UNIVERSIDADE FEDERAL DO PARANÁ

JOÃO VICTOR TOZATTI RISSO

SCALABLE GPU COMMUNICATION FRAMEWORK FOR STENCIL BASED APPLICATIONS

CURITIBA PR
2017

JOÃO VICTOR TOZATTI RISSO

SCALABLE GPU COMMUNICATION FRAMEWORK FOR STENCIL BASED APPLICATIONS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Daniel Weingaertner.

CURITIBA PR
2017

Termo de aprovação

Esta folha deve ser substituída pela ata de defesa ou termo de aprovação devidamente assinado, que será fornecido pela secretaria do programa após a defesa ter sido concluída e aprovada (vide arquivo `aprovacao.tex`*).*

To my parents, Rosemar and Edson, for their love and unconditional support, and for everything they have always done for me.

Acknowledgements

To my family, specially my grandmother Nadir,

For loving and supporting me throughout the time I was studying away from home.

To my advisor, Prof. Daniel Weingaertner,

For believing in my work, and the fruitful discussions throughout the development of this work.

To Paulo Carvalho,

For his help with the concepts, code and scripts that were heavily utilized in producing this work.

Abstract

As the importance of GPUs increase for general-purpose computing in supercomputers, so does the value and necessity of efficient GPU communication solutions in High-Performance Computing (HPC) software frameworks, such as waLBerla. In this work, we present optimizations to the state-of-the-art GPU communication solution, and an extension to the communication infrastructure to use buffers in GPU memory. Proposed solutions increase the performance of GPU Lattice Boltzmann simulations by 50%, in a worst case scenario for communication, with respect to the state-of-the-art communication solution. Communication latency was also overlapped with computations, and the latency could be hidden by 50%, also in a worst case scenario for communication.
Keywords: GPU communication engineering, waLBerla framework, Lattice Boltzmann method.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BGK	Bhatnagar-Gross-Krook Model
CFG	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
GbE	Gigabit Ethernet
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HPC	High-Performance Computing
IB	Infiniband
LBGK	Lattice Bhatnagar-Gross-Krook Model
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 Function
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

ρ	Macroscopic fluid density
ν	Macroscopic fluid viscosity
Δx	Lattice cell spacing in the x-direction
t	Current time step
Δt	Time step size
Ω_α	Lattice Boltzmann collision operator
c_s	Speed of sound
τ	Lattice Boltzmann relaxation parameter
$\vec{u}$	Discrete lattice velocity
$\vec{e}$	Stencil direction vector
g	Thickness of the ghost layer

Chapter 1

Introduction

High-Performance Computing (HPC) consists in aggregating computer resources to achieve a much higher performance than of a single desktop or server could deliver. Advances in HPC technologies and techniques over the years, allowed scientists and engineers to tackle even more complex problems, on unprecedented scales. Computational Fluid Dynamics (CFD) is one of the fields that greatly benefits from these advances, and is particularly important for the numerical simulation of physical phenomena.

A list of the high-end supercomputers in the world is organized in the TOP500 list [39]. Most supercomputers in the list are based on a cluster architecture, in which the application is parallelized using a distributed paradigm, often using a message-passing scheme. In the distributed paradigm, computations are performed by independent processes running on the nodes of the cluster, and data is exchanged through message-passing. These computing nodes are usually interconnected in a local area network with fast, low-latency network technologies, such as Gigabit Ethernet (GbE) or Infiniband (IB).

Graphics Processing Units (GPUs) have evolved into scalable parallel processors, with the introduction of general-purpose computation APIs, such as CUDA [27]. Advances in GPU performance, price and energy consumption in each new generation, specially for floating-point computations, make these processors appealing to high-end HPC systems, since they can accelerate application performance considerably. According to the TOP500 list [39], as of November of 2016, 21 of the top 100 supercomputers utilize NVIDIA GPUs. Ranked among the top 10 systems that use GPUs are the Titan (3rd position) and Piz Daint (8th position) supercomputers.

Although using GPUs as accelerators or main computation units in supercomputers have many advantages, achieving optimal performance is challenging. Since GPUs introduce their own memory space, often separated from the host CPU memory, communication strategies usually have to rely on staging data through the CPU as an intermediate step. Staging data through the host introduces additional overhead and latency to communication, reducing overall efficiency. Solutions have been proposed to remediate the problem (*e.g.* GPUDirect [26]), however efficient communication strategies are still necessary to obtain optimal performance. Therefore, improving communication performance and achieving a good balance between computation and communication are still subject of study and discussion.

Several methods for numerical simulation of Computation Fluid Dynamics (CFD) problems have been proposed [7, 15]. Stemming from gas cellular automata, the Lattice Boltzmann Method is one of the most widely accepted methods in academia and industry to numerically simulate CFD problems involving incompressible flows [11]. The method has gained increasingly importance, because it can take advantage of massively parallel supercomputers.

HPC software frameworks drastically reduce the development time and effort of applications. waLBerla [14] is a HPC framework for multi-physics simulations, which was originally designed for LBM applications. The framework provides data structures and routines that ease the development of HPC applications. Although waLBerla supports a wide range of physical phenomena and runs on some of the top 10 supercomputers in the world, its GPU communication infrastructure still needs to be further improved to achieve optimal performance.

In this thesis, a new communication infrastructure that bypasses host-based data staging, and extensions to the state-of-the-art GPU communication infrastructure of the waLBerla framework are presented. The main goal is to improve the performance and usability of the framework in GPU-based supercomputers.

1.1 Objectives

In general, the objectives of this thesis are:

- Evaluate different GPU communication strategies, accounting for different communication scenarios: inter-node, intra-node and intra-GPU.
- Hide communication latency by overlapping communication and computations.
- Provide a flexible mechanism for GPU memory alignment, in order to exploit the underlying hardware characteristics.
- Improve walBerla's performance and usability on modern GPU-based supercomputers.

1.2 Contributions

The main contributions of this thesis to the HPC field are:

- An extension to the state-of-the-art GPU communication infrastructure, such that it can take advantage of CUDA concurrency mechanisms.
- A novel GPU-to-GPU communication infrastructure for stencil-based applications, which can make use of CUDA support in modern MPI implementations, thus avoiding data staging through host memory.
- Support for all field memory layouts in the GPU communication infrastructure of the framework.
- A significant performance increase in GPU-based LBM simulations using waLBerla.

