocks on multiple processes that can be placed on multiple nodes. A process can run one or more blocks, but a block is always completely processed by a single process.

Listing 2.1 shows the structure of a very simple example of an application using waLBerla:

Listing 2.1: Structure of a simple application using waLBerla.

```
// Creation and initialization
shared_ptr<StructuredBlockForest> blocks = createUniformBlockGrid (
      2, 2, 2,        // number of blocks in x, y, z directions
      200, 200, 200, // number of cells per block in x, y, z directions
      1,              // length of one cell in physical coordinates
      true,           // run each block in a separate process
      false, false, false);  // domain periodicy in x, y, z directions
BlockDataID fieldId = blocks->addStructuredBlockData<
      GhostLayerField<double, 19> >(createGhostLayerField, "Field");

// Operation definition
SweepTimeloop timeloop(blocks, numberOfTimesteps);
UniformBufferedScheme<stencil::D3Q19> communicationScheme(blocks);
timeloop.add() << BeforeFunction(communicationScheme, "Communication")
      << Sweep(BoundaryHandling(fieldId), "BoundaryHandling");
timeloop.add() << Sweep(BlockSweep(fieldId), "Sweep");

// Simulation loop
timeloop.run();
```

2.4.2 Framework Architecture

This section presents the architecture of the base version of waLBerla. The architecture is presented in a simplified way, focusing only in aspects that are relevant for the understanding of the proposed solutions.

The waLBerla framework provides a set of classes and functions for coding HPC applications. In order to handle user defined data types while keeping a good performance, many of the framework classes are template classes. The class diagram depicting the main classes that are required to implement stencil based applications as well as their relationship are shown in Figure 2.5.

A description of each class depicted in Figure 2.5 is following:

- **IBlock**: This class represents an axis-aligned rectangular decomposition of the domain and manages all data that lies in that partition. It is a container for the actual simulation data. Each block can be attributed to a different processing unit through schemes provided by the framework, e.g. *UniformBufferedScheme*.
  - *getData()*: Retrieve simulation data stored in the *IBlock*, usually a *Field* or its derived classes, a *GPUField*, or a user defined type.
- **StructuredBlockForest**: This class manages a structured set of *IBlock*s resulted from a domain decomposition. It is a facade that provides a simplified interface to a complex system of classes that actually stores the *IBlock*s.
  - *addStructuredBlockData()*: Initialize data of every *IBlock* managed by the class.
- **Field**: This is the base class for storing lattices in CPU main memory. The stored lattice data corresponds to a single block of a decomposed domain. It is structured as a four-dimensional grid with (x, y, z, f) coordinates, where the first three dimensions correspond

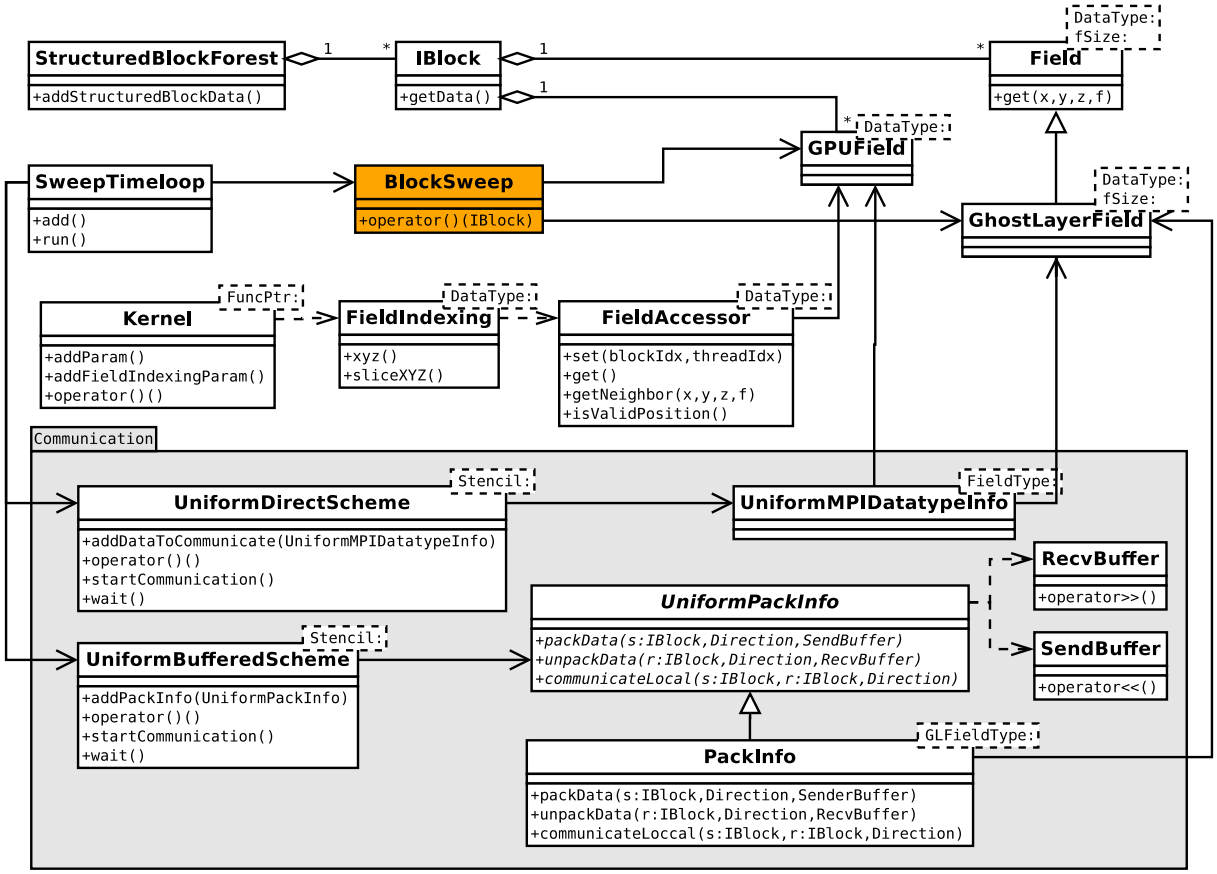


Figure 2.5: Class diagram for an application using the base version of waLBerla. Some intermediary classes and methods are omitted for simplicity. The class *BlockSweep* in orange is defined by the user. All remaining classes are part of the waLBerla framework. Classes directly involved in communication are grouped into the *Communication* package.

to 3D space and the fourth dimension is used for indexing within a cell. In LBM, the fourth dimension is usually used for indexing the particle distribution functions (PDFs). It is implemented as vector with cells arranged contiguously in memory by default, or padded for memory alignment. As depicted in Figure 2.6, two different memory layouts are supported: *fzyx* and *zyxf*. The *DataType* template parameter represents the data type stored by the class. The *fSize* template parameter is the size of the f coordinate.

  - *get(x, y z, f)*: Return a reference to a lattice cell value given its coordinates.

- **GhostLayerField**: This class extends *Field* with a ghost layer. A ghost layer is a layer of additional cells around the lattice cells. The ghost layer cells store locally the cell values from the neighboring lattices that a stencil requires. The framework updates the value of the ghost layer cells at every time step, which requires communication. The template parameters follow those of its base class.

- **GPUField**: This class is a field for storing lattices in GPU memory. Like *Field* and its derived classes, it is also structured as a four-dimensional grid with (x, y, z, f) coordinates. It is implemented as a 3D vector in GPU memory and only supports padded memory, which is allocated and automatically aligned by *cudaMalloc3D()*. This kind of padded

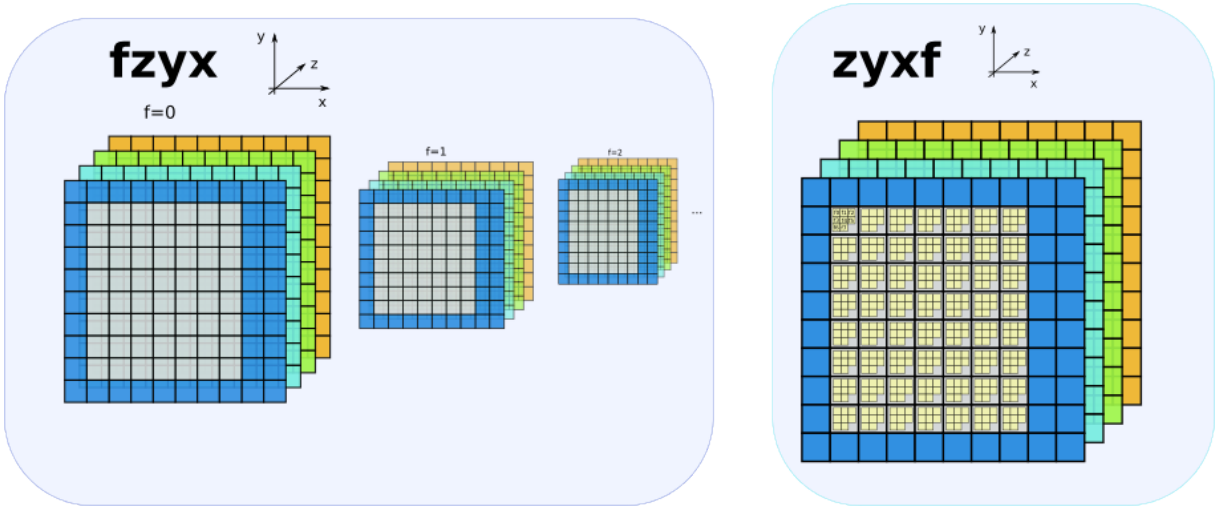


Figure 2.6: *fzyx* and *zyxf* memory layouts that are supported by field classes such as *Field*, *GhostLayerField* and *GPUField* [4]. The relative position of the dimensions in memory are specified in the layout names, where reading from left to right are specified the outmost to the innermost position in memory. That is, in the *fzyx* layout, f is the outmost and x the innermost, and in the *zyxf* layout, z is the outmost and f the innermost. As a consequence, contiguous elements are in x dimension for *fzyx* layout and in f dimension for *zyxf* layout.

memory is often referred as pitched memory [24]. Like *GhostLayerField*, it contains a ghost layer for storing locally cell values from the neighboring blocks. Both memory layouts *fzyx* and *zyxf* are supported. The *DataType* template parameter represents the data type stored by the class.

- **Kernel**: This is a CUDA kernel wrapper that is required to launch kernels from code not compiled with *nvcc* (NVIDIA CUDA Compiler). The wrapper is required because waLBerla makes use of many C++11 features that were not properly supported by *nvcc* when the GPU support was introduced in the framework. The *FuncPtr* template parameter is the type of the kernel function to be called.
  - *addParam()*: Add parameter to the kernel.
  - *addFieldIndexingParam()*: Add an indexed *GPUField* parameter to the kernel, where indexing is made by the *FieldIndexing* class.
  - *operator()()*: Call the CUDA kernel.
- **FieldIndexing**: This class conveniently maps (x, y, z, f) coordinates of a *GPUField* to CUDA's block and thread coordinates. Mapping depends on which memory layout is set on *GPUField*. For *fzyx* memory layout, mapping is done so that the *GPUField*'s coordinates x, y, z and f are mapped to CUDA's coordinates *threadIdx.x*, *blockIdx.x*, *blockIdx.y* and *blockIdx.z*, respectively. For *zyxf* memory layout, on the other hand, mapping is done so that the *GPUField*'s coordinates x, y, z and f are mapped to CUDA's coordinates *blockIdx.x*, *blockIdx.y*, *blockIdx.z* and *threadIdx.x*, respectively. The mapping created by this class is kept by the *FieldAccessor* class, which allows for accessing the *GPUField* data from inside a CUDA kernel. The *DataType* template parameter follows the one specified on *GPUField*.

  - *xyz()*: Create a mapping for all cells of a *GPUField*, so that a CUDA thread is allocated for each *GPUField*'s cell.
  - *sliceXYZ()*: Create a mapping for cells lying within a given slice of a *GPUField*, so that a CUDA thread is allocated for each *GPUField*'s cell within the slice.

- **FieldAccessor**: This class allows CUDA kernels to access data from *GPUField* instances. Access is made so that every *GPUField* element is conveniently mapped to a CUDA thread as defined by *FieldIndexing*. The *DataType* template parameter follows the one specified on *GPUField*.

  - *set(blockIdx, threadIdx)*: Set up the mapping of coordinates given the CUDA indexes *blockIdx* and *threadIdx*. It must be called before any other method call inside the kernel.
  - *get()*: Return the *GPUField*'s element mapped to the current kernel.
  - *getNeighbor(x,y,z,f)*: Return the *GPUField*'s element placed on the (x, y, z, f) relative coordinates to the element mapped to the current kernel.
  - *isValidPosition()*: Verify whether or not the element mapped to the current kernel is inside the *GPUField*'s domain limits.

- **BlockSweep**: This is not a class provided by waLBerla but a functor class that the user must implement in order to operate on every *IBlock*. The framework is responsible for traversing every *IBlock* instance managed by the *StructuredBlockForest* and pass its pointer to the *BlockSweep::operator()()*. The user can choose any name for this class.

  - *operator()(IBlock)*: This operator is called by the framework, receiving an *IBlock* pointer as argument. The programmer must code in this operator the routines to operate on the *IBlock*'s data, which is usually a lattice type such as *Field*, *GhostLayerField* or *GPUField*.

- **SweepTimeloop**: This class manages the execution of the time steps. It allows the addition of functions or functors to be called at every time step during the simulation execution. Therefore, in order to make the framework execute the operations coded in *BlockSweep::operator()()*, the user must register a *BlockSweep* instance into a *SweepTimeloop* instance.

  - *add()*: Register functions or functors to be executed at every time step.
  - *run()*: Start the execution of a simulation.

- **UniformPackInfo**: This is a communication abstract class that encapsulates key operations used in the communication between *IBlock*s. For communication between blocks owned by different processes, i.e. for remote communication, it encapsulates pack and unpack operations. For communication between blocks owned by the same process, i.e. for local communication, it encapsulates a local copy operation. Those operations must be implemented by a concrete derived class for performing data transfer between specific types of field. Implementations should proceed so that the proper data segments of a field are sent to the ghost layers of the neighboring blocks.

  - *packData(sender:IBlock, Direction, SendBuffer)*: This is the abstract method for packing data from the *sender* block into the *SendBuffer* for communication with the neighboring block towards the given stencil *Direction*.

  - *unpackData(receiver:IBlock, Direction, RecvBuffer)*: This is the abstract method for unpacking data from the *RecvBuffer* into the ghost layer of the *receiver* block, where data was sent by the neighboring block towards the given stencil *Direction*.
  - *communicateLocal(sender:IBlock, receiver:IBlock, Direction)*: This is the abstract method for local communication, where data from the *sender* block is copied into the ghost layer of the *receiver* neighboring block that lies in the given stencil *Direction*.

- **PackInfo**: This is a concrete specialization of *UniformPackInfo* that implements *packData()*, *unpackData()* and *communicateLocal()* operations for communication between *GhostLayerField*s, i.e. between fields stored in the main CPU memory. The *GLFieldType* template parameter is a fully qualified *GhostLayerField* type, i.e. a *GhostLayerField* with all its template parameters defined.

- **UniformBufferedScheme**: This is a communication scheme for a domain decomposed into uniform blocks, i.e. axis-aligned blocks with the same dimensions. It updates the ghost layers of all fields in the domain with information from the neighboring blocks through synchronous or asynchronous communication. The *Stencil* template parameter defines the communication neighboring topology.

  - *addPackInfo(UniformPackInfo)*: Register a *UniformPackInfo*. Regarding the presented class hierarchy, the concrete *UniformPackInfo* must be a *PackInfo*.
  - *operator()()*: Communicate synchronously.
  - *startCommunication()*: Start an asynchronous communication.
  - *wait()*: Wait an asynchronous communication completion.

- **SendBuffer**: This is a convenient representation of a buffer for outgoing MPI messages.

  - *operator<<()*: Add a data value to the buffer. Calls to this operator can be chained for convenience.

- **RecvBuffer**: This is a convenient representation of a buffer for incoming MPI messages.

  - *operator>>()*: Extract a data value from the buffer. Calls to this operator can be chained for convenience.

- **UniformDirectScheme**: Like *UniformBufferedScheme*, this is also a scheme for uniform block grid communication. However, it allows fields to communicate with the neighboring blocks directly via MPI data types rather than a *UniformPackInfo* class. To this end, it makes use of CUDA-aware MPI technology. Synchronous and asynchronous communication are supported. Fields of an arbitrary type are allowed as long as they have ghost layers for communication. The *Stencil* template parameter defines the communication neighboring topology.

  - *addDataToCommunicate(UniformMPIDatatypeInfo)*: Register an MPI data type through a *UniformMPIDatatypeInfo*.
  - *operator()()*: Communicate synchronously.
  - *startCommunication()*: Start an asynchronous communication.
  - *wait()*: Wait an asynchronous communication completion.

- **UniformMPIDatatypeInfo**: This class encapsulates the MPI data types from the field type given by the *FieldType* template parameter. Fields with ghost layer such as *GhostLayerField* and *GPUField* are supported. The MPI data types of a field allow the decomposition blocks to communicate without the buffers *SendBuffer* and *RecvBuffer*.

In the base version of waLBerla, there are basically two options for GPU communication: *UniformDirectScheme* and *GhostLayerField* copy. Although both are functional, as discussed in Section 6.2.2, tests have shown that they have poor performance, which prevents the user from implementing efficient HPC applications using GPUs.

The *UniformDirectScheme* communication can be set up by passing a fully qualified *GPUField* type to the template parameter of a *UniformMPIDatatypeInfo*, then adding it to an instance of the *UniformDirectScheme* class via the *addDataToCommunicate()* method.

The *GhostLayerField* copy communication can be implemented by first setting up an ordinary CPU communication for *GhostLayerField*s, creating *GPUField*s for processing data on GPU, then defining copy operations between *GhostLayerField* and *GPUField* to be performed at every step of the simulation, as shown in the sequence diagram in Figure 2.7. The required CPU communication can be set up by passing a fully qualified *GhostLayerField* type to the template parameter of a *PackInfo*, then adding it to an instance of a *UniformBufferedScheme* class via the *addPackInfo()* method.