Chapter 2

Background

In this chapter, we present the background for the solutions presented in this work. First, a brief overview of the architecture of modern GPUs is presented. Then, CUDA, MPI as well as CUDA-aware MPI are introduced. The lattice Boltzmann method (LBM) and a relevant benchmark problem, the 3-D lid-driven cavity, are briefly presented. Finally, an overview of the waLBerla framework is presented, with the most relevant details of its architecture.

2.1 GPU Architecture

A GPU is a multiprocessor comprised of several multi-threaded SIMD processors (also called streaming multiprocessors or SMs). GPU designs, contrary to CPUs, use most of the die area for several parallel processing units and specialized memories for graphics applications (e.g. texture memory), rather than big caches and control flow logic [24]. The main assumption in GPU designs is that applications have so many threads, that multi-threading can both hide the latency to global memory and increase the utilization of multi-threaded processors [30]. GPUs are particularly well-suited to data parallel and high arithmetic intensity problems, since the multi-threaded design of streaming processors allow for multiple concurrent threads to be executed independently on data elements, and the presence of multiple floating-point and special-purpose units increase the throughput of numerical computations.

In NVIDIA GPUs, parallel threads are scheduled in groups of 32 threads[1], which are called warps. There are two types of hardware schedulers: a warp scheduler and a thread block scheduler. A warp scheduler is a hardware scheduler, present in each streaming multiprocessor, which keeps track of instructions' status using a scoreboard, and schedules warps of threads when the instructions are ready for execution, thus exploring instruction-level parallelism (ILP). The thread scheduler is a hardware scheduler at the GPU level, which assigns and keeps track of blocks of threads in the multiprocessors, until kernel execution is completed.

As previously mentioned, besides multiple parallel execution units, GPUs employ different types of memories, mainly because of graphics applications. Threads may access data from any of these types of memories. The memory hierarchy is composed of:

- Global memory: stored in external DRAM, divided in several banks and is accessible by all SMs.
- Shared memory: used for thread block communication and is only visible within the same thread block, *i.e.* threads in the same SM.

[1] Assuming there is no divergence in the control flow.

- Local memory: per-thread local memory is private and only visible to a single thread. Local memory is allocated to external DRAM, in order to support larger allocations, and can be cached on-chip.
- Constant memory: read-only to a program running on the SM, it is stored in external DRAM and cached in the SM.
- Texture memory: holds large read-only arrays of data. Textures are cached in a streaming cache designed to optimize throughput of texture fetches from concurrent threads.

2.2 CUDA

Compute Unified Device Architecture (CUDA) is a scalable general-purpose parallel computing platform and programming model [24]. It eases programming and usage of GPUs for general computations, through extensions to the C, C++ and Fortran programming languages. There are three main abstractions in CUDA: a hierarchy of thread groups, shared memories and barrier synchronization.

GPU hardware has schedulers to handle parallel execution and thread management, it is not done by applications or the operating system. To simplify scheduling by the hardware, CUDA requires that thread blocks be able to execute independently and in any order.

Computation on the GPU is started by issuing kernels. Kernel is a code or function that executes on the GPU, and each kernel spans a grid. A CUDA grid is a group of thread blocks, and thread blocks are groups of threads, making use of the parallelism provided by GPU hardware, as described in Section 2.1. Grids and thread blocks can be one, two or three dimensional. Cache, grid and thread block configurations can be changed before each kernel call.

2.2.1 GPUDirect

GPUDirect is a set of CUDA features and extensions to the CUDA driver, which reduces data movement overhead when multiple GPUs or network adapters are utilized. There are three main features provided by GPUDirect: peer-to-peer memory access, peer-to-peer memory copies, and remote direct memory access (RDMA). Peer-to-peer (GPUDirect P2P) memory accesses and copies can only be used if the devices are in the same PCIe root complex [26, 25], and the same applies to RDMA operations with network adapters. GPUDirect RDMA consists in direct memory copies between GPU devices and Infiniband network cards, effectively bypassing host memory for data transfers between nodes [32].

The first version of GPUDirect was released in 2010, and allowed GPUs and network adapters to share the same pinned memory region on host memory, thus eliminating the need for one intermediate copy on the host side. GPUDirect Peer-to-peer was introduced in CUDA Toolkit version 4 (2011), and allows efficient DMA transfers between GPUs sharing the same PCIe root complex, as well as peer-to-peer memory accesses (similar to NUMA) within CUDA kernels. GPUDirect RDMA was introduced in the CUDA Toolkit version 5 (2012), and enables remote direct memory access (RDMA) transfers between GPUs and other PCIe devices, reducing memory copy overhead for inter-node communication.

2.2.2 Concurrency Mechanisms

In order to explore further parallelism in the GPU, CUDA offers concurrency mechanisms in the runtime API, which allow the programmer to launch asynchronous and concurrent

operations to the same device. These mechanisms were further extended by the introduction of HyperQ, which allow different threads to control and issue work to a given device. GPU hardware also provides different queues for operations, two for memory transfers in different directions and one for computing, which can be used to overlap asynchronous memory copies with computation.

Concurrency is exposed in CUDA through streams, which is a sequence of operations that execute in issue-order on the GPU. Operations in different streams may run concurrently or be interleaved, when there are resources available, no dependencies and no synchronization constructs. Explicit synchronization can be performed with streams and events, besides synchronization of the whole device. Other operations might also implicitly synchronize all other CUDA operations, such as memory allocations, synchronous memory copies and change to cache configuration.

2.3 MPI

MPI is a message-passing interface specification [13], that is standardized by a community of different stakeholders, e.g. parallel computing vendors, application developers and researchers. MPI stands for Message Passing Interface and it is one of the most utilized standards for distributed-memory parallel programming. The distributed-memory model of parallelization consists in message exchanges between a set of processes over a network or shared memory, i.e. a process cannot directly access another process' memory, so they must exchange data through messages. Several MPI implementations are available, actively maintained and in widespread use. Some implementations require licensing and are closed-source [19, 9], while others are open-source and maintained by a community [42, 22, 41].

All processes taking part in a parallel computation can be distinguished inside a communication group by a unique identifier called rank. The same program runs on all processes, and is written in a sequential language, i.e. C or Fortran [13, 18]. Processes are grouped in communicators, and a default communicator is assigned to all processes on the initialization phase. Different communication patterns and topologies are supported by MPI. Communication can be point-to-point (using sends and receives), one-directional (put and get) or collective (broadcast, reduction, gather and scatter). Virtual topologies are also supported, for example, cartesian grids, hypercubes or graphs.

2.3.1 CUDA-aware MPI

CUDA-aware MPI is a implementation of the MPI standard that allows the use of GPU memory [21], e.g. pointers allocated by the CUDA runtime, with MPI operations and datatypes. After the introduction of Unified Virtual Addressing (UVA) in CUDA 4.0, MPI implementations were able to integrate support for GPU pointers (and hence, buffers). UVA allowed for mapping of host and device memory allocations to the same virtual addressing space, and that in turn allowed host and device pointers to be distinguished through queries to the CUDA runtime API, which keeps track of GPU memory allocations. Through these pointer queries, the MPI implementation can handle data movement and other operations transparently, with the appropriate functions for host and device.

Optimizations to the underlying algorithms in MPI implementations have already been proposed in the literature [32, 33, 43, 8], with most of them implemented in MVAPICH2 [22, 44]. These optimizations improve the performance of CUDA-MPI programs, and allow the MPI

implementation to take advantage of advanced CUDA features such as GPUDirect P2P and RDMA. More details on some of these optimizations are presented in Section 3.

2.4 Lattice Boltzmann Method

The Lattice Boltzmann Method is an alternative to Navier-Stokes solvers (e.g. staggered grid) for incompressible flows, which satisfies the Navier-Stokes equations in the macroscopic limit with second order accuracy [7, 40]. It is based on a velocity discrete Boltzmann equation with an appropriate collision term. In 3D simulations, the simulation domain is typically discretized into a uniform cartesian grid [14]. In general, the Lattice Boltzmann equation can be written as

$$f_\alpha\left(x_i + e_\alpha \Delta t, t + \Delta t\right) - f_\alpha\left(x_i, t\right) = \Omega_\alpha\left(f\right) \tag{2.1}$$

with x_i denoting the i-th cell in the discretized simulation domain, e_α denoting the dimensionless discrete velocity set $\{e_\alpha | \alpha = 0, \ldots, N-1\}$, t denoting the current time step, Δt denoting the time step size, and $\Omega_\alpha(f)$ denoting the LBM collision operator. Algorithmically, the LB equation is typically separated into a collision (2.3) and a streaming step (2.2):

$$\overline{f}_\alpha(x_i, t) = f_\alpha(x_i, t) + \Omega(f_\alpha) \tag{2.2}$$

$$f_\alpha(x_i + e_\alpha \Delta t, t + \Delta t) = \overline{f}_\alpha(x_i, t) \tag{2.3}$$

with $\overline{f}_\alpha$ denoting the post-collision state of the distribution function.

In this thesis, we use a D3Q19 model, as shown in Figure 2.1, with a single-relaxation-time (SRT/LBGK) model, introduced by Sepka *et. al* [37] and implemented by Carvalho *et. al* [10], which is given by

$$\Omega_\alpha(f) = -\frac{1}{\tau}\left(f_\alpha(x_i, t) - f_\alpha^{(\mathrm{eq})}\right) \tag{2.4}$$

where τ is the relaxation time and $f_\alpha^{(\mathrm{eq})}(x_i, t)$ is the equilibrium distribution.

The equilibrium distribution is given by equation (2.5):

$$f_\alpha^{(\mathrm{eq})} = \rho w_i \left[1 + \frac{3}{c^2}\vec{e}_i \cdot \vec{u} + \frac{9}{2c^4}(\vec{e} \cdot \vec{u})^2 - \frac{3}{2c^2}(\vec{u} \cdot \vec{u})\right] \tag{2.5}$$

where ρ and $\vec{u}$ are the macroscopic density and velocity obtained from the 0th and 1st moments of the particle distribution function f_i with regard to the discrete velocity $\vec{e}_i$

$$\rho = \sum_{i=0}^{18} f_i(\vec{x}, t) \qquad \rho\vec{u} = \sum_{i=0}^{18} \vec{e}_i f_i(\vec{x}, t)$$

where w_i are direction-dependent constants, given by Equation (2.6), for the D3Q19 model.

$$w_i = \begin{cases} 12/36, & \text{for } i = 0. \\ 2/36, & \text{for } i \in [1, \ldots, 6]. \\ 1/36, & \text{for } i \in [7, \ldots, 18]. \end{cases} \tag{2.6}$$

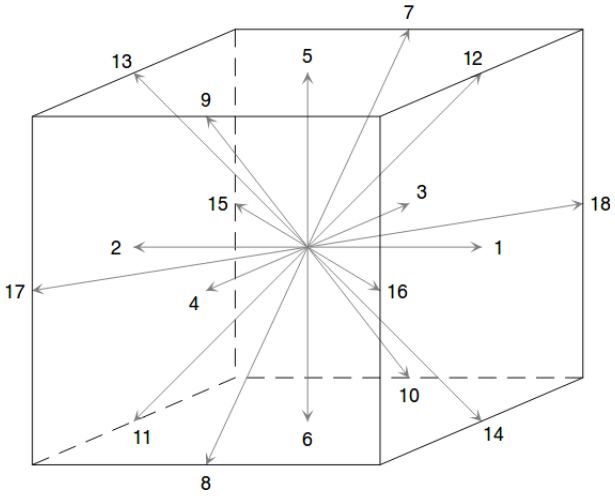

Figure 2.1: D3Q19 stencil [29] representation. The arrows represent the 19 stencil directions (center position is zero), with each arrow representing a neighboring cell in the corresponding direction.

c is the discrete lattice velocity, given by $c = \frac{\Delta x}{\Delta t}$, where Δx is the lattice spacing and Δt is the time step size. The pressure p and kinematic viscosity ν are given by

$$p = c_s \rho \qquad \nu = \frac{1}{6}(2\tau - 1)\Delta x\, c$$

where $c_s = 1/\sqrt{3}$ is the speed of sound and τ is a relaxation parameter.

2.5 Lid-Driven Cavity Problem

The lid-driven cavity problem is one of the most important benchmarks for numerical Navier-Stokes solvers and Lattice-Boltzmann methods [2]. Its importance stems from a simple driving of the flow using a tangential motion of a single lid at the top of the domain, using a constant velocity. Moreover, it is very simple, since it uses a regular square or cubic geometry for the domain and exhibits several interesting physical properties, such as the formation of eddies and counter-eddies [15].