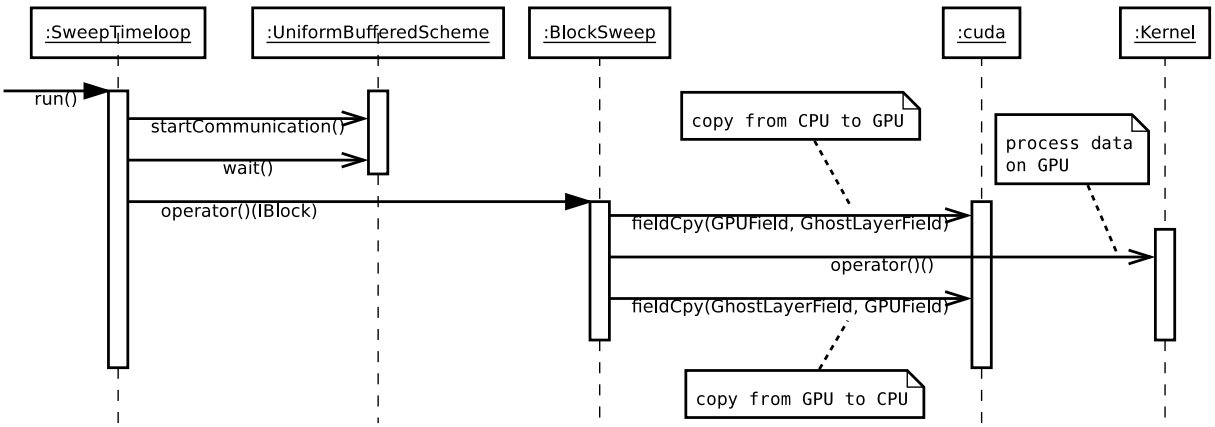


Figure 2.7: Sequence diagram for *GhostLayerField* copy communication. *SweepTimeloop* starts communication on the *UniformBufferedScheme* and waits for the communication completion. In *startCommunication()*, data is exchanged between the neighboring *GhostLayerField*s. The *BlockSweep::operator()()* is then called, copying the field data from *GhostLayerField* to *GPUField* via the *cuda::fieldCpy()* function. Then, the CUDA kernel is executed by calling *Kernel::operator()()* in order to process field data on GPU. Finally, the resulting data is copied from *GPUField* back to the *GhostLayerField*.

Chapter 3

Literature Review

This chapter presents relevant literature for the research presented in this dissertation, including works focused on kernel optimization, GPU communication and HPC framework architecture.

3.1 Optimized Kernels

This section presents literature review of works focused on single block GPU simulations with optimized LBM kernels.

Habich et al. [13] presents an optimized D3Q19 LBM solver implemented for GPUs using both, CUDA and OpenCL. The presented optimizations and developed experiments are focused on single block LBM. The optimization techniques are focused on improving memory access since it was identified that the implemented algorithm is limited by memory bandwidth. The LDC problem was used for verifying the presented optimizations and reached about 152 MFLUPS (Million Fluid Lattice Updates per Second) on an NVIDIA Tesla C2070 with double precision, contiguous memory, domain size of 200^3 cells and GPU's ECC (Error-Correcting Code) enabled.

A single block LDC application implemented with CUDA using the base version of waLBerla is introduced by Sepka [42]. The presented CUDA kernel showed to be highly optimized, specially with contiguous memory and GPU's ECC enabled. For a domain size of 200^3 cells, the LDC application reached 159 MLUPS (Million Lattice Updates per Second) on an NVIDIA Tesla C2075. For a domain size of 128^3 cells, it performed 413 MLUPS on an NVIDIA Tesla K40m.

3.2 GPU Communication

This section presents literature review of works focused on GPU-to-GPU communication.

A GPU-aware MPI design based on MVAPICH2 for simplifying programming on GPU clusters is introduced by Wang et al. [48]. The design can handle both contiguous and non-contiguous GPU data. As performance issues caused by non-contiguous data access were identified, the proposed design transforms MPI datatypes into vectors and uses CUDA kernels for packing and unpacking. The described approach also uses pipeline for overlapping GPU data pack/unpack operations, DMA data movement on PCIe and GPUDirect RDMA data transfers. The paper presents benchmarks with an optimized 3D LBM application on GPU clusters. On

the Oakley supercomputer, it was observed up to 19.9% improvement in application level performance for 64 GPUs.

Habich et al. [12] presents optimization strategies for an LBM solver with a D3Q19 stencil on NVIDIA GPUs, focusing on memory alignment and register shortage. It makes use of the STREAM benchmark [19] in order to demonstrate the GPU parallelization approach and obtain an upper limit for implementation performance. According to the paper, the optimized code is up to one order of magnitude faster than standard two-socket x86 servers with AMD Barcelona or Intel Nehalem CPUs. The potential benefits of multi-GPU parallelism in computer clusters is analyzed based on PCIe data transfer rates. The paper concludes that PCIe bus as well as the InfiniBand network have a substantial impact on performance since the communication can take as long as twice the computational time spent on the GPU. Consequently, the communication is the major limiting factor for the overall performance.

Shainer et al. [43] discusses issues involving the GPU-CPU-InfiniBand system architecture, which usually requires the CPU to initiate and manage memory transfers between remote GPUs via a high-speed InfiniBand network. The paper introduces for the first time the GPUDirect RDMA technology, which enables the Tesla GPUs to transfer data via InfiniBand without the involvement of the CPU or buffer copies, hence reducing the GPU communication time and increasing overall system performance. Performance was tested running two distinct molecular dynamics simulation programs, Amber [3] and LAMMPS [34], in a cluster with 8 nodes, each with a single NVIDIA Fermi GPU and Mellanox ConnectX-2 InfiniBand adapter. It was demonstrated that GPUDirect RDMA can improve communication performance up to 33%.

Potluri et al. [36] verified that communication is a critical bottleneck in achieving the full potential of GPU-based supercomputers. Although MPI libraries such as MVAPICH2 have provided solutions to alleviate communication overhead by using techniques like pipelining, data in the GPU memory still has to be moved into the CPU memory before it can be sent over the network. This paper also presents the GPUDirect RDMA as a solution to eliminate this overhead. Besides evaluating the first version of GPUDirect RDMA for InfiniBand, the paper proposes enhancements in the MVAPICH2 MPI library to efficiently take advantage of it for inter-node GPU-GPU communication. The benchmark performed has shown that the proposed functionalities improved the inter-node GPU-GPU communication latency on using *MPI_Send* and *MPI_Recv* on a Sandy Bridge platform by 69% and 32% for 4-Byte and 128-KByte messages, respectively, in comparison with an out-of-the-box MVAPICH2. In addition, the enhancements boosted the unidirectional bandwidth achieved using 4-KByte and 64-KByte messages by 100% and 35%, respectively. The impact of the proposed enhancements was also demonstrated using two applications, GPULBM [40] and AWP-ODC [33], showing improvements in their communication time by up to 35% and 40%, respectively.

Different approaches for using multiple GPUs in the simulation of physical systems are presented and compared by Bernaschi, M. et al. [2]. The benchmarks adopted are 3D Heisenberg spin glass model and a solution of Poisson equation. The results showed that the GPUDirect P2P memory copy along with the CUDA-aware MPI features of libraries such as MVAPICH2 and Open MPI can hide the communication overhead in case computation time is long enough. The most valuable contribution of this paper is the fact that it is one of the few works that compares the CUDA-aware MPI performance of MVAPICH2 and Open MPI libraries. The paper demonstrates that MVAPICH2 can take better advantage of CUDA-aware MPI features than Open MPI. Besides, MVAPICH2 can achieve higher parallel efficiency compared to Open MPI. In a naive case involving only intra-node communication with 8 GPUs, MVAPICH2 achieved 76% of parallel efficiency while Open MPI achieved 65%. In a more complex scenario involving inter-node communication with 8 nodes, one GPU per node and overlap between communication

and computation, MVAPICH2 achieved 80% of parallel efficiency while Open MPI achieved 73%.

A method for minimizing the GPU communication time of stencil based applications with a non-blocking InfiniBand interconnection network is presented by Schneible et al. [41]. It is affirmed that although minimizing the surface of the lattice decompositions increases communication performance by reducing message size, this approach is optimal only for naive cases where bandwidth is assumed to be equal between all GPUs and latency is assumed to be negligible. On more realistic systems, the intra-node communication between GPUs has greater bandwidth and lower latency than inter-node communication. As a consequence, the optimal solution to split the lattice is not obvious and depends upon the application and hardware parameters. The paper demonstrates that it is impractical to determine the optimal lattice decomposition through brute force method even for moderate sized applications. Thus, the method the paper proposes is focused on reducing communication time rather than communication size, demonstrating a significant performance increase over the minimal surface technique in certain cases. This method splits the performance model into two models: application model and communication model. The former is at a high level and takes into account the application parameters and node architecture. The second, on the other hand, is a GPU cluster communication model that takes into account the topology of the interconnections considering every combination of intra-node and inter-node communication with and without GPUDirect 1.0 and GPUDirect P2P. By improving overlapping of communication and computation, the proposed method was able to increase the performance of the most computationally intensive kernel of a lattice Quantum Chromodynamics (QCD) implementation by 25%. In addition, the paper concludes that while the optimal heterogeneous system design for stencil based applications has only a single GPU per node, this does not offer the best performance per dollar. Instead, the most cost efficient systems have two GPUs per node.

The use of multiple GPUs as well as heterogeneous processing and communication for parallel computation of an LBM solver using an early version of the waLBerla framework is addresses by Feichtinger [8]. It affirms that the waLBerla framework is able to attain good runtime performance on various platforms despite the inevitable overheads for flexibility. The paper shows that scalable multi-GPU implementation for HPC clusters can be achieved for LBM simulations. It states that the kernel performance can be sustained for weak scaling on InfiniBand clusters but using multiple GPUs in strong scaling scenarios is much less efficient than running CPU-only simulations on IBM BG/P and x86-based clusters. Optimizations in GPU memory access based on padding strategies to favor memory coalescence are suggested as a future work.

3.3 HPC Framework Architecture

This section presents literature review of works focused on architectural aspects of HPC software frameworks.

The architecture of the base version of waLBerla was presented by Godenschwager et al. [11]. waLBerla's design was driven by a performance analysis of petascale supercomputers. The low-level parts of the framework make extensive use of template programming techniques for achieving high performance while maintaining flexibility. In high-level, the non-performance critical parts make use of polymorphism to ensure flexibility and usability. The paper also presents the framework support for domains with arbitrary shapes. Each block of the domain can be recursively decomposed into eight equally sized smaller blocks geometrically resulting in a forest of octrees. The domain initialization process is also described highlighting the advantages of some waLBerla features compared to other frameworks. Results of the framework

performance are presented for varied implementations of 3D LBM running on SuperMUC and JUQUEEN supercomputers, respectively the number 27 and number 13 on the Top500 list of June 2016 [45]. Resolutions with more than one trillion cells were reached with performance up to 1.93×10^6 MLUPS using 1.8 million threads. The simulation of a complex vascular geometry of the human coronary tree presented excellent weak and strong scaling performance. A record of a 10^{12} fluid lattice cells was achieved in the largest weak scaling simulations. Although this paper is important to understand key design features of the current waLBerla framework, it does not discuss any feature related to GPU support.

Sepka [42] proposed relevant future works for the base version of waLBerla in order to improve its usability on GPU-enabled clusters, including: a GPU communication mechanism, a mechanism for improving memory usage efficiency, enhancements in LDC kernels for handling decomposed domains and generic boundary handling kernels for LBM simulations. A drawback of the dissertation is the proposed solution for contiguous GPU memory for waLBerla, which involves a massive code duplication.

Chapter 4

GPU Engineering for an HPC Framework

The use of software frameworks is crucial to lower the development cost of efficient HPC applications as well as to increase the HPC adoption in various fields of science and engineering. Besides, the use of GPUs in HPC systems is a trend to lower costs while increasing processing power. Therefore, enhancing GPU performance and usability of a HPC framework such as waLBerla [6] is a valuable contribution.

Communication is a major performance bottleneck in harnessing the full potential of GPUs in HPC [35, 36]. Although the base version of waLBerla presents good performance on CPUs [11], preliminary tests have shown that its available GPU communication solutions perform poorly. Therefore, working to improve the GPU communication is an obvious choice to enhance the overall GPU performance for the framework. Besides, GPU communication is an important functionality that can be implemented in a generic way in the framework so that multiple applications can take advantage of it.

An efficient use of the GPU memory is also an important feature of an HPC framework as it allows using less cluster nodes for a given simulation or running simulations with larger domains. Thus, promoting a better usage of the GPU memory also contributes with waLBerla's applicability.

This chapter presents solutions to improve waLBerla performance and memory efficiency, including a new communication infrastructure and its associated programing interface for waLBerla. This new infrastructure employs state-of-the-art technologies and strategies in order to provide efficient inter-node and intra-node GPU communication. Software engineering and advanced programming techniques are employed in order to keep usability while improving performance. In addition, an application implementing a lid-driven cavity simulation was developed in order to verify the validation, performance and memory efficiency of the implemented solutions for the framework.

4.1 Architecture Overview

A major objective of the proposed architecture is allowing new functionalities and enhancements improve performance and memory efficiency while keeping usability and maintainability. To this end, the best practices in software engineering were considered when implementing the new functionalities, such as encapsulation, code reuse and extensibility.

The new functionalities proposed for waLBerla required extending the framework with new classes. As depicted in the class diagram in Figure 4.1, for developing a new GPU communication mechanism, the *GPUPackInfo* class was created, which is a new concrete class derived from *UniformPackInfo*. The base class *GPUSweepBase* was created allowing users to

derive *BlockSweep*s from it in order to easily improve memory management. Finally, in order to make field indexing in CUDA kernels more flexible, a new pair of indexing classes, *FieldIndex3D* and *FieldAccessor3D*, were also developed.

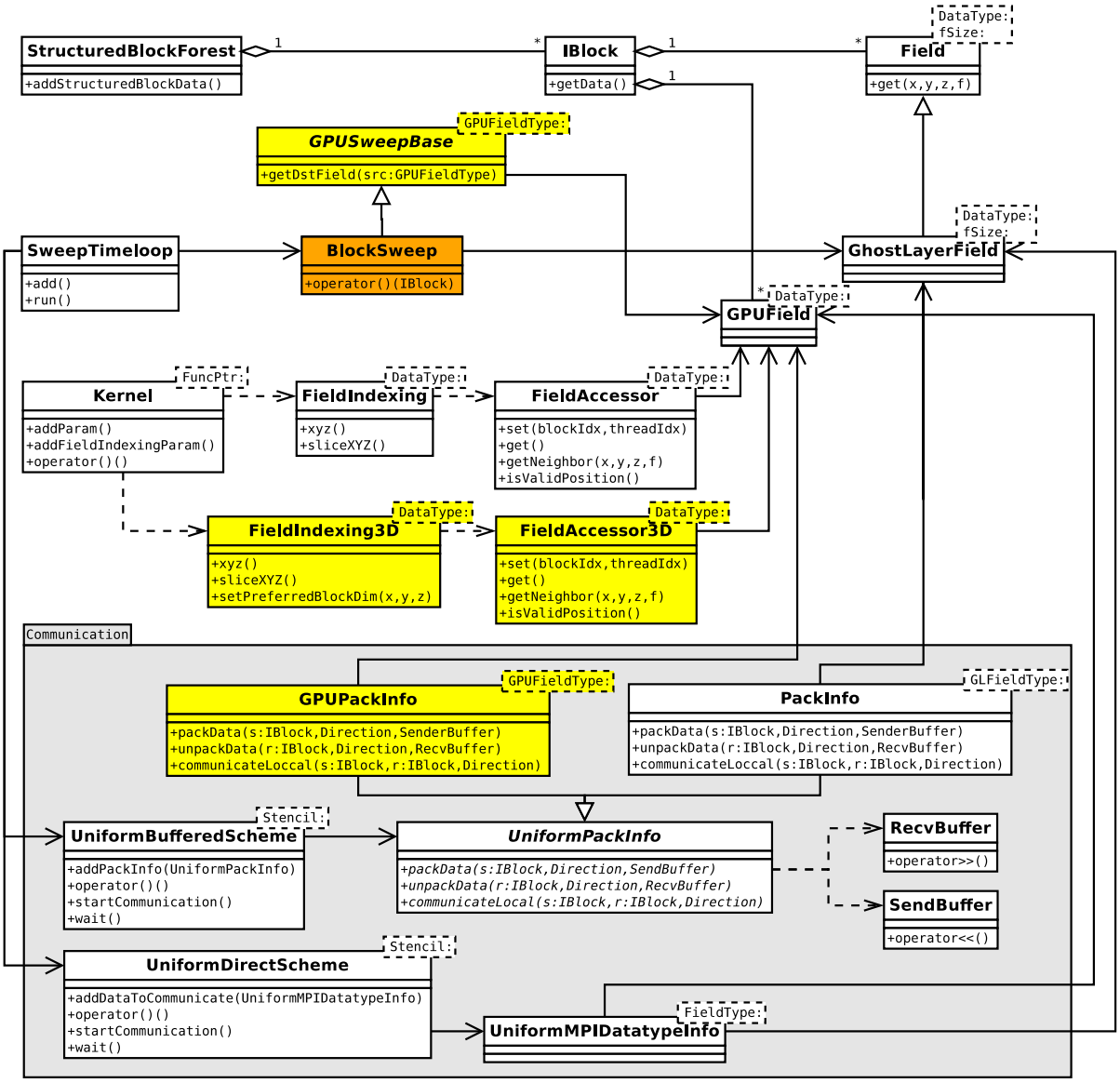


Figure 4.1: Class diagram for the new architecture for waLBerla. The new classes are highlighted in yellow. The class *BlockSweep* in orange is defined by the user. Some intermediary classes and methods are omitted for simplicity.

Details about the new classes developed for waLBerla can be found in the next sections of this chapter. *GPUPackInfo* is described in Section 4.2. *GPUSweepBase* is presented in Section 4.3.1. The pair of classes *FieldIndex3D* and *FieldAccessor3D* are described in Section 4.3.3.

4.2 GPU Communication Engineering

Improving GPU communication performance for waLBerla has become a major objective as the two GPU communication mechanisms available in base version of framework have shown poor performance in preliminary tests.

The *GhostLayerField* copy communication mechanism, as introduced in Section 2.4.2, needs to copy the whole domain data from CPU to GPU and vice versa on every simulation step. It was basically a temporary solution for allowing using *GPUField* for multi-node simulation while there was no specific solution for GPU communication at that time. Thus, it was never expected to be efficient.