In the lid-driven cavity problem in three dimensions, the physical setup consists in a cubic container filled with a fluid. The lid at the top of the container moves at a given, constant velocity, thereby setting the fluid in motion. No-slip conditions are imposed on all five segments of the boundary, with the exception of the top boundary, along which a velocity in the x-direction is not set to zero, but equal to the given lid velocity to simulate the moving lid. A representation of the problem is shown in Figure 2.2.

2.6 waLBerla Framework

waLBerla is a massively parallel software framework for multi-physics simulations. It was initially developed for simulation of fluid flows using the lattice Boltzmann method (LBM), hence the acronym waLBerla, which stands for “widely applicable lattice Boltzmann from

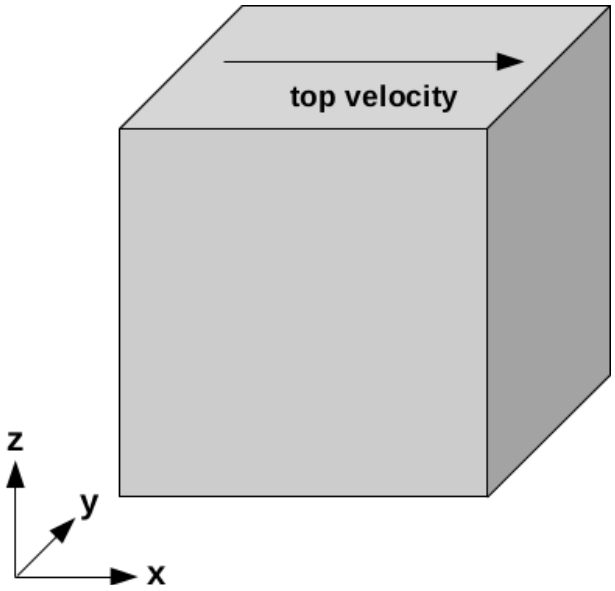

Figure 2.2: Representation of the simulation domain for the 3D lid-driven cavity problem [10].

Erlangen". Over time, the framework evolved into a general HPC framework for algorithms that utilize block-structured grids [3].

Through fully distributed data structures, it reduces the memory footprint of multi-process simulations. It also offers efficient algorithms for input and output of simulation data, domain partitioning and load balancing [14]. WaLBerla has two primary design goals: being efficient and scalable on current supercomputer architectures, while at the same time being flexible and modular to support different applications [12].

2.6.1 Application Structure

In waLBerla, users can construct their simulations in terms of operations, and several routines are already provided for geometry and simulation setup, communication, visualization and post-processing. Thus, it reduces the overall time and effort required to setup a simulation pipeline, allowing the user to focus solely on the application they want to simulate.

Below is a list of basic operations that are performed in a waLBerla application:

1. **Creation:** First step in the application, it usually sets up the block structure and performs the domain decomposition to the requested number of processes, by reading the configured number of processes either via Python [3] or through another configuration file format (*e.g.* a .prm file).

2. **Initialization:** In this step the initial domain quantities (*e.g.* particle distribution functions), domain obstacles, and other relevant aspects of the simulation are created.

3. **Operation definition:** An operation usually extracts block data from the block structure through the use of a block identifier. Operations execute over a block at a time, and they perform the actual computations of the simulation. For example, boundary handling and lattice Boltzmann codes are defined as different operations, they extract block data representing the simulation domain, and operate on it.

4. **Simulation loop:** Operations are added to a timeloop, which executes each operation over all blocks. The simulation loop represents the time stepping (advance the simulation with respect to time), which is very common in physical applications.

2.6.2 Framework Architecture

In this section, an overview of the relevant classes and data structures of the waLBerla framework are presented. The framework provides classes and functions that ease the development of HPC applications, while maintaining performance. A description of each relevant class is given below:

- **IBlock:** Base class for blocks, which defines basic operations for all types of blocks that might be implemented in the framework. Blocks are rectangular subsets of the simulation domain, *i.e.* the simulation is partitioned into one or more blocks. They also have an internal state and the framework design allows them to hold any kind of data. Therefore, a block can hold a flag field as well as a velocity field, for example.

- **StructuredBlockForest:** Manages the block structure, providing mechanisms for setting bounding boxes on the domain, grid refinement strategies, the physical size of each cell in x, y and z dimensions. It also provides mechanisms to convert between physical coordinates to simulation coordinates, and vice-versa.

- **Field:** A 4-D (x, y, z, f) data container, which uses contiguous memory to provide access to its data members. These members can be accessed regardless of the simulation's current time step. It supports two memory layouts[2]: array-of-structures (AoS), structure-of-arrays (SoA). In the array-of-structures (AoS or fzyx) layout, f is the slowest changing dimension, followed by z and y, and x is the fastest changing dimension. The structure-of-arrays (SoA or zyxf) layout has the z dimension as the slowest changing dimension, followed by y and x, and f is the fastest changing dimension.

- **GhostLayerField:** It is an extension of the Field class, adding the ghost layers, which are required for algorithms that rely on ghost layer based communication. The number of ghost layers can be adjusted on the initialization phase of the application, depending on the requirements of the latter.

- **GPUField:** This class is an implementation of the GhostLayerField class for CUDA GPUs, therefore making use of CUDA runtime functions and data structures for memory management. It supports the same memory layouts as the Field class (which is the parent class of GhostLayerField), and has functions to perform copies between CPU and GPU memory.

- **Kernel:** Wrapper class around a CUDA kernel. When CUDA support was introduced in waLBerla, the *nvcc* compiler did not support C++11 functionalities, which are used extensively in the framework. Using this wrapper, it is possible to compile CUDA-specific code separately with *nvcc*, while the rest of the code can be compiled with other compilers.

- **FieldIndexing:** This class implements an indexing scheme, which map field coordinates (x, y, z, f) to CUDA's threads and blocks. FieldIndexing maps all the elements of the innermost field coordinate within a thread block, as a mean to provide coalescent access to field data.

[2]Which are also called linearization strategies in the documentation.

- **FieldAccessor:** The FieldAccessor class is instantiated by the FieldIndexing, and provides transparent access to the GPUField data within the CUDA kernels. It computes the GPUField index for each thread, depending on the indexing strategy defined by the field indexing.

- **BlockSweep:** This is a functor class that must be implemented by the user, and operates on a block at a time. The framework traverses the block structure, passing each block's pointer to *BlockSweep::operator()* function. Any name can be used for this class, which is chosen by the user.

- **SweepTimeloop:** This class represents a loop with time stepping, which executes a collection of sweeps. Sweeps are functions that operate on a single block, and the framework provides a pointer to the block. With the block, the user can extract any registered data (*e.g.* velocity field). Before and after functions can also be defined for each sweep. Multiple sweeps are also supported for a given time loop.

- **UniformPackInfo:** This is an abstract class that provides the interface for insertion and extraction of data from blocks. Therefore, it enables communication between neighboring blocks. Other PackInfos must be derived from this class and implement concrete functions for its abstract interface. This ensures compatibility with the communication algorithms already implemented in waLBerla.

- **PackInfo:** Concrete implementation of the UniformPackInfo class, providing functions to insert and extract data from a determined type of block. It can be used, for example, to insert and extract information from blocks that reside on the GPU (GPUField).

- **UniformBufferedScheme:** This class implements the communication algorithm for uniform block grids. It implements the MPI communication for neighboring blocks, specifically when these blocks have to be communicated all at once. For example, one can synchronize multiple ghost layers fields in a single step, by exchanging data with processes that hold neighboring blocks' data. To extract and insert data from and into the blocks, the user can register one or more PackInfo's, depending on the data that has to be communicated. The communication process, which is also called ghost layer communication, is depicted in Figure 2.3.

- **SendBuffer:** Wrapper class for MPI send buffer, providing convenience functions for adding and manipulating data within the underlying MPI buffer.

- **RecvBuffer:** Wrapper class for MPI receive buffer, also providing a set of convenience functions to extract and manipulate data within the underlying MPI buffer.

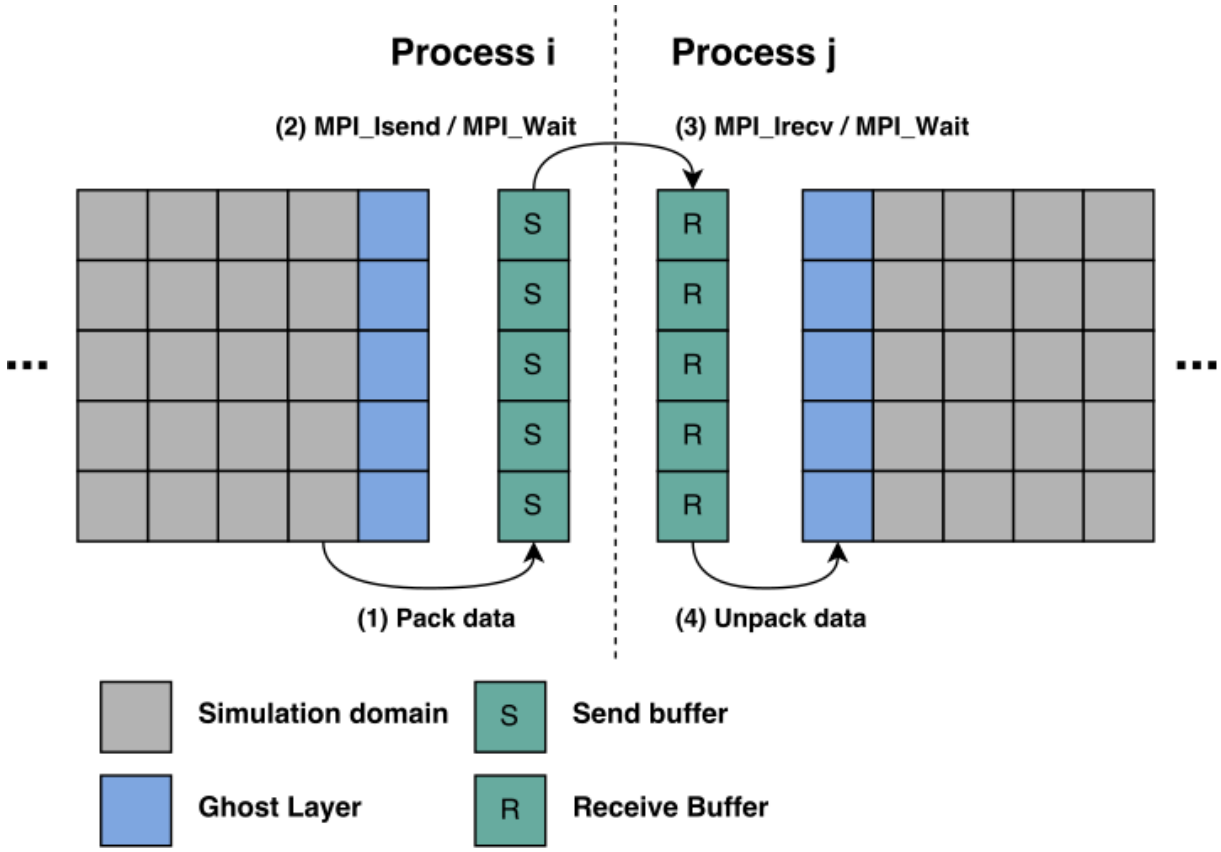

Figure 2.3: Ghost layer communication from a source (Process i) to a destination (Process j) process. Data is packed from the boundary layer of the field into the send buffer. Then, buffers are exchanged via MPI and the destination process unpacks the data from the receive buffer into the ghost layer of the destination field.

Chapter 3

Literature Review

3.1 Optimized Kernels

In this section, a literature review of works focused on single block GPU simulations with optimized lattice Boltzmann method (LBM) kernels is presented.

Obrecht et. al [29] applied optimization techniques to a D3Q19 lattice Boltzmann method implementation on GPUs using global memory. Using a structure-of-arrays (SoA) type of data organization, and propagation performed through global memory transactions, the GPU LBM kernel achieved almost 90% memory bandwidth usage. Two schemes with different collision operators were tested: LBGK (Lattice Bhatnagar-Gross-Krook) with a split (push scheme) and a reversed scheme of propagation (pull scheme) using MRT (Multiple Relaxation Time). Separate source and destination lattices were used in global memory to avoid overwrites, and they were alternated in each time step.