The *UniformDirectScheme* communication mechanism, on the other hand, despite using CUDA-aware MPI and GPUDirect features, surprisingly showed a even worse performance in many scenarios. Tests showed that the poor performance arises when the Open MPI needs to communicate non-contiguous data types directly from a pointer to the GPU memory.

This section presents the solution for improving GPU communication performance, including design and implementation details.

4.2.1 Base Communication Architecture

Among some possible solutions to improve GPU communication performance, it was chosen to create a new GPU communication mechanism for the framework that reuses much of the architecture that has been used by CPU communication. Most of the proposed communication mechanism is encapsulated in the *GPUPackInfo* class and helper functions. Besides promoting performance improvements and code reuse, the new mechanism is also an important step for future work towards the development of heterogeneous computing involving GPU and CPU computing nodes.

For a better understanding of the proposed GPU communication mechanism, it is important analyzing how *UniformBufferedScheme* manages communication and its relation with *UniformPackInfo*, *RecvBuffer* and *SendBuffer*. Algorithm 1 describes in a very simplified way how classes collaborate in the *UniformBufferedScheme*'s method *startCommunication()* for allowing communication between every block and its neighbors via the abstract class *UniformPackInfo*.

Algorithm 1 *UniformBufferedScheme::startCommunication()* pseudocode.

```
1: uniformPackInfo: UniformPackInfo
2: sendBuffer: SendBuffer
3: recvBuffer: RecvBuffer
4: for each block ∈ IBlock do
5:    for each dir ∈ Stencil do
6:       neighbor ← block.getNeighbor(dir)   // get the neighboring block in dir direction
7:       if block and neighbor are owned by the same process then
8:          uniformPackInfo.communicateLocal(block,neighbor)
9:       else
10:         uniformPackInfo.packData(block,dir,sendBuffer)
11:         mpiSendAndReceive()   // communicate via MPI
12:         uniformPackInfo.unpackData(block,dir,recvBuffer)
13:      end if
14:   end for
15: end for
```

4.2.2 Communication Architecture Extension

In order to reuse the existing architecture, the proposed GPU communication solution consists of developing a concrete *UniformPackInfo* derived class, *GPUPackInfo*, for properly

packing/unpacking *GPUField*s data. Besides being compatible with *UniformBufferedScheme*, *GPUPackInfo* also allows for any other communication scheme that is compatible with *UniformPackInfo* to be promptly able to handle communication between GPU blocks.

As a concrete specialization of *UniformPackInfo* for communication data between *GPUField* block, *GPUFieldPackInfo* implements the *UniformPackInfo*'s abstract methods as following:

- *packData(sender:IBlock, Direction, SendBuffer)*: Pack data from the *sender GPUField* block into the *SendBuffer* for communication with the neighboring *GPUField* block in the given *Direction*. Packing is made through *cudaMemcpy3D()*.
- *unpackData(receiver:IBlock, Direction, RecvBuffer)*: Unpack data from the *RecvBuffer* into the ghost layer of the *receiver GPUField* block, where data was sent by the neighboring *GPUField* block towards the given *Direction*. Unpacking is made through *cudaMemcpy3D()*.
- *communicateLocal(sender:IBlock, receiver:IBlock, Direction)*: Copy data from the *sender GPUField* block into the ghost layer of the *receiver GPUField* neighboring block that lies in the given *Direction*. Copy is make through GPUDirect P2P memory transfer with *cudaMemcpy3D()*.

When coping data from a *GPUField* into the *SendBuffer*, the field's region that needs be by copied is a slice adjacent to the ghost layer in a given direction. The coordinates and dimensions of that slice are calculated using the *getSliceBeforeGhostLayer()* function. Similarly, when coping data from the *RecvBuffer* into the *GPUField*'s ghost layer, the ghost layer region in the given direction is calculated using the *getGhostRegion()* function.

SendBuffer and *RecvBuffer* are buffers in the CPU main memory for outgoing and incoming MPI messages. Both provide handy operators for helping data transfer. *SendBuffer* provides *operator<<()* for adding data to the buffer, while *RecvBuffer* provides *operator>>()* for extracting data from the buffer. However, those buffer operators were prepared only to handle pointers to the CPU main memory. Thus, enhancements were required for the buffer classes so data transferring data from GPU memory to the *SendBuffer* and from the *RecvBuffer* to the GPU memory can be done efficiently.

The solution for *SendBuffer* comes by creating a method *forward(uint_t elements)*, which makes the buffer skip the given number of elements so that *cudaMemcpy3D()* can be used to transfer data from the *GPUField* to the gap created in the buffer.

Similarly, the solution for *RecvBuffer* required changes in the existing method *skip(size_t elements)* in order to skip the given number of elements in the buffer and return a pointer to the beginning of the skipped region. Thus, *cudaMemcpy3D()* can be used to transfer data from the skipped region in the buffer to the *GPUField*'s ghost layer.

4.2.3 Data Transfer Details

GPUField represents a 4D structure in device memory with (x, y, z, f) coordinates. However, CUDA Runtime API supports directly objects with up to 3D dimensions [25]. Thus, additional effort is required when handling *GPUField*'s memory using CUDA Runtime API functions and data structures.

Regarding the *fzyx* memory layout, when allocating device memory with *cudaMalloc3D()* from the CUDA Runtime API for storing the *GPUField*'s lattice, it is required a memory

mapping for fitting the *GPUField*'s 4D structure into the allocated 3D memory object. It was done so that *GPUField*'s x and y dimensions are directly mapped into the memory object's x and y dimension, but *GPUField*'s z and f dimensions are allocated as a 2D structure in the memory object's z dimension, with *GPUField*'s z placed as the innermost dimension and f as the outmost dimension.

In *UniformPackInfo*, when data is transferred between a *GPUField* and one of the buffer classes, or between two *GPUField*'s, the *cudaMemcpy3D()* function from the CUDA Runtime API is employed. *cudaMemcpy3D()* is capable of copying any axis-aligned 3D region from a 3D memory object into another 3D memory object, where any of the memory objects can be placed on either device or host memory. Again, the 4D *GPUField*'s structure and its mapping to 3D memory object have to be taken into consideration and additional steps are required in order to copy a 3D region of a *GPUField* into the *SendBuffer*, copy the *RecvBuffer* into a 3D region of a *GPUField* or copy a 3D region of a *GPUField* into a 3D region of another *GPUField*. As long as the *GPUField*'s f dimension is mapped in device memory as the outmost dimension for *fzyx* memory layout, a copy of a 3D region of a *GPUField* is made slice by slice in f dimension using *cudaMemcpy3D()*. Thus, for the D3Q19 stencil, 19 *cudaMemcpy3D()* calls are required for a single 3D region copy.

4.3 GPU Memory Engineering

This section presents the developed solutions for waLBerla that enhances the GPU memory support in different aspects. The developed *GPUSweepBase* base class for *BlockSweep* provides a simple *GPUField* management for sweeps and improves memory usage efficiency. The support for contiguous memory developed for *GPUField* showed to have improved performance in many scenarios. Finally, the developed pair of classes *FieldIndex3D* and *FieldAccessor3D* provide a flexible way to index *GPUField*s in device memory.

4.3.1 Field Memory Management

In LBM and other stencil-based application, when a simulation step is calculated for a domain, the resulting data is usually stored in another domain. The former is referred as the source domain and the latter as the destination domain. The destination domain role is temporarily holding the calculations of a simulation step. Thus, in order to prepare of the next simulation step, source and destination domain pointers are swapped so that the source domain holds up-to-date data while the destination domain is ready for receiving the next step calculations.

As the calculation of a simulation step requires a source and a destination domain, it typically uses two times the amount of memory as required to store domain's data. However, by taking advantage of domain block decomposition, it is possible to use for the destination domain only a fraction of the amount of memory required by the source domain. A requirement for making it possible is allocating many blocks per process rather than a single block per process. To this end, waLBerla supports decomposing domains so that each process can own a group of blocks.

The proposed solution for promoting a better memory management takes into consideration that the blocks within the same process are calculated sequentially by a *BlockSweep* class. For uniform block decomposition where all blocks have the same size, instead of allocating a full sized domain as the destination domain, a single field with the dimension of a single source block plays the role of destination domain. Then, as soon as a simulation step is calculated

for a given source block, the pointers to source and destination fields are swapped so that the destination field is ready for holding the calculations of the next source block within the process. This is done until all source blocks owned by the process are calculated before proceeding to the next simulation step.

The presented memory management solution for the destination domain was encapsulated into the *GPUSweepBase* class as part of waLBerla. As depicted in Figure 4.1, a framework's user can derive a *BlockSweep* class from it in order to easily take advantage of its functionality. *GPUSweepBase* is also a template class whose template parameter *GPUFieldType* makes it flexible in terms of the specific *GPUField* type.

Provided that the user derives *BlockSweep* from *GPUSweepBase*, whenever the framework calls *BlockSweep::operator(IBlock)* in order to calculate a simulation step for the given block, all user's code has to do for getting the destination field is calling *getDstField(src)* passing the field of the source block as a parameter. By doing so, the application is ready to make a more efficient use of the device memory. Note that for a domain decomposed into B uniform blocks, the memory effectively used for the destination domain is $1/B$ of the memory required for the source domain.

In order to make the solution more generic in terms of domain decomposition, it was developed taking non-uniform block decomposition into consideration as well, where blocks can differ in sizes. To this end, *GPUSweepBase* keeps a collection of destination fields with unique dimensions. Whenever a source field is passed as a parameter to *getDstField(src)*, it is checked whether or not the destination field collection already owns a field with the same dimensions as *src*. In case it does, the field is returned. Otherwise, a new destination field with same dimensions as *src* is created, inserted into the collection and returned.

4.3.2 Contiguous Memory Support

Only pitched memory is supported by *GPUField* in the base version of waLBerla. However, some factors leaded to the decision of implementing support for contiguous memory as well: performance, memory usage and flexibility.

Frequently, pitched memory is recommended for allocations of data structures with two or more dimensions since it ensures that the allocation is properly padded to meet the alignment requirements for fast memory access [24]. Nonetheless, preliminary tests have shown that contiguous memory performs better than pitched memory in many scenarios, as can be observed in Figure 4.2. Besides, tests performed by Sepka [42] have also shown better performance for contiguous memory than pitched memory. Even though Habich et al. [13] has indeed shown better performance for pitched memory, that paper actually proposes a handcrafted pitched memory implementation for 128-byte alignment, which differs greatly from the 512-byte alignment that *cudaMalloc3D()* allocates for *GPUField*. Supporting a configurable handcrafted memory alignment for *GPUField* is proposed as a future work.

In addition, contiguous memory has no padding while pitched memory has paddings in the innermost dimension when it does not meet the memory alignment constraint. Depending on how many bytes that the innermost dimension size in bytes is apart from the multiple of 512 that can fit it, pitched memory paddings can occupy a large amount of the memory, which makes contiguous memory more efficient in terms of memory usage.

In order to improve the memory usage efficiency and simulation performance when using pitched memory, the domain sizes are often adjusted to better fit the memory alignment. However, provided that domain sizes meet the memory alignment constraint, contiguous memory has shown to perform as good as pitched memory.

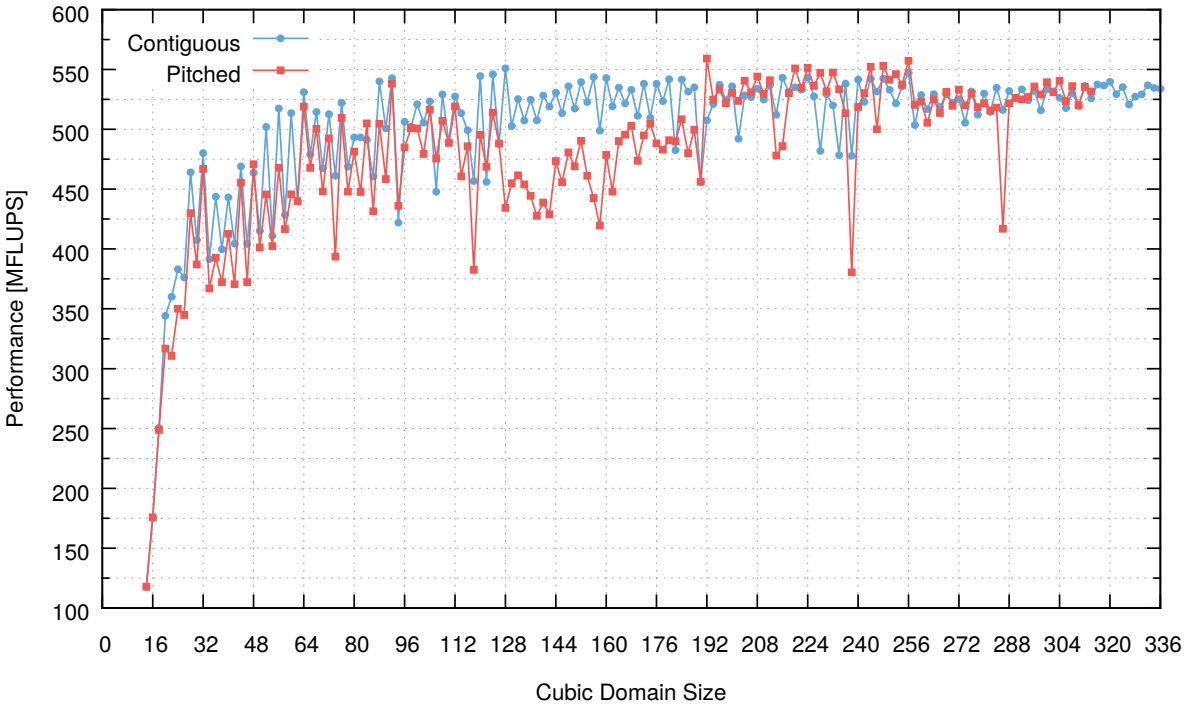


Figure 4.2: Preliminary performance comparison between contiguous and pitched memory. The test consist of running 200 iterations of the lid-driven cavity kernels, introduced in Section 4.4.1, in a GPU of the *cluster-ib* system, described in Section 5.1, for domain sizes varying from 14^3 to 336^3 cells. In order to mitigate warm-up interference in performance verification, every scenario is repeated 10 times and only the best performance is regarded. Note that, even though the best case occurred using pitched memory with domain size of 192^3 cells, contiguous memory performed better for most domain sizes. In addition, contiguous memory allowed simulating larger domains in the same GPU due to its better memory usage efficiency.

Sepka has proposed a contiguous memory solution for *GPUField* that involves a massive code duplication: classes such as *GPUField*, *FieldIndexing* and *FieldAccessor* as well as stand-alone functions such as *fieldCpy* were duplicated into pitched and contiguous counterparts. The main drawbacks of that solution is that it makes the waLBerla's API for programming GPU applications much more complex and rendered the maintenance of the framework's source code harder.

The solution proposed in this section adds to *GPUField* the support for contiguous memory while maintaining the pitched memory option in the same class without any code duplication. Instead, data structures and operations that have been used for pitched memory were adjusted to support contiguous memory as well. From the user's perspective, selecting contiguous or pitched memory can be done by simply setting the boolean constructor's parameter *usePitchedMem* to *false* or *true*, respectively.

GPUField works as a wrapper for CUDA *cudaPitchedPtr*, which holds a pointer to a segment on the GPU memory, where the domain data is stored. Besides, *cudaPitchedPtr* also holds memory metadata, i.e. information about the structure of the allocated memory itself:

- *pitch*: Stride of the x dimension in bytes.
- *xsize*: Size of the x dimension in bytes.
- *ysize*: Size of the y dimension in elements.

Although *GPUField* represents a 4D structure with (x, y, z, f) coordinates, pitched memory is allocated with *cudaMalloc3D()* in a 3D fashion and all *cudaPitchedPtr*'s metadata is set by the *cudaMalloc3D()* function. As described in Section 4.2.3, a memory mapping for fitting the *GPUField*'s 4D structure into the allocated 3D memory object is required. Memory alignment is always held in *xsize* dimension. Therefore, in case *xsize* matches the memory alignment of the device, usually 512 bytes, the value of *pitch* equals the value of *xsize*. Otherwise, *cudaMalloc3D()* pads the allocated memory to ensure memory alignment and sets *pitch* accordingly [25].

The implemented contiguous memory for *GPUField* reuses the *cudaPitchedPtr* structure for holding contiguous memory as well. As soon as a segment of contiguous memory is allocated using *cudaMalloc()*, *cudaPitchedPtr*'s data pointer is set up to hold it. Since, waLBerla and CUDA Runtime API rely on *cudaPitchedPtr*'s metadata for all operations on pitched memory, the metadata is set up using *make_cudaPitchedPtr()* so that *pitch* always equals the given *xsize* even if it does not meet the memory alignment. Thus, all operations that rely on *cudaPitchedPtr*'s metadata to handle *GPUField*'s pitched memory can also handle contiguous memory.

Listing 4.1 depicts details of how *cudaPitchedPtr* is set up and how memory type is selected in *GPUField*'s constructor. *pitchedPtr_* is the *cudaPitchedPtr* attribute of *GPUField*. *usePitchedMem_* is the boolean *GPUField*'s attribute that represents the memory type, where *true* means pitched memory and *false* means contiguous memory. The local variable *extent* was previously set up with the mapped dimensions field.

Listing 4.1: Memory type selection in *GPUField*'s constructor.

```
1  if (usePitchedMem_)  // use pitched memory
2  {
3     cudaMalloc3D(&pitchedPtr_, extent);
4  }
5  else  // use contiguous memory
6  {
7     pitchedPtr_ = make_cudaPitchedPtr(NULL, extent.width, extent.width,
8           extent.height);
9     cudaMalloc(&pitchedPtr_.ptr,
10          extent.width * extent.height * extent.depth);
11 }
```

In addition to those changes, the *GPUField*'s method *isPitchedMem()* was created in order to allow runtime check for which memory type is in use. Some framework functions related to *GPUField* required minor changes in order to support memory type selection, e.g. *addGPUFieldToStorage()*, *createGPUField()* and *createGPUFieldFromCPUField()*. The pair of *fieldCpy()* functions used to copy data between *Field* and *GPUField* required bug fixes in order support contiguous memory and prevent undefined behavior in some situation when using pitched memory.

4.3.3 Field Indexing

The pair of classes *FieldIndexing* and *FieldAccessor*, as described in Section 2.4.2, are provided by the base version of waLBerla for, respectively, indexing and accessing *GPUField*'s elements. Although they showed very good performance, the mapping for *GPUField*'s elements to CUDA threads is rigid and does not perform very well in some scenarios. For this reason, a new pair of field index classes was developed: *FieldIndexing3D* and *FieldAccessor3D*. They allows for choosing thread block sizes so that different arrangement of CUDA threads and blocks can be used and optimal strategies to favor memory access coalescence can be employed [30].

FieldIndexing3D conveniently maps (x, y, z) spatial coordinates of a *GPUField* to CUDA's block and thread coordinates through methods such as *xyz()* and *sliceXYZ()*. These methods follow the same interface and functionalities as provided by *FieldIndexing* and more details about them can be found in Section 2.4.2. In order to provide flexible thread block sizes, the *setPreferredBlockDim(x,y,z)* method can be used for defining the x, y and z the dimensions of every thread block. Then, mapping is done so that given a *GPUField* region, a sufficient number of thread blocks for handling one cell per thread is allocated in each of x, y and z directions.

Similarly to *FieldAccessor* and *FieldIndexing* relationship, *FieldAccessor3D* allows CUDA kernels to access *GPUField*'s elements so that every element is conveniently mapped to a CUDA thread as defined by the *FieldIndexing3D* class. Although internal details in *FieldAccessor3D* differ greatly from *FieldAccessor* due to the different memory indexings, they follow the same interface and functionalities.

In initial tests, *FieldIndexing3D* showed to improve kernel performance considerably in comparison with *FieldIndexing*. The optimal thread block size was $(32, 2, 2)$ in most cases. However, as kernels were being increasingly optimized for this work, *FieldIndexing* eventually surpassed in performance in most of the tested scenarios. Nevertheless, *FieldIndexing3D* was kept in the framework for its flexibility and for the fact that field indexing performance showed to be very sensitive to kernel implementation. Thus, it is expected that *FieldIndexing3D* can outperform *FieldIndexing* in other scenarios.

4.4 Lid-Driven Cavity Application

An application implementing a 3D lid-driven cavity (LDC) simulation, as introduced in Section 2.3, was developed in order to verify the validation, performance and memory efficiency of the implemented solutions for waLBerla.

The developed lid-driven cavity application is highly configurable and allows for setting up a variety of scenarios through the waLBerla's Python script extension [1]. For instance, parameters such as the domain size, block decomposition scheme, domain periodicity, number of simulation iterations, communication scheme, GPU memory scheme, field indexing type and boundary handling type can be configured.

4.4.1 Lid-Driven Cavity Kernels

This section presents the LBM kernels implemented in CUDA for the lid-driven cavity simulation. Although most of the kernels' code is generic enough to be used for a variety of LBM applications, some parts were strictly optimized for a LDC simulation. Implementing optimized kernels is very important as scalability results require optimized kernels to be relevant [10].

As introduced in Section 2.2, there are basically three major routines associated to the LBM: streaming, collision and boundary handling. Streaming and collision operate on fluid cells of the domain while boundary handling operates on boundary cells.

An LDC simulation requires different treatment for fluid, top boundary and the remaining boundary cells. Thus, the application needs a mechanism to define and efficiently identify different types of cells. To this end, waLBerla supports a mechanism for creating a flag field whose every cell is an 8-bit bitmap. In the implemented solution, as depicted in Figure 4.3, there is a flag for fluid cells and there are two distinct flags for boundary cells: *noSlip* and *simpleUBB*. The former represents the cavity's side and bottom boundary while the latter represents the top boundary, which implements the top velocity. In addition, there is the *nearBoundary* flag, which is present in every fluid cell that is a neighbor of a boundary cells.

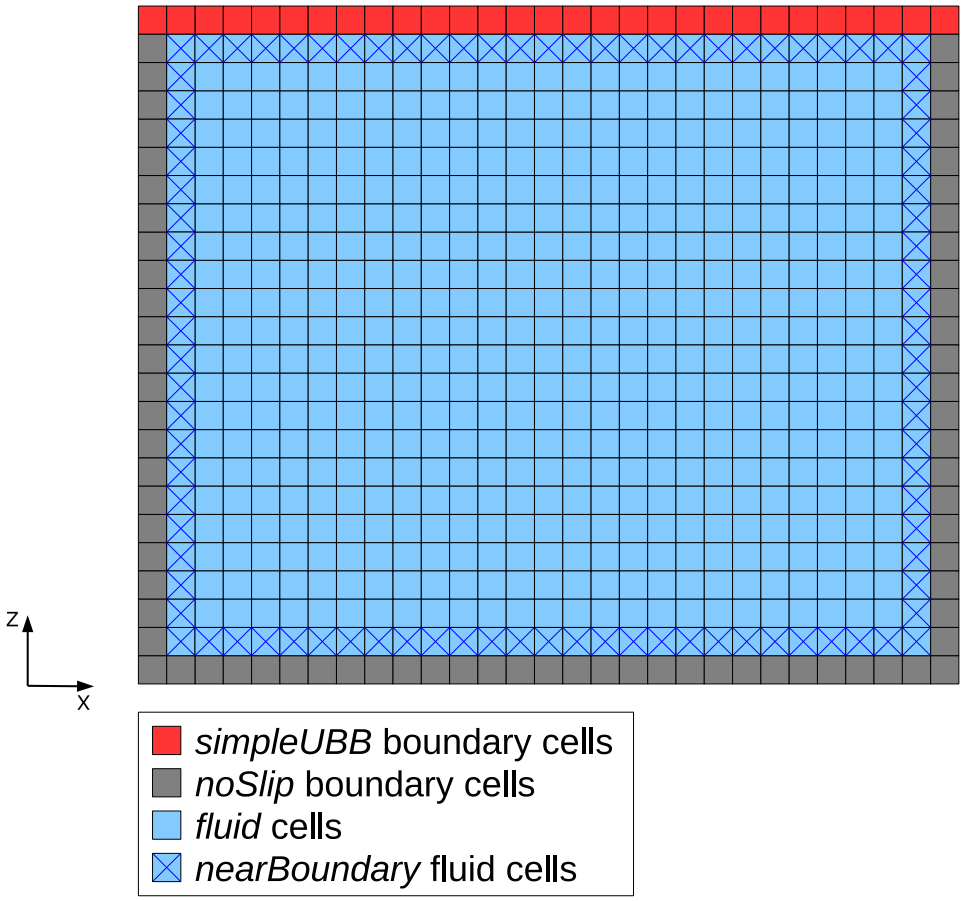


Figure 4.3: Cell types in the domain of the LDC simulation as defined by the flag field.

The streaming and collision operations for handling fluid are implemented in a single kernel, *streamAndCollideKernel()*, in order to make some code optimizations possible. It was introduced by Sepka [42] and as it showed in preliminary tests to be highly optimized, no major change was required.

New boundary handling kernels were created. There is a boundary handling kernel for each of the six faces of the box-shaped LDC domain. Splitting boundary handling into six kernels allows for basically two optimizations:

1. Neighboring optimization: As long as each kernel handles only the boundary on one of the six faces of the domain, the kernel only needs to consider the neighboring cells towards the directions the boundary cells are located in relation to the near-boundary cells being treated, which reduces the number of iterations on neighboring cells from 18 to only 5.

2. CUDA thread optimization: Instead of allocating a CUDA thread to calculate boundary handling for each domain cell, only a reduced number of threads is allocated for handling the near-boundary cells, so that most of the fluid cells are completely ignored.

All boundary handling kernels, however, share the same basic algorithm which is coded as the device function described in Listing 4.2. The assignment operation, represented as *CELL_ASSIGNMENT()*, has a implementation for *noSlip* boundary handling and another one for *simpleUBB*. The key optimization of that function is the *dir* parameter that represents the direction of the neighboring cells to be iterated. Declaring the required constant vectors as

local variables allows the compiler to store them in register memory, which showed to improve performance. Note that the vector *inv* maps directions to the corresponding inverse directions. The vectors *cx*, *cy* and *cz* map directions to the relative positions of neighbor cells towards the given directions, respectively, in x, y and z coordinate offsets. The function is implemented as a template so that different field indexings can be selected at compile time.

Listing 4.2: Boundary handling.

```
template<template <typename> class FieldAccessor_T>
__device__ __forceinline__
void boundaryHandling(FieldAccessor_T<double> & src,
      FieldAccessor_T<flag_t> & flag,
      flag_t nearBoundaryFlag, flag_t boundaryMask,
      const Direction dir[], uint_t dirSize)
{
   src.set(blockIdx, threadIdx);
   flag.set(blockIdx, threadIdx);
   if(src.isValidPosition() && (flag.get() & nearBoundaryFlag))
   {
      const Direction inv[] = {C,S,N,E,W,B,T,SE,SW,NE,NW,BS,BN,BE,BW,TS,TN,
            TE,TW};
      const int cx[] = {0,0,0,-1,1,0,0,-1,1,-1,1,0,0,-1,1,0,0,-1,1};
      const int cy[] = {0,1,-1,0,0,0,0,1,1,-1,-1,1,-1,0,0,1,-1,0,0};
      const int cz[] = {0,0,0,0,0,1,-1,0,0,0,0,1,1,1,1,-1,-1,-1,-1};
      for(int i = 0; i < dirSize; ++i)
      {
         Direction d = dir[i];
         int x = cx[d];
         int y = cy[d];
         int z = cz[d];
         if(flag.getNeighbor(x, y, z) & boundaryMask)
         {
            CELL_ASSIGNMENT();
         }
      }
   }
}
```

The assignment operation for *noSlip*, can be simply implemented as in Listing 4.3.

Listing 4.3: *noSlip* cell assignment.

```
src.getNeighbor(x, y, z, inv[d]) = src.get(d);
```

The assignment operation for *simpleUBB*, as shown in Listing 4.4, requires a double value *topVelocity* representing the top velocity of the LDC simulation and a vector of weights w from the equilibrium equation of the LBM.

Listing 4.4: *simpleUBB* cell assignment.

```
src.getNeighbor(x, y, z, inv[d]) = src.get(d)
      - (6.0 * computeRho(src) * w[d] * (cx[d] * topVelocity));
```

All required boundary handling kernels can be defined in terms of *boundaryHandling()* by properly passing its required directions to the *dir* parameter. For instance, the *noSlip* boundary handling for the east face of the domain can be defined as in Listing 4.5. A generic *noSlip* kernel that can be applied for other LBM simulations, can also be defined by passing all stencil direction to *dir*.

Listing 4.5: *noSlip* kernel for the east boundary.

```
template < template <typename> class FieldAccessor_T >
__global__
void noSlipEastKernel(FieldAccessor_T<double> src,
FieldAccessor_T<flag_t> flag, flag_t nearBoundaryFlag, flag_t noSlipMask)
{
   const Direction dir[] = {E,SE,NE,BE,TE};
   boundaryHandling(src, flag, nearBoundaryFlag, noSlipMask, dir, 5);
}
```

The kernels are launched from the host side, where a sweep called *StreamAndCollideSweep* launches the stream and collide kernel for fluid handling, and a sweep called *BoundaryHandlingSweep* launches the boundary handling kernels. In order to promote an efficient memory handling, the sweep *StreamAndCollideSweep* is derived from the *GPUSweepBase* class introduced in Section 4.3.1.

The *StreamAndCollideSweep* sweep has a *streamAndCollide()* method implementing the kernel launch, as depicted in Listing 4.6. That method first creates a instance of the template class *Kernel* initialized with *streamAndCollideKernel()*. The template argument *FieldAccessor* means that it is going to be used to access *GPUField*'s cells from inside the kernel. As a consequence, *FieldIndexing* must be used for indexing all *GPUField*s passed as arguments: the source field *src_*, the destination field *dst_* and the flag field *flag_*. The argument *fluidMask_* is a bit mask for the fluid flag. More details about the *Kernel* and *FieldIndexing* operations can be found in Section 2.4.2.

Listing 4.6: Stream and collide kernel launch.

```
void StreamAndCollideSweep::streamAndCollide()
{
   // create a kernel instance
   auto kernelStreamAndCollide = cuda::make_kernel(
         &streamAndCollideKernel<cuda::FieldAccessor>);

   // add parameters to the kernel
   kernelStreamAndCollide.addFieldIndexingParam(
         cuda::FieldIndexing<double>::xyz(*src_));
   kernelStreamAndCollide.addFieldIndexingParam(
         cuda::FieldIndexing<double>::xyz(*dst_));
   kernelStreamAndCollide.addFieldIndexingParam(
         cuda::FieldIndexing<double>::xyz(*flag_));
   kernelStreamAndCollide.addParam<flag_t>(fluidMask_);

   // launch the kernel
   kernelStreamAndCollide();
}
```

Similarly, *BoundaryHandlingSweep* has a *noSlipOptimized()* method for launching all the five no-slip boundary handling kernels and a *simpleUBBOptimized()* method for launching *simpleUBBKernel()*. Differently from *streamAndCollideKernel()* launch, the boundary handling kernels are launched so that only slices covering the field's near boundary cells are indexed by using *FieldIndexing::sliceXYZ()*. Therefore, only the number of CUDA threads that are sufficient to cover the boundary cells are executed.

4.4.2 Kernel Synchronization

Although the CUDA execution environment ensures that kernel launches performed into the same stream are executed in order, a kernel launch itself is asynchronous from the host perspective, i.e. when launching a kernel, the control may return to the host before the kernel has being completely processed by the device [24]. It allows for a basic parallelism between kernels running on GPU and host-side routines. However, some situations require synchronizing kernels to the host execution.

Sepka's solution [42], for example, always explicitly synchronizes kernels by calling *cudaDeviceSynchronize()* [25] at the end of every simulation loop. However, that approach showed to harm parallelism between host and device during a simulation due to the fact that the simulation loop, which is controlled by the host side, has to wait for the execution of all kernels launched in a simulation step before proceeding to the next step.

In the proposed lid-driven cavity application, it was verified that explicitly synchronizing kernels with the host is only necessary when measuring wall-clock time. However, in order to accurately take wall-clock measurements while minimizing the performance impact, two distinct types of measurements are considered for synchronization: overall wall-clock time and sub-step wall-clock time. In all other situations where synchronization is required, kernels are implicitly synchronized on demand by CUDA Runtime API functions such as *cudaMemCpy3D()*.

For the overall wall-clock time measurement, an explicit kernel synchronization is always made after the simulation loop for waiting all kernels to complete their execution before taking the measurement. As it is made outsize the simulation loop rather than inside, a considerable increase in performance was achieved in single block simulations with small or moderate sized domains. For instance, performance increase of about 5% and 1% were achieved for domains with 32^3 and 64^3 cells, respectively.

On the other hand, when measuring wall-clock time of simulation sub-steps such as communication and kernel execution, it is required to explicitly synchronize kernel at the end of every sub-step. As a consequence, performance is impaired in this case like in Sepka's solution. In order to keep the increase in performance when sub-step wall-clock time measurement is not required, sub-step synchronization was made configurable.

Chapter 5

Materials and Methods

This chapter describes hardware, software and methods employed to verify the validity, performance and memory usage of the proposed solutions. The validation experiments are designed to verify whether the proposed solutions produce the expected results. The performance experiments are designed to verify how the proposed solutions perform in different scenarios, their scalability and whether they have good performance compared to the state of the art and previous solutions available on waLBerla. Models of memory usage and the maximum domain size a give system can support are also presented and verified empirically. The lid-driven cavity simulation with double precision D3Q19 stencil presented in Section 2.3 is the use case employed in all experiments. The results of all proposed experiments is then presented in Chapter 6.

5.1 Hardware and Software

Two systems were used for validation and performance experiments:

1. **cluster-ib**:
   - GPU: NVIDIA Tesla K40m [23]
     - CUDA driver: 7.5
     - CUDA capability: 3.5
     - CUDA cores: 2880
     - Core clock: 745 MHz
     - Total memory: 11520 MB (12079136768 bytes)
     - Memory clock: 3.0 GHz
   - CPU: Intel® Xeon® CPU E5-4627 v2
     - Max clock rate: 3.30 GHz
     - Cores: 8
     - Sockets: 4
   - Nodes with GPU: 2
   - GPUs per node: 2
   - RAM per node: 256 GB
   - Bus: PCIe v3.0 x16
   - Network controller: Mellanox Technologies MT27500 Family [ConnectX-3]

- Compiler: g++ v4.9.2, nvcc v7.5.17
- MPI: Open MPI v1.10.2
- Location: DINF, Federal University of Paraná (UFPR) [32]

2. **achel**:

- GPU: NVIDIA Tesla C2075 [22]
  - CUDA driver: 7.5
  - CUDA capability: 2.0
  - CUDA cores: 448
  - Core clock: 1.15 GHz
  - Total memory: 5375 MB (5636554752 bytes)
  - Memory clock: 1.50 GHz
- CPU: Intel® Xeon® CPU E5-4627 v2
  - Max clock rate: 3.30 GHz
  - Cores: 8
  - Sockets: 4
- Nodes with GPU: 1
- GPUs per node: 1
- RAM per node: 256 GB
- Bus: PCIe v3.0 x16
- Compiler: g++ v4.8.5, nvcc v7.5.17
- MPI: Open MPI v1.10.2
- Location: DINF, Federal University of Paraná

The GPU support to CUDA capability 2.x is the minimum required to compile and use most of the proposed solutions, however CUDA capability 3.0 or newer is recommended. CUDA 7 or newer is required to compile the proposed solutions with waLBerla as long as C++11 template features where used to make LDC kernels generic in terms of the *FieldIndexing* used.

All solutions were developed based on the waLBerla commit 840c7bf1-2016-03-29 from branch *topic/cuda*. All experiments are built with waLBerla's *Release* build type which implies that compiler option *-O3* is used. The waLBerla options *WALBERLA_BUILD_TUTORIALS*, *WALBERLA_BUILD_WITH_CUDA* and *WALBERLA_BUILD_WITH_PYTHON* are set to *ON* and *CUDA_NVCC_FLAGS* is set to *-std=c++11 -arch=sm_35* on *cluster-ib* and *-std=c++11 -arch=sm_20* on *achel*.

The software ParaView v4.0.1 [17] is used to extract velocity values and generate streamlines from the validation experiment results. The online tool WebPlotDigitizer v3.9 [39] is used to extract numerical data from reference paper plots for the sake of comparison.