A lid-driven cavity was used for both physical validation and for performance measurements. Physical validation was performed on lines in the z-axis, with $x, y = L/2$ (L was the lattice size in one direction) and x-axis with $y = L/2$ and $z = L/2$. For a Reynolds number of 1000, LBGK code outcomes were in good accordance with the reference values and MRT implementation achieved almost perfect correspondence. Compared to the LBGK model, the more stable and accurate MRT, despite its higher computational cost, yields equivalent performance on GPUs. LBM was shown to be limited by global memory bandwidth on GPUs, achieving up to 86% of the maximum attainable bandwidth on a NVIDIA GTX 295 GPU. Performance achieved on both schemes and models was between 470 and 500 MLUPS for lattice sizes varying between 64^3 and 160^3 cells.

Habich et. al [17] established a model to estimate the upper bound on the performance of a lattice Boltzmann method implementation on GPUs. The implementation consisted in a D3Q19 lattice Boltzmann method with the BGK collision operator. An upper bound for performance was established using the vector triad from the STREAM benchmark [23] on a NVIDIA 8800GT and GTX 200 GPU. Domain lattice was stored as a structure of arrays (SoA) to allow global memory coalescing, and each CUDA thread was assigned one domain cell. In order to avoid misaligned writes to global memory, which would impose a big performance penalty, particle distributions were temporarily stored to shared memory and synchronized, before they could be written using coalesced, aligned stores.

Performance estimate was met for both GPU models using specific domain sizes. Also, to ensure coalescent access on domain sizes that were not divisible the warp size, a padding strategy was employed, and the 128-byte alignment was determined to be the best alignment size. Memory bandwidth was improved by 30% on the GT200, in comparison with the 8800GT. Even

though the performance of the GPU implementation was higher than that of the CPU version for single precision, the same was not true for double precision, especially considering the effort required to implement the CUDA version.

Rinaldi et. al [34] presented a lattice Boltzmann method based in a single-loop pull scheme using 19 discrete velocities in three dimensions. Validation and performance tests consisted in running a 3D lid-driven cavity simulation. Validation with Reynolds numbers of 100 and 1000 were conducted for the GPU implementation. Results were in conformance with the literature for velocities on the x and y components in the z-axis mid-plane. Global memory accesses were coalesced and reduced by storing values of x-axis adjacent in memory and using shared memory, respectively.

Performance of the GPU version was compared to a single thread CPU implementation, with grid accesses on the latter implementation optimized for cache-based architectures. However, CPU and other software specifications (such as compiler optimization level) were not provided. The GPU version was executed on a NVIDIA GTX 260 GPU. Both implementations utilized single precision floating-point numbers. Domain sizes between 16^3 and 160^3 cells were tested on both CPU and GPU, their performance in MLUPS measured and compared. GPU LB implementation was 92 to 130 times faster than its CPU equivalent and with some optimizations, it was possible to obtain 400 MLUPS performance, with 65% of effective memory bandwidth of the GPU utilized.

Extending the work on [16], Habich et. al [16] optimized the D3Q19 lattice Boltzmann method implementation on GPUs further, using CUDA and adapted the implementation to OpenCL as well. An upper bound on performance was established for three different device models, using the STREAM benchmark [23]. Two grids were used to eliminate data dependencies, and further optimizations were applied to reduce the number of registers utilized by the stream-collide kernel. Data was stored in a structure of arrays (SoA) layout, and usage of shared memory was shown not to be advantageous. Memory padding was shown to be particularly important on some GPU models, especially when ECC was enabled.

Utilization of ECC in the simulation showed a loss of 10% to 18% in performance. LBM kernels were able to sustain 83% of available memory bandwidth, in single precision and with ECC disabled. Those kernels also gave a speedup of two compared to a highly optimized CPU implementation in a full socket Intel Xeon server. The OpenCL version was on par with its CUDA equivalent in terms of performance on the GPUs studied, however more optimizations are required to execute the same implementation on the CPU, since the compiler did not utilize SIMD instructions when generating the code.

3.2 GPU Communication

In this section, a literature review of works focused on GPU communication and multi-GPU simulation approaches are presented.

Jacobsen et al. [20] implemented a incompressible fluid flow simulation using the Navier-Stokes equations along with a temperature equation to model buoyancy effects. Pressure equation was solved using a Jacobi iterative solver, and a projection algorithm was utilized to solve the discretized Navier-Stokes equations. Single precision was used in all computations. Implementation was validated using cubic lid-driven cavity and natural convection in heat cavity problems. Results agreed with benchmark data and other results reported in the literature.

The 3D cartesian staggered grid was decomposed into 1D layers, which eliminated the need to gather and scatter data in non-major dimensions of the grid, even though the size of data

to communicate increases faster on 1D decompositions as the number of nodes increase. Computations were overlapped with inter- and intra-node communication. Three implementations were tested: non-blocking MPI with no overlapping of computation, overlapping computation with MPI, and overlapping computation with MPI communication and GPU data transfers.

Performance and scalability tests were performed in the NCSA Lincoln cluster, where each node had two Intel quad-core processors operating at 2.33GHz, 16GB of RAM and two NVIDIA Tesla Tesla C1060 GPUs. Scalability experiments were performed with the lid-driven cavity problem with a Reynolds number of 1000. Overlapping implementations showed improved performance and parallel efficiency. When the workload per GPU was small, network latency dominated the simulations and the overlapping implementations could not hide communication overhead. Finally, weak scalability experiments were able to achieve 2.4 TFLOP/s on 128 GPUs.

Playne et al. [31] simulated a Cahn-Hilliard equation using a discretization obtained with a finite difference method, with second order accuracy. Cahn-Hilliard systems are used for simulations of phase separation in materials science. Domain field was decomposed into layers in the highest dimension (y in 2D, z in 3D), split into equal layers and spreaded across the GPUs in each node of the GPU cluster. Computation was overlapped with communication by computing boundary points and exchanging them asynchronously, while computing inner points of the domain field.

Two algorithms were proposed, which differed in their pattern of communication: bi-directional and one-directional communication. Intra-node performance was shown to scale almost linearly on a machine with four NVIDIA GTX480 GPUs for both communication schemes. Evaluation was performed on a cluster of 16 desktops with Intel Core 2 Quad Q9400 processors and NVIDIA GTX470 GPUs. Performance for the bi-directional algorithm showed unreliable performance and did not scale well. Uni-directional algorithm provided more reliable results and scalable performance. The uni-directional algorithm showed a speedup of almost 2x over a single GPU with two nodes, but such gains did not scale. In the three dimensional case, communication time outweighed the computational gain of more nodes.

Xian et al. [45] performed scalability and performance experiments with a D3Q19 lattice Boltzmann simulation, on the multi-node GPU cluster Tsubame. The computation of a lid-driven cavity flow problem in single precision with 96^3 lattice nodes using a single NVIDIA Geforce GTX280 was tested and a performance of 270 MLUPS was measured, which was 87 times faster than its CPU equivalent. Further details of the CPU implementation were not provided, apart from the processor model.

Data communication process was carried out by staging memory copies through the host. Distinct domain partitions were also utilized, because data size for communication greatly decreases with the increase in the number of ranks (GPUs) for 2D and 3D partitionings. Using 1D partitioning, the performance does not scale with a higher number GPUs, which was not the case for 2D and 3D decompositions. With the same partitioning, performance for 384^3 was half than that of 768^3, because of higher GPU load. Communication time decreased for 2D and 3D partitioning and increased for the 1D case. Overlap of computation and communication was achieved by using a CUDA stream for inner point computation, and another for boundary computations and communication. It was verified that using overlapping mode, performance increased by at least 8% (on 96 GPUs). Also, the computational time was smaller in the overlapping mode.

MVAPICH2-GPU design was proposed by Wang et al. [44] to support efficient GPU to GPU communication using MPI transparently. Through such design, it is possible to pipeline GPU memory copies with RDMA data transfers inside MPI. The proposed approach was shown to improve the performance even when compared with an optimized pipelined implementation at

the user application level, thus eliminating the need for the use of such technique, since the MPI library itself is more efficient at pipelining.

Evaluation of the proposed design was performed in two clusters, one with GPUDirect RDMA and the other without. The first cluster was composed of 8 nodes with Tesla C2050 GPUs and Mellanox QDR interconnect. As for the second cluster, it consisted in 256 nodes, with two GPUs per node and Infiniband QDR interconnects. All experiments were conducted with the OSU microbenchmarks for one-sided, point-to-point and collective communication operations. One process was run per node and one GPU was used per process for all experiments. When GPUDirect was present, an improvement of up to 45% was achieved in point-to-point latency for a message size of 4MB. The same improvement was also observed with one-sided communication. Improvements of up to 37% were also achieved with collective operations.

Wang et al. [43] implemented a new design for enabling high-performance communication support between GPUs for non-contiguous datatypes. Through the MPI datatype concept, (un)packing data was made possible between GPUs, utilizing a pipelined approach to improve communication performance and overlap between data copies and network transfers. Part of (un)packing datatype processing was offloaded to the GPU, which was in contrast with CPU based approaches. Evaluation of the proposed design was shown to improve the latency of communication up to 88%, for 4MB vector data messages. Data movement was performed using a packing operation from non-contiguous to contiguous data inside the GPU, then asynchronous copies of the GPU data to the CPU were issued in chunks and communicated. Packing and unpacking latency of non-contiguous data was shown to dominate the pipeline performance.

Experiments were carried out on cluster with eight nodes. Each node was equipped with dual Intel Xeon Westmere CPUs operating at 2.53GHz, 12GB RAM, NVIDIA Tesla C2050 GPUs and Mellanox Infiniband QDR interconnect. Three different designs were evaluated: a blocking one, both on MPI and memory copies, a non-blocking design with pipelining at the application level, and the proposed solution. Non-blocking and proposed solution designs had similar performance, since their principles of implementation were the same, however the proposed solution was integrated into the MPI library, which reduces code complexity for applications. A 2D stencil benchmark was also performed, with improvements from 20% to 40% were reported for different process grid configurations.

Potluri et al. [33] introduced a design for intra-node MPI communication on multi-GPU nodes, taking advantage of IPC capabilities provided in CUDA. Using IPC memory handles, processes can map memory regions belonging to other processes in the same node. Since IPC handle creation and destruction generate severe overhead, a cache was designed to alleviate the cost of handle management. CUDA IPC allows for peer-to-peer communication between GPUs, bypassing host memory, as long as the devices are in the same bus.

A node with a 12-core Intel Westmere node, with 24GB of RAM and two NVIDIA Tesla C2075 GPUs was utilized in the experiments. All experiments were run using two processes, where each process was mapped onto a different socket and each used a different GPU device. Experimental evaluation have shown that the new designs improved two-sided GPU-to-GPU communication latency by up to 79% and can quadruple the bandwidth, for 4 MB messages. One-sided communication latency was also improved by 74% improvement. Using multi-GPU lattice Boltzmann application, a 16% improvement in time steps was verified for 512x512x64 grid, using either one or two-sided communication.

Bernaschi et al [5] presented development techniques of multi-GPU codes for mesh based simulations. By using two different streams for communication and computation, communication latency could be completely hidden by computations, provided that the time for computations was long enough. Two different communication strategies were evaluated: naive

and overlap. The naive communication scheme consisted in performing the communication, boundary and computation steps serially, on the same CUDA stream. Overlap communication scheme used two CUDA streams concurrently, with one stream for communication, boundary handling and outer point computations, and the other for bulk (inner domain point) computations. Two MPI implementations were tested, in order to compare their performance and GPU-aware capabilities.

Tests were conducted on two different clusters with similar platforms and processors. The first machine consisted in a single node with 8 NVIDIA Tesla C2050 GPUs connected to 2 PCIe buses (4 GPUs per bus), and the second one consisted in three nodes with 2 NVIDIA Tesla C2070 GPUs and QDR Infiniband interconnect among nodes. Parallel efficiency and GFLUPS performance were measured for a 3D Heisenberg Spin Glass and the Himeno benchmark. Parallel efficiency for the overlap version, with or without CUDA-aware MPI support, increased from 70% to almost 99% using 4 GPUs. GPU-aware MPI implementations seemed to interfere with CUDA streams in a way that prevented computation and communication to overlap efficiently.

A hybrid parallel GPU implementation of the lattice Boltzmann method was presented by Xiong et al. [46], making use of the iD2Q9 stencil. In order to improve performance, communication was executed asynchronously with computations, through the use of CUDA streams. Boundary handling and communication were carried out in one stream and computation of inner points in another stream. MPI was utilized to extend the code for multiple GPUs in separate nodes and OpenMP to improve performance of intra-node communication. In this work, it was not stated- which MPI implementation or version was utilized. Data transfers for communication were performed by staging data through host memory.