5.2 Validation

The CUDA implementation of the lid-driven cavity simulation presented in Chapter 4 is executed with varied parameters so that every proposed solution can be verified. The results

of the simulations are then compared to the results of the previously validated lid-driven cavity simulation bundled with waLBerla, which is used as a reference implementation.

As proposed in Rinaldi et al. [38], the profiles of the velocity components of the midplane perpendicular to the top velocity ($y = 0.5$ midplane) are used to compare the results of tested application with the results of the reference application. Streamlines are also analyzed in order to verify the presence of a main vortex rotating around an axis in y direction.

It is required a set of test cases in order to verify all proposed solutions and features. Most of the test cases validate many solutions and all of them validate *StreamAndCollide* kernels, *FieldIndexing3D* template class implementation and *GPUFlagField* changes.

All test cases (including the reference test case) consist of 10000 iterations of the lid-driven cavity simulation using D3Q19 stencil, double precision float points, *fzyx* memory layout, total domain size of $128 \times 128 \times 128$ cells, top velocity vector $v = (0.1, 0, 0)$ and $\omega = 1.4$. Except where expressed otherwise, the validation test cases use contiguous memory, generic boundary handling, *FieldIndexing3D* for indexing and *GPUPackInfo* for communication. All simulations are were executed on *cluster-ib*. Following are the description of every validation test cases:

1. **Domain decomposition**: This test case validates the implementation of *GPUPackInfo* intra-GPU communication (i.e. local communication between blocks running on the same GPU), the memory management of destination fields implemented in *GPUSweepBase* template base class, the contiguous memory support for *GPUField* and the *FieldCopy()* functions bug fixes. The scenario consists of $4 \times 4 \times 4$ blocks of $32 \times 32 \times 32$ cells running on a single process and on a single device of a single node. As shown in Figure 5.1, it tests every possible domain decomposition direction including fluid and boundary.

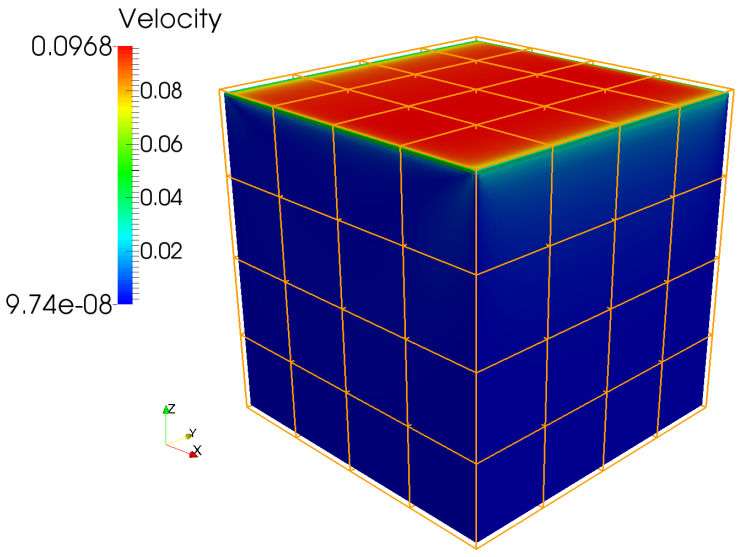
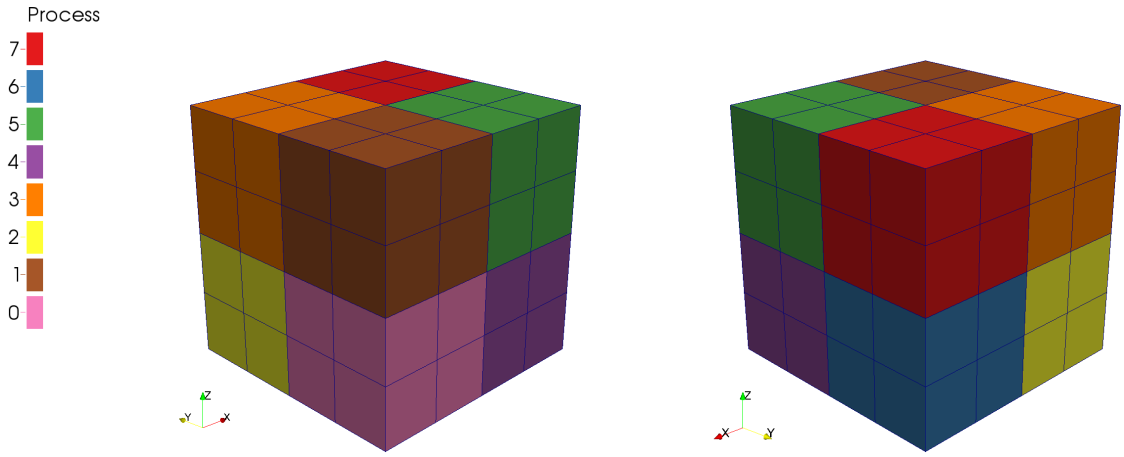


Figure 5.1: Lid-driven cavity simulation with $4 \times 4 \times 4$ blocks of $32 \times 32 \times 32$ cells running on a single process. The block decomposition of the domain is shown as lines in orange. Only the fluid part of the domain is showed.

2. **Multiple devices per node**: This test case validates the inter-GPU communication in a single node. The intra-node buffer handling and the MPI communication are validated, but no InfiniBand communication is tested. Validation includes the intra-node use of

GPUPackInfo pack and unpack functions, the *RecvBuffer::skip()* method implementation, the *SendBuffer::forward()* method implementation and the *GPUCopy()* function enhancements. The scenario consists of $4 \times 4 \times 4$ blocks of $32 \times 32 \times 32$ cells running on $2 \times 2 \times 2$ processes, where blocks are processed on 2 devices of a single node. Processes are distributed as shown in Figure 5.2, where blocks running on processes with even ranks are processed on the device 0 and blocks running on processes with odd ranks are processed on the device 1.



(a) (b)

Figure 5.2: Process distribution of the LDC simulation with $4 \times 4 \times 4$ blocks of $32 \times 32 \times 32$ running on $2 \times 2 \times 2$ processes. Every process is represented using a unique color and its rank is shown in the legend. The block decomposition in every process is shown as lines in back. (a) is the view from the $(0,0,1)$ corner and (b) is a view from the $(1,1,1)$ corner.

3. **Multiple nodes**: This test case validates the inter-node communication, including buffer handling, MPI, InfiniBand and the inter-node usage of *GPUPackInfo* pack/unpack functions. The scenario consists of $4 \times 4 \times 4$ blocks of $32 \times 32 \times 32$ cells running on $2 \times 2 \times 2$ processes, where blocks are processed on 2 nodes with 2 devices per node. Processes are distributed as shown in Figure 5.2, where blocks running on processes with ranks 0, 1, 2 and 3 are on the node 0 and with ranks 5, 6, 7 and 8 are on the node 1. Again, blocks running on processes with even ranks are processed on the device 0 of each node and with odd ranks are processed on the device 1.

4. **Pitched memory**: This test case validates the use of pitched memory rather than contiguous memory on the *GPUField*. Except for the contiguous memory, all simulation parameters for this test case are identical to the Item 1 parameters.

5. **Optimized boundary handling**: This test case validates all optimized boundary handling kernels. Except for the boundary handling, all simulation parameters for this test case are identical to the Item 1 parameters.

When comparing, the overall results of the reference and the target tests must be very similar. However, due to the nature of float point arithmetics, the different optimization techniques used and the different hardware architecture involved, minor differences among the results may occur.

5.3 Performance

This section presents experiments to verify the performance of the proposed solutions. The experiments are designed to determine how the proposed solutions perform and scale for varied domain sizes and decompositions. A kernel comparison with a state-of-the-art solution and a communication performance comparison with the previously available solutions for waLBerla are also presented.

The lid-driven cavity simulation with double precision D3Q19 stencil presented in Section 2.3 is employed as a use case in all experiments. In all cases, contiguous memory is used, memory layout is *fzyx*, *FieldIndexing* is used for indexing *GPUField*s within CUDA kernels and the device ECC (Error-Correcting Code) is turned on.

Performance is measured based on the wall-clock time of the simulation loop, ignoring initializations and finalizations. In order to mitigate warm-up interference in performance verification, 200 simulation iterations are executed and every scenario is repeated 10 times, where only the one with the shortest wall-clock time is regarded. Some experiments require taking measurements of simulation sub-steps, such as communication and kernel execution. In order to measure sub-step wall-clock time accurately, it is required synchronizing all launched CUDA kernels at the end of every sub-step, which may harm performance for small and moderate sized domain blocks, as described in Section 4.4.2.

5.3.1 LBM Kernel Performance

This section has the objective of verifying the LBM kernel performance. This is important for quality assurance of the experiments described in the next sections since scalability results require an optimized kernel to be relevant. Unoptimized kernel implementation may show good scalability despite taking a longer total wall-clock time [10]. A performance comparison with the state of the art was made in order to verify the kernel's performance of the proposed solution.

The LDC implemented in CUDA introduced by Habich et al. [13] was chosen as a reference implementation for comparison since it can be reproduced on one of the available systems. Even though none of the GPU cards used by the reference work matches exactly one of the available GPU cards, the NVIDIA Tesla C2075 equipped on *achel* has equivalent specifications to the NVIDIA Tesla C2070 used by the reference work, including the number of cores, processor clock, memory size and memory clock [22, 21].

The simulation were conducted on *achel* using a single block decomposition and a domain size of 200^3 cells. The resulting performance in MFLUPS (Million Fluid Lattice Updates per Second) is compared with the equivalent scenario of the reference paper with ECC enabled.

5.3.2 Communication Performance

The objective of this section is to verify the communication performance of the proposed GPU communication solution introduced in Section 4.2 compared to the previously available solutions for waLBerla.

Regarding domains with multiple uniform decomposition blocks in all three directions, even though the kernel performance is expected to be very similar among the blocks, the communication sub-step performance may vary considerably as some of the blocks have more neighbors than others and thus more data to communicate than others. For a non-periodic domain

and the D3Q19 stencil, blocks on the domain boundaries may have from 6 up to 13 neighbors while internal blocks have always 18 neighbors to communicate.

As long as the overall communication performance depends on the blocks with greater communication time, a convenient block to have its communication performance determined is the internal block. A $3 \times 3 \times 3$ domain decomposition is the minimal required to obtain an internal block in a non-periodic domain with the D3Q19 stencil, however none of the available systems described in Section 5.1 is equipped with 27 GPUs. In order to mitigate this limitation and approximate an internal node for the communication overhead experiments, a domain with period in all 3 directions was used so that every node has exactly 18 neighbors to communicate.

The proposed experiment consists of running the lid-driven cavity simulation on *cluster-ib* using the proposed GPU communication solution, *GPUPackInfo*, and the previously available solutions for waLBerla, *UniformDirectScheme* and *GhostLayerField* copy, in order to compare their performance in MFLUPS. Regarding the considerations about the available system limitations, a fully periodic domain is used. It is decomposed into 4 cubic blocks ($B_x = 1$, $B_y = 2$ and $B_z = 2$) running on 2 nodes with 2 devices each so that among the 18 communications per block, 16 are inter-GPU and only 2 are intra-GPU. Among the inter-GPU communications, 10 are inter-node and 6 are intra-node. The simulation was repeated for cubic block sizes of 32, 64, 128 and 256 cells.

5.3.3 Communication Overhead

The objective of this section is to determine the communication overhead of the proposed GPU communication solution introduced in Section 4.2.

In order to determine the communication overhead, it is also required to obtain an accurate wall-clock time measurement of the communication sub-step of the simulation, which is achieved by synchronizing all launched CUDA kernels at the end of every sub-step.

The proposed experiments consist of running the lid-driven cavity simulation on *cluster-ib* using the proposed GPU communication solution, *GPUPackInfo*. The communication overhead is determined by measuring and analyzing the sub-step and total wall-clock times. The considerations about the varied communication performance related to block position and the system limitations described in Section 5.3.2 are regarded, thus the domain is fully periodic and is decomposed into 4 cubic blocks ($B_x = 1$, $B_y = 2$ and $B_z = 2$) running on 2 nodes with 2 devices each. The simulation was repeated for cubic block sizes of 32, 64, 128 and 256 cells.

5.3.4 Communication Direction Imbalance

The objective of this section is to demonstrate that communication has different performance depending on the communication direction and how it can impact overall performance depending on the way a domain is decomposed. An experiment was performed in order to determine the communication wall-clock time for each of the directions with higher communication overhead, x (east/west), y (north/south) and z (top/bottom).

As described in Section 4.2, an essential part of the communication using *GPUPackInfo* involves copying axis-aligned slices of the fields. If communication happens within the same process, copies are made directly from the source to the destination field, otherwise copies are made from the source field to a local buffer, the data is transmitted to the destination block via MPI and then copied from a local buffer on the destination process to the destination field. In any case, the source of a copy is always inside the domain of a block while the destination is in the block's ghost layer.

The slice geometry depends on the stencil and the communication direction, i.e. the direction the destination neighbor field lies in relation to the source field. For the D3Q19 stencil, a given field can have up to 18 neighbor fields to communicate. Up to 6 slices have dimensions of $n_i \times n_j \times g$ and up to 12 slices have dimensions of $n_i \times g$, where n_i and n_j are two different dimensions of the field, and g is the thickness of the ghost layer. As long as the ghost layer thickness is typically $g = 1$ in LBM case [1], the slices have a thickness of 1 while usually being much larger in the other dimensions.

Regarding the discrepancy in slice dimensions, it is expected that slice copies perform differently depending on the orientation of the smaller dimension of the slice due to memory coalescence. Note that, for contiguous memory and the *fzyx* memory layout, a field's element with (x, y, z, f) coordinates can be addressed in device memory by calculating an index i_p in relation to the data pointer as following:

$$i_p = f n'_z n'_y n'_x s + z n'_y n'_x s + y n'_x s + x s \tag{5.1}$$

where s is the size in bytes of an element and n'_x, n'_y and n'_z are, respectively, the number of field cells in x, y and z directions, including ghost layer cells.

Therefore, according to Equation 5.1, any element in an x-thinner slice, i.e. a slice thinner in the x direction, is $n'_x s$ bytes apart from an adjacent element in the y direction and $n'_y n'_x s$ bytes apart from an adjacent element in the z direction. As a consequence, a copy function has no way to coalesce access to elements of an x-thinner slice for practical field sizes.

On the other hand, in y or z-thinner slices, the copy function has coalescent access to the elements since an adjacent element in the x direction is also adjacent in the device memory, as depicted in Figure 5.3. Thus, communication in the x direction is expected to have poorer performance than in any other direction. A similar issue was also reported by Wang et al. [48].

The proposed experiment to verify the communication direction imbalance consists of running the lid-driven cavity simulation with the domain decomposed into 2 blocks running on separate nodes of *cluster-ib*, doing so for each one of the 3 decomposition directions, x, y and z. In order to prevent interference of any performance impact that might be introduced by directional changes in block shape, every block is cubic so that the total domain is twice as long towards the communication direction as the other directions. The experiment was repeated with cubic block size varying in $n_\alpha \in \{32, 64, 128, 256\}$ cells. Communication, kernel and total wall-clock times of each scenario were measured.

5.3.5 Scalability

This section has the objective of verifying the scalability of the proposed solutions. Two scalability metrics are determined, weak and strong scaling. A theoretical upper limit for both scalability metrics is also defined.

For weak scaling, block size was fixed to 320^3 cells and the simulation was run with 1, 2 and 4 blocks, with exactly 1 block per GPU. For strong scaling, the overall domain size was fixed to 320^3 cells and the simulation was run decomposing the domain into 1, 2 and 4 blocks, with exactly 1 block per GPU. Performance was determined in MFLUPS in both scalings.

The theoretical performance upper limit for both scalability metrics regards the ideal scenario where there is no overhead of including more GPUs to process the simulation. It can be determined by multiplying the performance of the single-GPU scenario by the number of GPUs being used.

Both scalability experiments were performed on *cluster-ib*. Due to the restriction of 4 GPUs imposed by the test system, it was not possible to test decomposition in all 3 directions

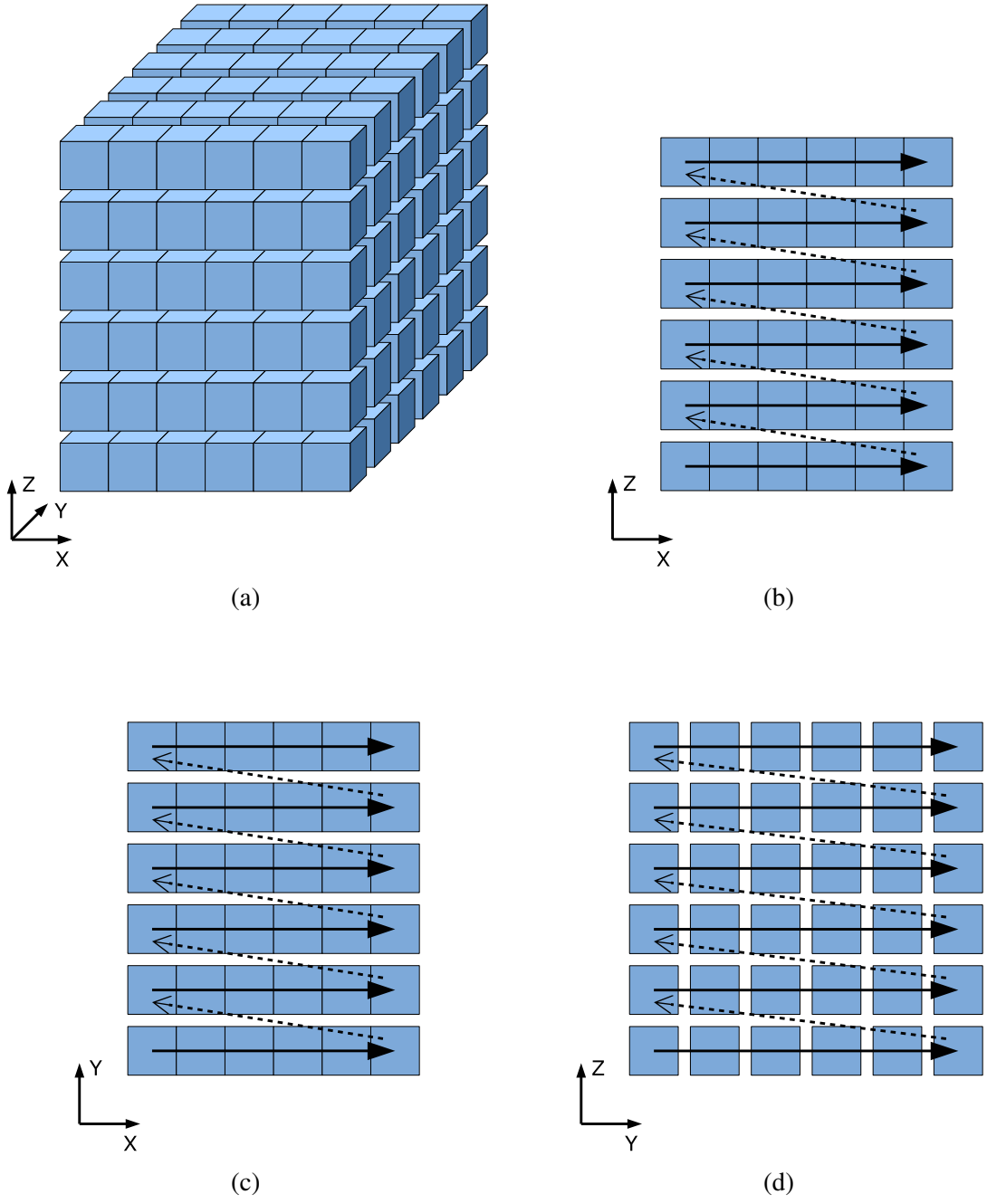


Figure 5.3: Memory coalescence of a domain block on GPU for *fzyx* memory layout. In (a), each row in x direction represents PDF elements that are adjacent in memory. Coalescent memory access can be achieved when CUDA threads within the same warp access adjacent elements withing a row [24]. Thus, coalescent access can be achieved in y planes (b) and z planes (c), but not in x planes (d).

in a single scenario since it would require at least 8 GPUs. Regarding that the direction of the communication can affect performance as described in Section 5.3.4, two different scenarios for each scaling were tested in order to cover all decomposition directions:

1. ***xz* decomposition**: The first decomposition is made in x direction and the second is made in the z direction.

2. ***yz* decomposition**: The first decomposition is made in y direction and the second is made in the z direction.

5.4 Memory Usage

This section presents models and experiments to verify the memory usage of the proposed solutions. The memory efficiency is verified through a model and an experiment designed to determine how the GPU memory usage of the proposed solutions behaves for different domain decompositions. A model to determine the maximum domain size a system can support for given parameters is also introduced and verified empirically.

The lid-driven cavity simulation with double precision D3Q19 stencil presented in Section 2.3 is employed as a use case in all experiments. In all cases, contiguous memory is used and memory layout is *fzyx*.

5.4.1 Memory Efficiency

This section has the objective of demonstrating how *GPUSweepBase* can provide an efficient use of the GPU memory by using multiple blocks per node. A model representing the GPU memory usage and how it behaves with different domain decompositions is proposed. An experiment is also proposed in order to verify empirically how close to the memory usage model the application performs.

As described in Section 4.3.1, *GPUSweepBase* implements a convenient sweep base class to manage fields in an efficient way. For uniformly decomposed domains, it allows allocating a destination domain using only $1/B$ of the allocated memory for the source domain, where B is the total number of blocks decomposing the domain.

The following model allows to calculate the total amount M in bytes of GPU global memory allocated by the *GPUField* instances in order to simulate the lid-driven cavity, which is given by:

$$M = M_s + M_d + M_f \tag{5.2}$$

where M_s, M_d and M_f are respectively the sizes in bytes allocated in GPU global memory by source, destination and flag fields.

The total number of cells in the domain field including ghost layers is represented by N' and is calculated as following:

$$N' = (N_x + 2gB_x) \times (N_y + 2gB_y) \times (N_z + 2gB_z) \tag{5.3}$$

where N_x, N_y and N_z are respectively the number of domain cells in x, y and z directions, B_x, B_y and B_z are respectively the number of decomposition blocks in x, y and z directions and g is the thickness of the ghost layer.

Given the Equation 5.3 and M_c the total size of a cell in bytes, M_s, M_d and M_f are calculated as following regarding contiguous memory and uniform decomposition:

$$M_s = M_c N' \qquad M_d = M_c \frac{N'}{B} \qquad M_f = N' \tag{5.4}$$

By substituting the Equations 5.4 in Equation 5.2, M is given by the following equation:

$$M = N' \left(M_c + \frac{M_c}{B} + 1 \right) \tag{5.5}$$

Note in Equations 5.5 that to some extent, the more blocks the domain is decomposed the less GPU memory is used by the application. However, as shown in Equation 5.3, increasing the number of blocks also increases the total number of cells when ghost layers are taken into consideration. Therefore, it is expected that decomposing the domain to some extent improves the memory efficiency. On the other hand, creating too many decomposition blocks would decrease the memory efficiency once the optimum number of blocks are achieved.

In order to verify empirically how the GPU memory usage behaves for different domain decompositions and how close to the memory usage model the application performs, the proposed experiment consists of running the simulation with a fixed cubic domain size $N = 256^3$ and different cubic block decompositions $B \in \{1, 2^3, 4^3, 8^3\}$. A graph comparing the following 3 values is presented:

1. The estimated total amount M of allocated GPU memory as given by Equation 5.5.
2. The total GPU memory allocated by all *GPUField* instances.
3. The GPU memory footprint of the lid-driven cavity application measured with the function *cudaMemGetInfo()* provided by the CUDA Runtime API.

5.4.2 Maximum Domain Size

This section has the objective of exploring the upper limit of the domain size that the lid-driven cavity application can simulate for a given GPU with total memory size M_{gpu}. Regarding a cubic domain and a cubic decomposition, a model to determine the maximum domain size and the optimal decomposition in terms of memory usage is presented, as well as an experiment is defined in order to verify the model empirically.

As shown in Equation 5.5, the memory usage depends not only on the domain size but also on the block decomposition. For cubic domain size and decomposition, we have from Equation 5.3 that $N' = (N_\alpha + 2gB_\alpha)^3$, where the domain size in any axis is given by $N_\alpha = N_x = N_y = N_z$ and the number of blocks in any axis is given by $B_\alpha = B_x = B_y = B_z$. Regarding that the total number of blocks is given by $B = B_\alpha^3$, by isolating N' from Equation 5.5 and replacing it by $(N_\alpha + 2gB_\alpha)^3$, we have the following equation to determine the domain size per axis:

$$N_\alpha = \sqrt[3]{\frac{M}{M_c + \frac{M_c}{B_\alpha^3} + 1}} - 2gB_\alpha \tag{5.6}$$

Note that the optimal number of decomposition blocks per axis $B_\alpha > 0$ can be obtained by calculating the local maximum for $N_\alpha = f(B_\alpha)$ in Equation 5.6, given $M_c > 0$, $g > 0$ and $M = rM_{gpu}$, where $0 < r \leqslant 1$ is a ratio of the total GPU memory aimed to be allocated by *GPUField* instances. The $f(B_\alpha)$ local maximum for can be found through numerical methods such as the golden section search [37].

Note that for practical simulations, there are the requirements that N_α and B_α must be integer values and that N_α must be evenly divisible by B_α. In order to comply with those requirements, N_α and B_α can be calculated as described in Algorithm 2.

Algorithm 2 Maximum domain size and optimal decomposition calculation.

Output: $N\alpha$, $B\alpha$

1: Calculate $B\alpha$ on the $f(B_\alpha) = N_\alpha$ local maximum from Equation 5.6
2: $B\alpha \leftarrow round(B\alpha)$
3: Calculate $N\alpha$ from Equation 5.6 given $B\alpha$
4: $N\alpha \leftarrow floor(N\alpha/B\alpha) \times B\alpha$

An experiment was performed on a single GPU of the system *cluster-ib* in order to verify empirically the presented model with the proposed lid-cavity application. The memory size of a GPU of *cluster-ib* is 12079136768 bytes (approximately 11520 MB) and a ratio $r = 0.98$ of the memory is assumed, therefore we have $M = 11837554032.64$. Assuming the double precision D3Q19 stencil, we have $M_c = 19 \times 8 = 152$ and $g = 1$. Thus, from Algorithm 2 we have the maximum cubic domain $N_\alpha = 412$ and the optimal cubic decomposition $B_\alpha = 4$.

In order to define a maximum domain size curve for *cluster-ib*, besides the optimal decomposition, different maximum domain sizes were calculated and tested for values of $B_\alpha \in \{1,2,4,8\}$.

Chapter 6

Results and Discussion

This chapter presents and discuss the results of the experiments proposed in Chapter 5.

6.1 Validation Results

In this Section the results of all lid-driven cavity simulations proposed in Section 5.2 as validation experiments are presented and compared through velocity profile with the results of the reference implementation simulation like in Rinaldi et al. [38].

Figure 6.1 shows the velocity profiles of the simulation results along the cavity midplane $y = 0.5$ with the field dimensions normalized to the interval $[0, 1]$. The Vz velocity component is obtained from the midplane along the x direction and $z = 0.5$. The Vx velocity component is obtained along the z direction and $x = 0.5$. All validation simulations give the same profile and are represented as curves. The reference simulation results are represented as dots. It can be observed that the results of the validation simulation match the results of reference simulation.

Figure 6.2 shows the streamlines obtained from the *multiple nodes* validation test results. Figure 6.2(b) is a corner view of the field ($azimuth = -45°$, $elevation = 22.5°$) and Figure 6.2(a) is a side view. It was calculated using 400 points distributed in a radius of 50 cell units from the center point. A large vortex can be observed rotating around an axis in y direction. Identical features can be found in streamlines obtained from the reference simulation and from the remaining validation tests.

Regarding that all validation simulation produced velocity profiles that match the reference simulation results and the streamlines behaves as predicted, all proposed solutions are validated.

6.2 Performance Results

This section presents the results and analysis of the performance experiments introduced in Section 5.3.

6.2.1 LBM Kernel Performance Results

This section presents and analyzes the results of the kernel performance experiment described in Section 5.3.1.

The experiment conducted on *achel* showed that the proposed solution performance reaches 177 MFLUPS on a single block decomposition of 200^3 cells. The reference paper, on

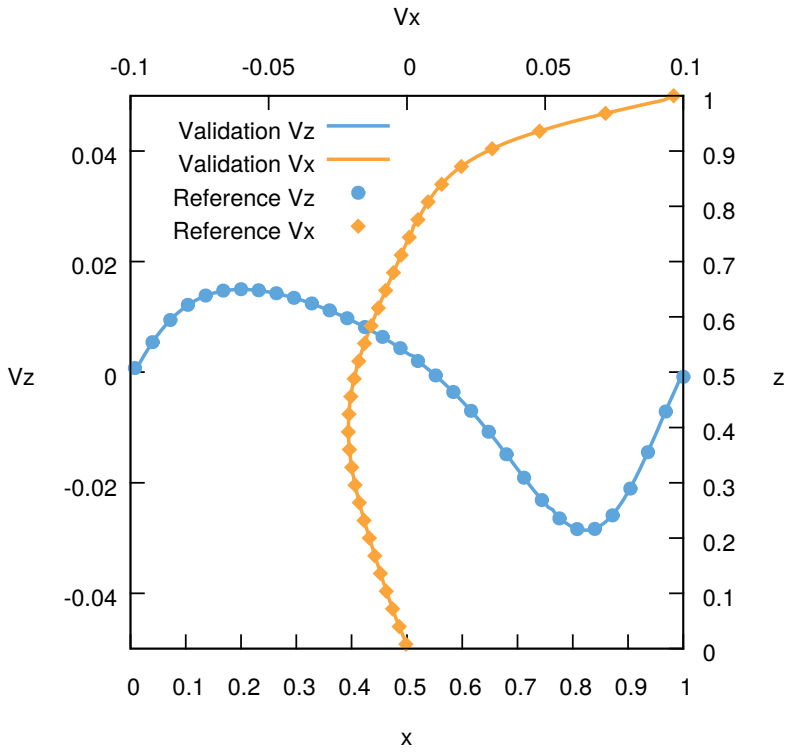
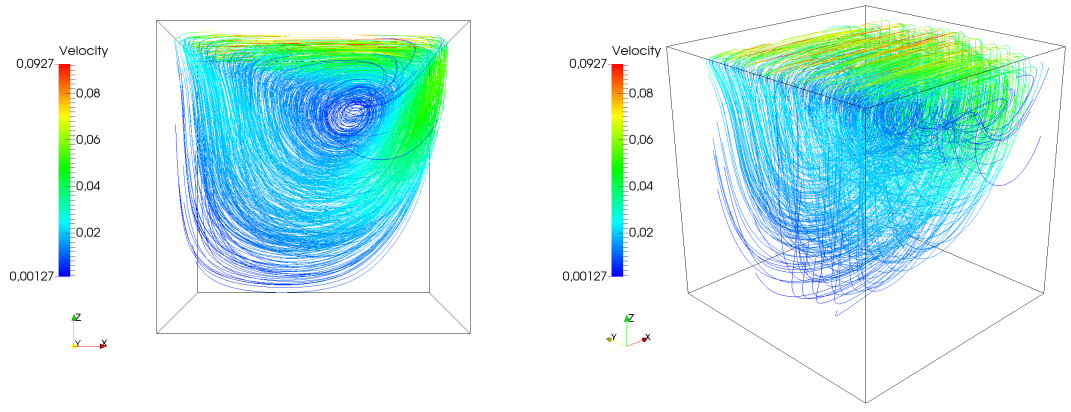


Figure 6.1: Profiles of the velocity components V_z and V_x in the field midplane $y = 0.5$. The curves are the validation simulation and the dots are the reference results.



(a) (b)

Figure 6.2: Streamlines obtained from the *multiple nodes* validation test results. (a) is the front view and (b) is a corner view with $azimuth = -45°$ and $elevation = 22.5°$.

the other hand, reported a performance of about 152 MFLUPS under the same conditions [13], value extracted from paper's Figure 3 using WebPlotDigitizer v3.9 [39].

As there is no communication in a single block decomposition, the presented performance results are good representations of the kernel performance of each solution. Therefore, the kernel of the proposed solution performs about 16% faster than the reference solution in the presented conditions, which is a very satisfactory result and allows to confidently proceed to scalability tests.

6.2.2 Communication Performance Results

This section presents and analyzes the results of the communication performance experiments described in Section 5.3.2.

As shown in Figure 6.3, the proposed communication solution, *GPUPackInfo*, performs much better than the previously communication solutions available for waLBerla. For a block size of 256^3 cells, *GPUPackInfo* achieved 583 MFLUPS while *GhostLayerField* copy and *UniformDirectScheme* only reached 23 and 1.5 MFLUPS, respectively. Note that even though *GhostLayerField* copy performs slightly better than *GPUPackInfo* for small block sizes such as 32^3, the former does not perform well as larger blocks are used while the performance of the latter increases greatly as lager blocks are used. On the other hand, even though an increasing performance is noticeable for *UniformDirectScheme*, it performs very badly whatever block size is used.

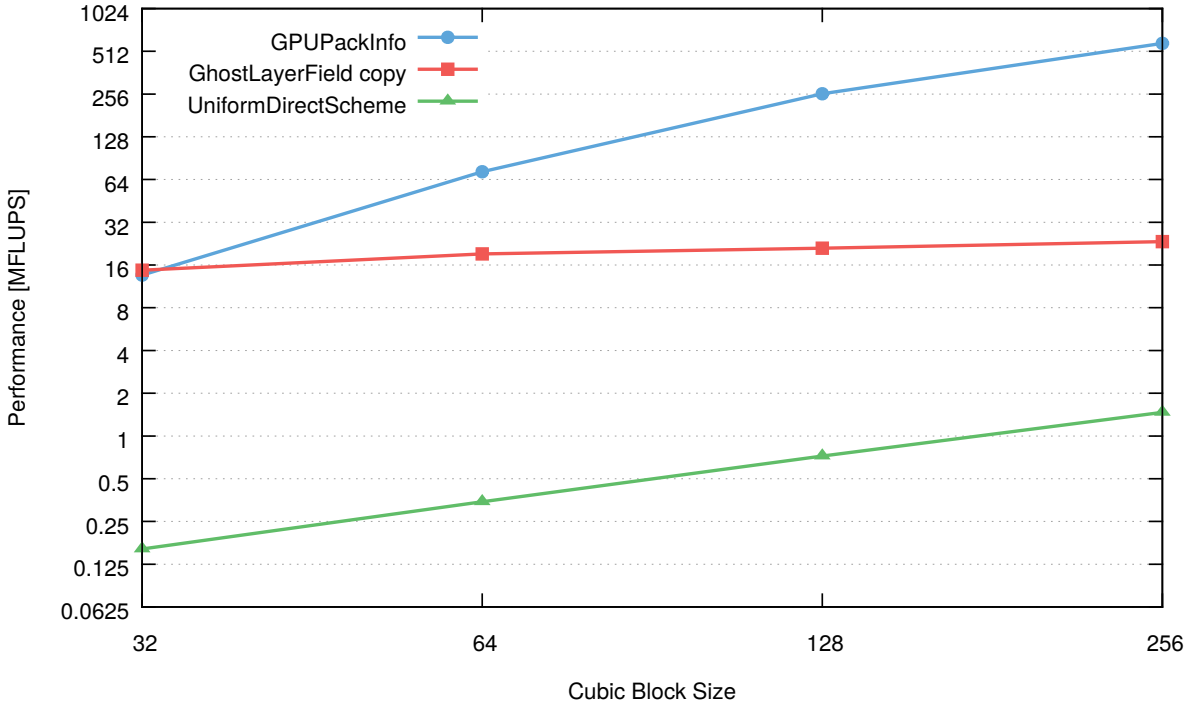


Figure 6.3: Performance comparison of the proposed communication solution (*GPUPackInfo*) with the previously available GPU communication solutions for waLBerla (*GhostLayerField* copy and *UniformDirectScheme*) for different block sizes.

The presented results demonstrate that the proposed GPU communication solution has the best performance among the available solutions for waLBerla.

6.2.3 Communication Overhead Results

This section presents and analyzes the results of the communication overhead experiments described in Section 5.3.3.

As depicted in Figure 6.4, the communication sub-step has a high overhead. For a block size of 128^3 cells, the communication sub-step takes 29 ms out of 33 ms of the total step time, i.e. 89% of the total step time, while the kernel sub-step takes only 4 ms. On the other hand, for a block size of 256^3 cells, the communication wall-clock time is 84 ms out of 115 ms, i.e.

73% of the total step time, while the kernel sub-step takes 31 ms. Therefore, even though the communication overhead is high, it tends to decrease as larger blocks are used.

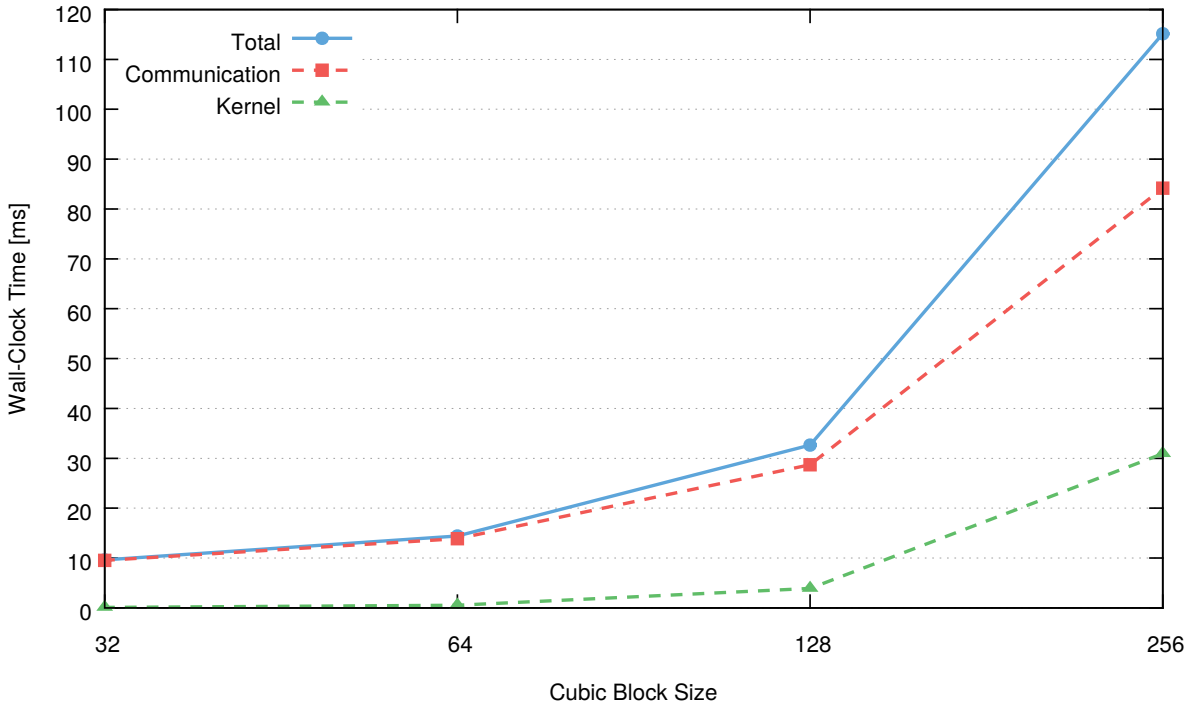


Figure 6.4: Wall-clock time of the communication and kernel execution sub-steps for different block sizes.

Although *GPUPackInfo* is the best GPU communication approach for waLBerla at the moment, the communication overhead results show that there is still much room for improvements. A major performance issue is the communication direction imbalance discussed in Section 5.3.4 with proposed solutions presented in Section 6.2.4. In addition, the communication performance can be improved as following:

1. The copy step of the GPU communication introduced in Section 4.2 is currently performed via the CUDA function *cudaMemcpy3D()*. Creating specialized kernels to perform the copies might improve performance [48].

2. The communication latency can be hidden by overlapping communication and computation by processing cells in the block near boundaries before processing inner cells [2]. As soon as cells in the block boundaries are processed, communication can be safely performed while the inner cells are being processed by the CUDA kernels.

3. Currently, when a block needs to communicate with its neighboring blocks, the copy step of the communication mechanism is done sequentially on the GPU. Allowing a block to communicate concurrently with all of its neighbors, which requires a unique CUDA stream for each neighbor, may improve the communication performance [48, 24].

4. waLBerla currently only supports Open MPI [46] for CUDA-aware MPI communication, however most of the recent literature makes use of MVAPICH2 [18] rather for its good InfiniBand support [36, 47, 48]. There is also a paper stating that MVAPICH2 has better performance than Open MPI [2]. Therefore, adding MVAPICH2 support for waLBerla may also improve communication performance.

6.2.4 Communication Direction Imbalance Results

This section presents and discusses the results of the experiments related to the communication direction imbalance described in Section 5.3.4.

When running the simulation with block size of 256^3 cells, the communication sub-step took 111 ms for the scenario with only x-direction communication while for the scenarios with communication in only y or only z direction, the communication sub-step took 13 ms. The communication overhead for the x-direction scenario is 78% of the total step time while it is only 30% for the scenarios with communication in y or z direction. As depicted in Figure 6.5, the poor performance of the x-direction communication also appears in all block sizes tested.

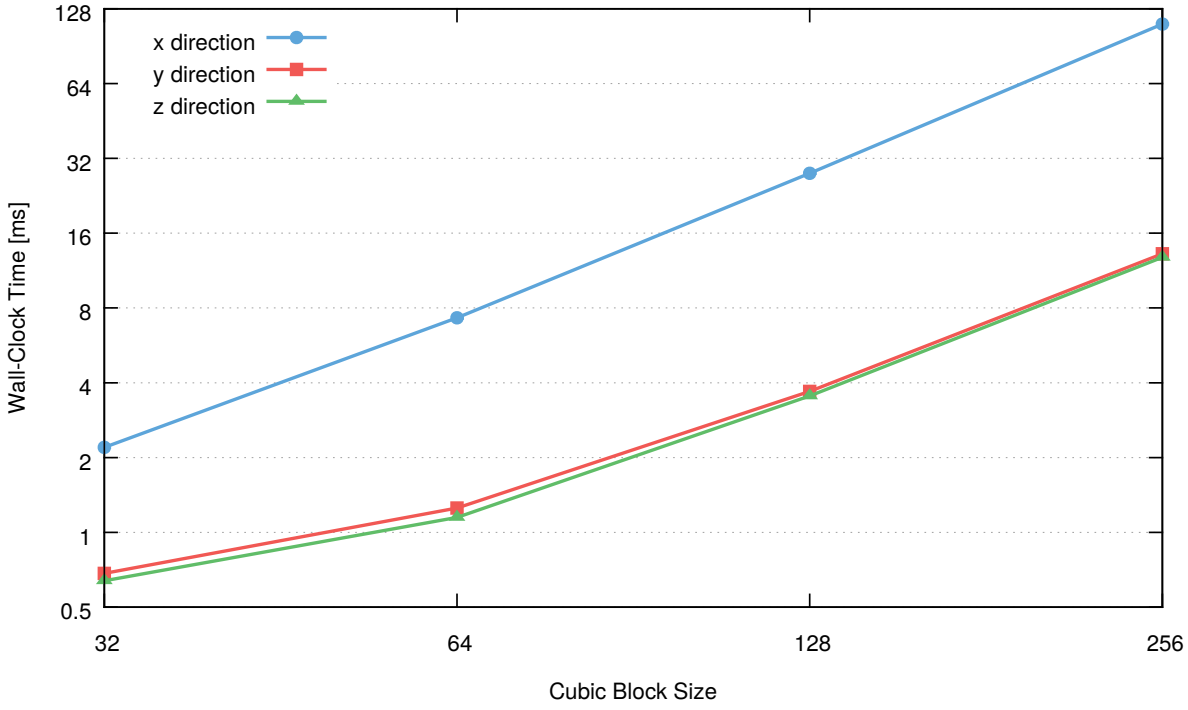


Figure 6.5: Wall-clock of the communication sub-step in different directions.

The discrepancy between the performance of the x-direction communication and the communication in y and z directions is confirmed and very high. Besides, x-direction communication was identified as the current communication performance bottleneck. The solutions to improve communication performance proposed in Items 2 and 3 of Section 6.2.3 have the potential to mitigate or even overcome this issue, provided that the kernel computation takes long enough to completely hide communication latency.

We propose additional solutions for improving the communication performance taking the communication direction imbalance into consideration. For relatively small domains, a simple method to improve the simulation performance is completely avoiding x-direction domain decompositions. For instance, regarding a domain decomposed into 2 blocks of 256^3 cells each, avoiding x-direction decompositions has shown to improve the simulation performance more than three times. Avoiding x-direction decompositions may also be adequate for large domains which are much thinner in the x-direction compared to the other axis directions. Even for domains thinner in other directions other than x, this method may also be a good choice as long as the domain is rotated in a preprocessing step in order to make it thinner in the x direction.

For many large domains, however, it is impractical to avoid x-direction decompositions. When fitting a simulation into an HPC system, avoiding x-direction decompositions may render

the block aspect ratio very disproportionate as many decompositions are made only in y and z directions. This may greatly impair performance due to a high surface-to-volume ratio of the blocks, which causes too many data to be communicated in relation to the total number of cells.

In case it is not feasible to avoid x-direction decompositions, taking the communication direction imbalance into consideration when decomposing a domain may still result in performance improvement. As the x-direction communication has a higher overhead than the other directions, a strategy to improve performance is getting the communication balanced among the x, y and z directions. This can be achieved by decomposing the domain more in y and z directions than in x direction, which causes the x direction having less data to communicate than y and z directions. This strategy of communication balance could not be tested because it requires a system with at least 27 GPUs and none of the available systems described in Section 5.1 meet this requirement.

6.2.5 Scalability Results

This section presents and analyzes the results of the weak and strong scaling experiments described in Section 5.3.5.

Figure 6.6 depicts the weak and strong scalings for yz and xz decompositions. The theoretical performance upper limit based on the ideal scenario where there is no overhead of including more GPUs to process the simulation is also depicted.

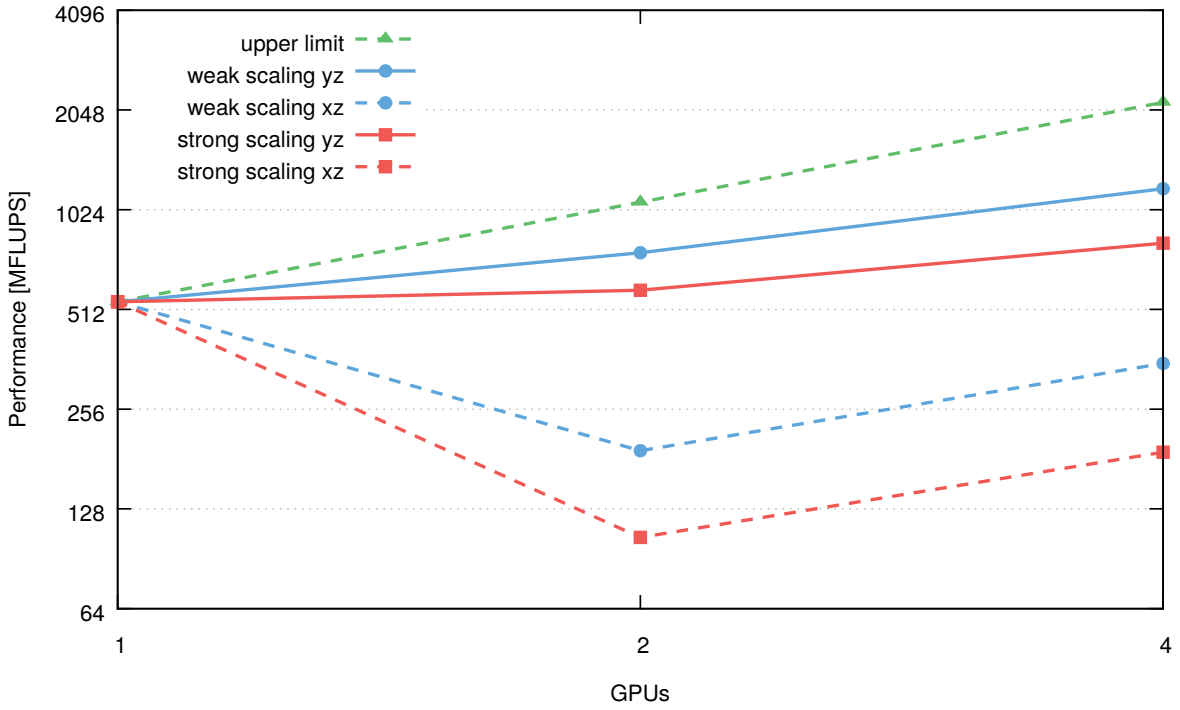


Figure 6.6: Weak and strong scaling. For weak scaling, block size is fixed to 320^3 cells. For strong scaling, domain size is fixed to 320^3 cells.

For weak scaling, the yz decomposition has an increasing performance as more GPUs are added, reaching 1184 MFLUPS when running on 4 GPUs. The xz decomposition, on the other hand, loses performance when the x-direction decomposition is made, dropping from 540 MFLUPS when running on a single GPU to 192 MFLUPS when running on 2 GPUs,

gaining performance again when 2 more GPUs are added through a z-direction decomposition, performing 352 MFLUPS when running on 4 GPUs.

For strong scaling, the yz decomposition also has an increasing performance as more GPUs are added, reaching 811 MFLUPS when running on 4 GPUs. The xz decomposition presents the same problem as in weak scaling, losing performance when the x-direction decomposition is made, dropping from 540 MFLUPS when running on a single GPU to 105 MFLUPS when running on 2 GPUs and gains performance again when 2 more GPUs are added through a z-direction decomposition, performing 190 MFLUPS when running on 4 GPUs.

Note that for both weak and strong scalings, the yz decomposition scales well. Even though the restriction of 4 GPUs imposed by the system does not allow defining where the scalability curve stabilizes, an increasing scalability is noticeable from 2 to 4 GPUs. On the other hand, the poorer performance of the x-direction communication described in Section 5.3.4 is responsible for the poorer performance of the xz decomposition. However, the restriction of 4 GPUs does not allow to determine whether or not the x-direction communication scales for a high number of GPUs.

As the single-GPU scenario performs 540 MFLUPS, the performance upper limits for 2 and 4 GPUs are, respectively, 1080 and 2160 MFLUPS. The performance for both scalings can get closer to the upper limit by improving the communication performance as proposed in Section 6.2.3 and Section 6.2.4.

6.3 Memory Usage Results

This section presents the results and analysis of the experiments related to the memory usage introduced in Section 5.4.

6.3.1 Memory Efficiency Results

This section presents the results of the memory efficiency test described in Section 5.4.1 and demonstrate how *GPUSweepBase* can provide an efficient use of the GPU memory by using multiple blocks per node.

Regarding a D3Q19 stencil and double precision, we have $M_c = 152$ bytes for Equation 5.5, therefore the estimated memory usage for 1, 8, 64 and 512 blocks are respectively 4995, 2883, 2726 and 2942 MB. As depicted in Figure 6.7, these values match exactly the measured memory usage of *GPUField* instances. As expected, the memory footprint is slightly bigger due to additional GPU memory used by the CUDA kernels and by the system.

The experiment demonstrates that the proposed solution provides an enhancement in the GPU memory usage. As depicted in Figure 6.7, decomposing a domain with 256^3 cells into 64 blocks provides the best memory efficiency among the tested decompositions by using about 55% of the memory that a single block domain uses.

6.3.2 Maximum Domain Size Results

This section presents the results of the maximum domain size experiment described in Section 5.4.2 for a single GPU of the system *cluster-ib* with approximately 11520 MB of memory. As long as part of the GPU's memory is used by the system and kernel execution, a ratio $r = 0.98$ of the total memory was assumed for calculations.

The Figure 6.8 shows the maximum domain size curve of a *cluster-ib*'s GPU for different decompositions. All scenarios executed successfully. The curve confirms that $B_\alpha = 4$ is the

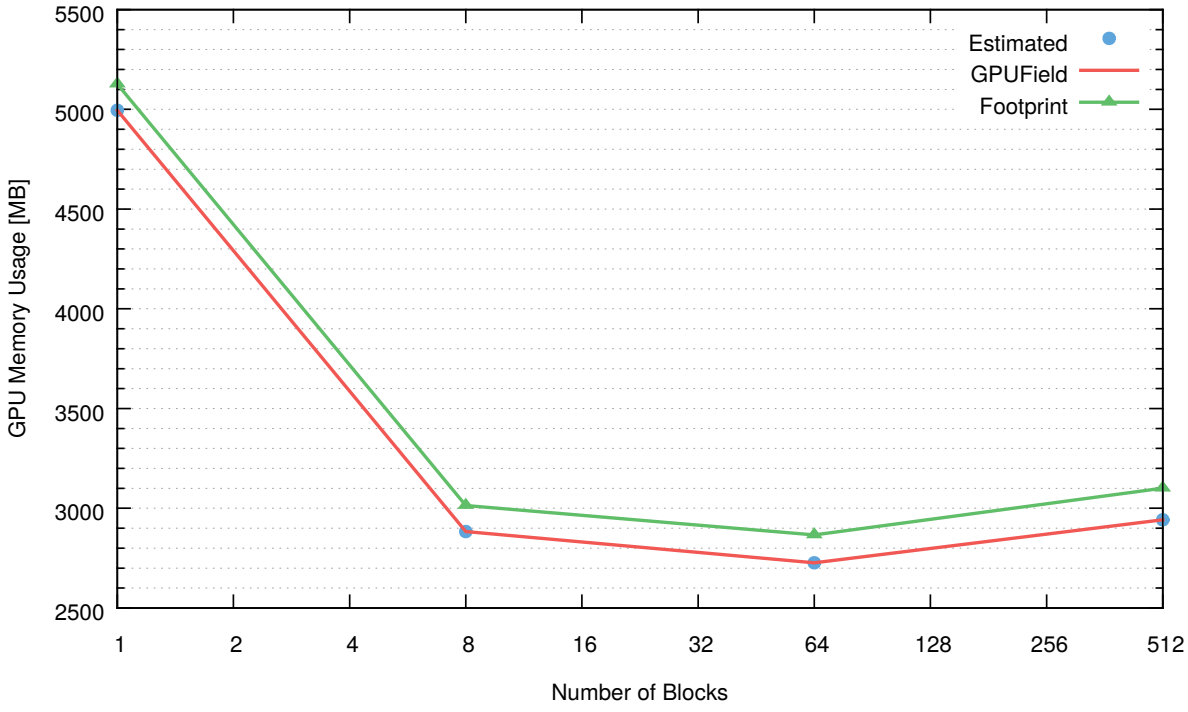


Figure 6.7: Memory efficiency for a domain with 256^3 cells.

optimal decomposition in terms of memory efficiency allowing a maximum cubic domain size of $N_\alpha = 412$ cells. On the other hand, $B_\alpha = 2$ and $N_\alpha = 404$ can also be considered for practical simulations as this scenario provides good memory efficiency while allowing a lower intra-GPU communication overhead as fewer blocks are used.

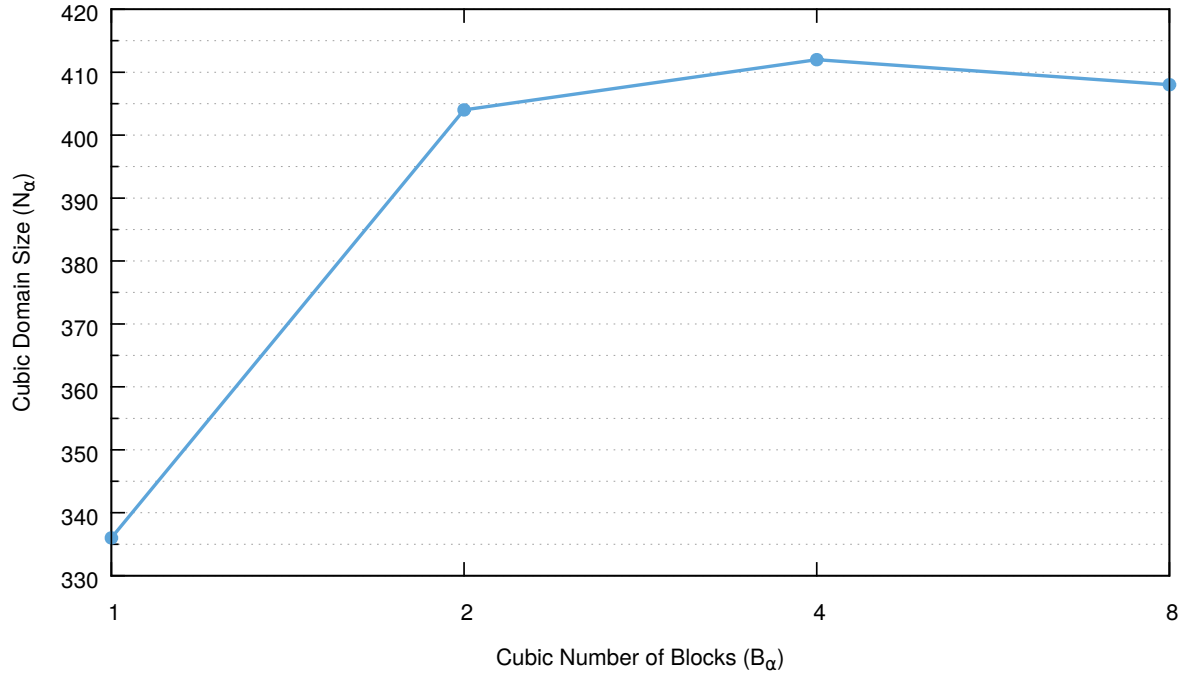


Figure 6.8: Maximum domain size that a GPU with 11520 MB of memory can simulate given the number of decomposition blocks.

Note that the proposed solution makes possible creating larger sub-domains per GPU. As a consequence, it allows for running simulations for larger total domains or using fewer GPUs for running simulations with a given domain size.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, we presented solutions for improving communication performance, memory usage efficiency and usability of GPUs in the HPC field, considering inter-node, intra-node and intra-GPU scenarios. Different GPU communication strategies and memory management mechanisms were evaluated.

The increasing importance of GPUs for general-purpose computation in supercomputers makes a good GPU support by HPC software frameworks an important feature. Therefore, embedding the proposed solutions into an HPC software framework so that they are easily accessible by researchers, engineers and programmers consists of a valuable contribution.

The GPU support previously available for the waLBerla framework has shown poor communication performance and issues related to memory usage efficiency. In order to improve its performance, memory usage efficiency and usability on GPU-based high performance computers, our proposed solutions were transparently integrated into the framework.

Our proposed GPU communication solution reuses much of the framework's communication architecture used for CPU and is compatible with many classes implementing CPU communication. Experiments have shown that the proposed GPU communication mechanism runs 25 times faster than the fastest GPU communication mechanism previously available for waLBerla.

A deep analysis of the GPU communication performance was presented. A bottleneck involving imbalance of communication directions was identified as well as the causes of the issue. Due to the lack of coalescent GPU memory access, communication in x direction has much poorer performance than communication in other axis directions. Solutions for that issue consisting of better strategies for decomposing the simulation domain were proposed.

The proposed mechanism for improving GPU memory usage efficiency consists of using for the destination domain only a fraction of the memory used by the source domain. It relies on the fact that decomposition blocks within the same process are calculated sequentially. Our experiment showed that the solution for memory efficiency allows for using only 55% of the memory that would be required by the naive approach of allocating the same amount of memory for source and destination domains. Thus, the solution allows for running simulations with larger domains or using fewer GPUs for running simulations with a given domain size.

The support for contiguous memory allocation in GPU was implemented by reusing most of the data structures and functions that had been used for pitched memory. In comparison with previously proposed contiguous memory implementation for waLBerla, it avoids code duplication and keep the framework's source code easier to maintain. The contiguous memory

showed to perform better than pitched memory in many scenarios, besides being more efficient in memory usage and more flexible in relation to domain sizes and decompositions.

A more flexible solution for indexing field data in GPU memory was also presented. It allows for configuring CUDA thread block sizes, which makes possible to fine tune indexing to favor memory access coalescence. It initially showed better performance than the previously available mechanism. As kernels were being increasingly optimized for this work, the previously available mechanism eventually surpassed it in performance. Nonetheless, as field indexing performance showed to be very sensitive to kernel implementation details, the proposed mechanism was kept in the framework for its flexibility.

A 3D LDC application was developed for performing all experiments and testing all presented solutions. The CUDA kernels that implement the LBM has shown to be highly optimized and suitable for experiments involving GPU communication performance.

Finally, there are still many challenges for utilizing the full potential of GPUs on supercomputers. Improving kernel performance and memory usage efficiency showed to bring valuable benefits, but improving communication performance is currently the biggest challenge. Transferring non-contiguous data efficiently is a major bottleneck for the GPU communication and is subject for future research. In addition, new technologies such as NVIDIA NVLink [29] for allowing ultra-fast GPU-to-GPU and GPU-to-CPU communication come with the opportunity for researching different approaches of communication optimization.

7.2 Future Work

During the development of the proposed solutions and by analyzing the results of the proposed experiments, the following possibilities for future work were identified:

- In the developed GPU communication solution, creating specialized kernels to perform CPU-GPU copies and GPU-GPU copies might improve communication performance [48]. Currently, for the D3Q19 stencil, 19 calls to *cudaMemcpy3D()* are required for a single 3D region copy.

- Using CUDA streams for allowing communication with all neighboring blocks to be executed concurrently in order to improve communication performance [48, 24].

- Implementing MVAPICH2 [18] support on waLBerla, which currently only supports Open MPI [46] for CUDA-aware MPI communication. Most of the recent literature makes use of MVAPICH2 for its good InfiniBand support [36, 47, 48]. Besides, Open MPI showed poor performance for non-contiguous data types when communicating directly from GPU memory. In case MVAPICH2 presents better CUDA-aware MPI communication for non-contiguous data types, the performance of *UniformDirectScheme* communication might increase considerable.

- Testing the proposed domain decomposition strategy for addressing the communication direction imbalance problem, which could not be tested because it requires a system with at least 27 GPUs, but none of the available systems met this requirement.

- Development of heterogeneous computing involving GPU and CPU computing nodes. The proposed GPU communication infrastructure is an important step towards the development of heterogeneous HPC for waLBerla. The next step consists of developing a new communication scheme like *UniformBufferedScheme* that can distinguish and properly

handle blocks allocated in GPU memory and blocks allocated in CPU main memory. It must also be able to select and use the appropriate *UniformPackInfo* for each type of block: *GPUPackInfo* for GPU blocks and *PackInfo* for CPU blocks. Finally, it is required to make sure that *GPUPackInfo* and *PackInfo* pack and unpack data using the same data layout in the buffers, so that data packed with *GPUPackInfo* can be unpacked with *PackInfo* and vice versa.

- Although *zyxf* memory layout usually presents poor performance on GPUs, implementing its support for the developed solutions might be useful for helping heterogeneous computing implementation and for load balance between GPU and CPU blocks, since *zyxf* has good performance on CPUs.

- Development of a handcrafted pitched memory allocation like in Habich et al. [13], which showed better performance than the pitched 512-byte aligned pitched memory currently supported by waLBerla. This pitched memory implementation should allow configurable alignment so that alignment can be adjusted for different problems and GPU model.

- Using CUDA streams for allowing boundary handling kernels to be executed concurrently on GPU in order to improve boundary handling performance.

- The overall simulation performance can be improved by overlapping communication and computation [2]. It can be developed by processing cells in the block near boundaries before processing inner cells. As soon as cells in the block boundaries are processed, communication can be safely performed while the inner cells are being processed by the CUDA kernels.

- The latest nvcc versions such as v7.5 bring good support to C++11, which was not the case when support for CUDA-enabled NVIDIA GPUs was introduced to waLBerla. The better support to C++11 features allows for supporting the standard CUDA kernel launch syntax, currently available exclusively via the *Kernel* class. Besides, it allows for implementing direct *GPUField* element access from inside CUDA kernels, which is currently available exclusively via field indexing/accessor classes.

References

[1] Martin Bauer, Florian Schornbaum, Christian Godenschwager, Matthias Markl, Daniela Anderl, Harald Köstler, and Ulrich Rüde. A python extension for the massively parallel multiphysics simulation framework walberla. *International Journal of Parallel, Emergent and Distributed Systems*, pages 1–14, December 2015.

[2] Bernaschi, M., Bisson, M., Fatica, M., and Phillips, E. An introduction to multi-gpu programming for physicists. *Eur. Phys. J. Special Topics*, 210:17–31, 2012.

[3] D.A. Case, V. Babin, J.T. Berryman, R.M. Betz, Q. Cai, D.S. Cerutti, T.E. Cheatham, III, T.A. Darden, R.E. Duke, H. Gohlke, A.W. Goetz, S. Gusarov, N. Homeyer, P. Janowski, J. Kaus, I. Kolossváry, A. Kovalenko, T.S. Lee, S. LeGrand, T. Luchko, R. Luo, B. Madej, K.M. Merz, F. Paesani, D.R. Roe, A. Roitberg, C. Sagui, R. Salomon-Ferrer, G. Seabra, C.L. Simmerling, W. Smith, J. Swails, R.C. Walker, J. Wang, R.M. Wolf, X. Wu, and P.A. Kollman. *The Amber Molecular Dynamics Package*, April 2015 (accessed on April 12, 2015). `http://ambermd.org/`.

[4] Universität Erlangen-Nürnberg. *waLBerla Framework Documentation*, April 2015 (accessed on April 12, 2015). `http://www.walberla.net/doxygen/`.

[5] Universität Erlangen-Nürnberg. *LSS – Department of System Simulation*, April 2015 (accessed on April 20, 2015). `https://www10.informatik.uni-erlangen.de/`.

[6] Universität Erlangen-Nürnberg. *waLBerla Framework*, March 2015 (accessed on March 30, 2015). `http://www.walberla.net/`.

[7] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. Walberla: Hpc software design for computational engineering simulations. *Journal of Computational Science*, 2(2): 105–112, 2011. Simulation Software for Supercomputers.

[8] Christian Feichtinger. A flexible patch-based lattice boltzmann parallelization approach for heterogeneous gpu–cpu clusters. *Parallel Computing*, 37(9):536–549, 2011.

[9] Christian Feichtinger. *Design and Performance Evaluation of a Software Framework for Multi-Physics Simulations on Heterogeneous Supercomputers*. PhD thesis, Universität Erlangen-Nürnberg, July 2012.

[10] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rüde, and Takayuki Aoki. Performance modeling and analysis of heterogeneous lattice boltzmann simulations on cpu–gpu clusters. *Parallel Computing*, 46(0):1–13, 2015.

[11] C. Godenschwager, F. Schornbaum, M. Bauer, H. Kostler, and U. Rude. A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–12, November 2013.

[12] J. Habich, T. Zeiser, G. Hager, and G. Wellein. Performance analysis and optimization strategies for a d3q19 lattice boltzmann kernel on nvidia gpus using cuda. *Advances in Engineering Software*, 42(5):266–272, 2011. PARENG 2009.

[13] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein. Performance engineering for the lattice boltzmann method on gpgpus: Architectural requirements and performance results. *Computers & Fluids*, 80(0):276–282, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[14] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall/CRC Computational Science. CRC Press, 2010.

[15] Mark Harris. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*, January 2015 (accessed on July 15, 2016). `https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/`.

[16] IBM. *IBM Platform MPI*, April 2015 (accessed on April 20, 2015). `http://www.ibm.com/systems/platformcomputing/products/mpi/`.

[17] Kitware, Sandia National Laboratories, Los Alamos National Laboratory, Army Research Laboratory, and CSimSoft. *ParaView*, May 2016 (accessed on May 1, 2016). `http://www.paraview.org/`.

[18] Network-Based Computing Laboratory. *MVAPICH2: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE*, June 2016 (accessed on June 28, 2016). `http://mvapich.cse.ohio-state.edu/`.

[19] John D. McCalpin. *STREAM Benchmark*, April 2015 (accessed on April 6, 2015). `https://www.cs.virginia.edu/stream`.

[20] A.A. Mohamad. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. SpringerLink: Bücher. Springer London, 2011.

[21] NVIDIA. *Tesla C2050 and Tesla C2070 Computing Processor Board Specification*, September 2011.

[22] NVIDIA. *Tesla C2075 Computing Processor Board Specification*, September 2011.

[23] NVIDIA. *Tesla K40 GPU Active Accelerator Board Specification*, November 2013.

[24] NVIDIA. *CUDA C Programming Guide Version 7.5*, September 2015. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[25] NVIDIA. *CUDA Runtime API v7.5*, September 2015. `http://docs.nvidia.com/cuda/cuda-runtime-api`.

[26] NVIDIA. *CUDA Toolkit Documentation v7.5*, September 2015. `http://docs.nvidia.com/cuda`.

[27] NVIDIA. *GPUDirect*, April 2015 (accessed on April 12, 2015). `https://developer.nvidia.com/gpudirect`.

[28] NVIDIA. *MPI Solutions for GPUs*, April 2015 (accessed on April 20, 2015). `https://developer.nvidia.com/mpi-solutions-gpus`.

[29] NVIDIA. *NVLink High-Speed Interconnect*, July 2016 (accessed on July 29, 2016). `http://www.nvidia.com/object/nvlink.html`.

[30] NVIDIA. *CUDA FAQ*, May 2016 (accessed on May 13, 2016). `https://developer.nvidia.com/cuda-faq`.

[31] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. A new approach to the lattice boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12):3628–3638, 2011. Mesoscopic Methods for Engineering and Science – Proceedings of ICMMES-09 Mesoscopic Methods for Engineering and Science.

[32] Federal University of Paraná. *DInf – Department of Computer Science*, April 2015 (accessed on April 20, 2015). `http://www.inf.ufpr.br/`.

[33] Kim Olsen, Steven Day, and Yifeng Cui. *AWP-ODC – Anelastic Wave Propagation*, April 2015 (accessed on April 20, 2015). `http://hpgeoc.sdsc.edu/AWPODC/`.

[34] Steve Plimpton, Aidan Thompson, Paul Crozier, and Axel Kohlmeyer. *LAMMPS Molecular Dynamics Simulator*, April 2015 (accessed on April 12, 2015). `http://lammps.sandia.gov/`.

[35] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda. Optimizing mpi communication on multi-gpu systems using cuda inter-process communication. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1848–1857, May 2012.

[36] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D.K. Panda. Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89, October 2013.

[37] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 2nd edition, 1992.

[38] P.R. Rinaldi, E.A. Dari, M.J. Vénere, and A. Clausse. A lattice-boltzmann solver for 3d fluid simulation on GPU. *Simulation Modelling Practice and Theory*, 25:163–171, 2012.

[39] Ankit Rohatgi. *WebPlotDigitizer User Manual Version 3.9*. Austin, Texas, USA, October 2015. `http://arohatgi.info/WebPlotDigitizer/`.

[40] C. Rosales. Multiphase lbm distributed over multiple gpus. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 1–7, September 2011.

[41] Joseph Schneible, Lubomir Riha, Maria Malik, Tarek El-Ghazawi, and Andrei Alexandru. Communication efficient work distributions in stencil operation based applications. *Concurrency and Computation: Practice and Experience*, 2014.

[42] José Aurimar Sepka, Jr. Extensão do framework walberla para uso de gpu em simulações do método de lattice boltzmann. Master's thesis, Universidade Federal do Paraná, Paraná – SC, August 2015.

[43] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian R. Trott, Greg Scantlen, and Paul S. Crozier. The development of mellanox/nvidia gpudirect over infiniband – a new model for gpu to gpu communications. *Computer Science – Research and Development*, 26(3–4):267–273, 2011.

[44] Erich Strohmaier, Jack Dongarra, Horst Simon, Martin Meuer, and Hans Meuer. *TOP500 Supercomputers*, April 2015 (accessed on April 12, 2015). `http://www.top500.org/`.

[45] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. *June 2016 TOP500 Supercomputers*, June 2016 (accessed on July 21, 2016). `https://www.top500.org/lists/2016/06/`.

[46] The Open MPI Development Team. *Open MPI: Open Source High Performance Computing*, April 2015 (accessed on April 19, 2015). `http://www.open-mpi.org/`.

[47] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. Optimized non-contiguous mpi datatype communication for gpu clusters: Design, implementation and evaluation with mvapich2. In *2011 IEEE International Conference on Cluster Computing*, pages 308–316, September 2011.

[48] Hao Wang, S. Potluri, D. Bureddy, C. Rosales, and D.K. Panda. Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10):2595–2605, October 2014.