Results were obtained for a cluster of 362 nodes, with each node containing two quad-core CPUs, six NVIDIA Tesla C2050 GPUs and QDR Infiniband interconnect. Validation of the implementation was obtained through the analysis of a two-dimensional Couette flow, which was utilized to evaluate the accuracy of the GPU implementation. The domain size chosen was 2048x2048 and Reynolds number of 400. Simulation was run in parallel on four GPUs. Simulated result was found to be in agreement with the analytical solution.

Performance measurements were taken by running the same Couette flow simulation on five different scenarios, with varying domain sizes. Computation domain was split in either one or two dimensions. All test cases were run 10 times with 10000 iteration steps on each run, and wall-clock times were recorded after arithmetical averaging. Most of the data transfer and communication times were successfully hidden by overlapping communication and computation, especially for bigger domain sizes. Varying the number of GPUs from 1 to 6 in the same node was shown to deteriorate the performance, since the GPUs shared the PCIe bus, the bandwidth became disputed.

Bernaschi et al. [4] evaluated the performance of two major CUDA-aware MPI implementations: OpenMPI and MVAPICH2. Alternatively, a 3D torus high-performance interconnect for clusters, APENet+, was utilized and compared with both MPI implementations, since it was one of the first adapters to allow RDMA transfers and direct access to global GPU memory. Using a 3D Heisenberg spin model with periodic boundaries in x, y and z directions as benchmark, time measurements were taken in picoseconds, corresponding to the time required to update a single spin of the model. In all tests, single precision floating-point was utilized. APENet+ performance was verified to be inferior to Infiniband.

Model computations on the domain were split in red and blue points, in a checkerboard fashion. Domain decomposition was performed in a single axis, since the goal was to compare different techniques of data exchange between GPUs. Computation and communication were overlapped through CUDA streams, one stream was used for the computation of red and

blue inner domain points, and another stream to compute red and blue boundary points and communication.

Obrecht et al. [28] Multi-GPU implementation of the D3Q19 lattice Boltzmann method using a multi relaxation time (MRT) collision operator. Intra-node parallelization was achieved through the use POSIX threads, with global synchronization between sub-domains performed using thread synchronization constructs at each time step. One thread was assigned to each GPU, in order to hold its corresponding CUDA context. Inter-GPU communication was made using page-locked GPU memory and zero-copy memory transactions.

Global lattice was split in regular cuboids along the direction corresponding to the major dimension, for simplicity. Four buffers were assigned to each interface between sub-domains: two for incoming data and two for outgoing. Validation of the code was performed using the well-known lid-driven cavity for cubic domains. Scalability tests were made on a 192x192x192 lattice which could be handled by one up to six GPUs. Also, performance tests were realized on a 384x384x384 lattice using three, four and six computing devices. Measurements were reported using the MLUPS metric, on NVIDIA Tesla C1060 GPUs on a Tyan B7015. Performance was in the same order than the one obtained with optimized double precision code on supercomputers.

Scalability was optimal with no less than 90% parallel efficiency and performance for single precision using one GPU was reported to be 387 MLUPS on a 192x192x192 lattice, achieving 80% of the peak memory bandwidth. Simulations were run for up to six GPUs, which allowed for execution of bigger lattices, 480x480x480 and 384x384x384 cells for single and double precision, respectively. These bigger lattices, in turn, allowed for simulations with higher Reynolds numbers, up to 30.000. Running the LBM solver for Reynolds number of 30.000, the flow had shown to lose symmetry at an early stage of the simulation. To verify if the issued persisted, different precisions were evaluated: single, double and mixed precisions. Mixed precision had shown no improvement over single precision and the performance trade-off has been shown not to be worthy. Symmetry loss was discovered to be related to accumulation of round-off errors.

Potluri et al. [32] proposed and evaluated three novel designs to improve inter-node GPU-to-GPU MPI communication in MVAPICH2. Through the use of GPUDirect RDMA over Infiniband, extra memory copies from device to host and vice-versa can be completely eliminated in inter-node GPU-to-GPU communication, thus reducing communication overhead.

In the first design, MV2-GDR, the rendezvous protocol was utilized for all message transfers, to circumvent limitations of the eager protocol for small messages. While using GDR with the rendezvous protocol gives a low latency path for small message transfers from GPU memory, it suffered with limited read bandwidth, due to chipset limitations of Intel Sandy Bridge architectures. In the second design, MV2-GDR-H, the bandwidth limitation arising from P2P transfers in the underlying chipset was avoided by staging large messages through the host, while maintaining the rendezvous protocol for small messages, as in the first design. In order to circumvent hardware limitations, i.e. bandwidth for PCI-e transfers in Sandy Bridge architectures, the MV2-GDR-H-Advanced design was proposed. It consisted in staging the send buffer through the host and using RDMA on the receiving process.

Proposed hybrid (MV2-GDR-H-Advanced) solution achieved 69% and 32% improvement for 4Byte and 128KByte messages using MPI Send/Recv operations, on a Sandy Bridge platform with NVIDIA Kepler K20 GPUs and Mellanox IB FDR adapters. Similar improvements were also observed for bi-directional bandwidth. In collective operations, latency of 4Byte and 1MByte messages was reduced up to 53% and 48%, respectively. Tests were also conducted with two applications that rely on nearest neighbor communication: AWP-ODC and

GPULBM. Improvements in overall application execution time (time to solution) for AWP-ODC and GPULBM were of 30% and 35%, respectively.

Sourouri et al. [38] proposed a intra-node communication scheme for multiple GPUs sharing the same PCIe bus. State-of-the-art approaches had used one thread or process per GPU, in order to eliminate kernel launch and synchronization overheads. A pair of streams was created, one for inner domain point computations and another for boundary point computations and communication. However, the proposed scheme explored further concurrency by using multiple threads and CUDA streams. Multiple OpenMP threads were used per GPU, one pair for send and receive operations of each neighboring subdomain and an assistant thread to coordinate groups of CUDA streams. Using multiple threads reduced kernel launch overhead, avoided stalling the main host thread and improved application performance by reducing the gap between computation and communication.

A group of CUDA streams was also utilized in a per-thread basis, with each group of streams assigned to an assistant thread. A CUDA stream group consisted in one pair of streams for sending and receiving, and a one stream for computation of inner domain points. Multiple streams were used to stack communication such that data exchange could occur simultaneously on both sides of a subdomain. Usage of multiple streams allowed for a more fine-grained control of different operations.

Experimental evaluation was performed in three different systems with different GPU devices: NVIDIA Tesla K20, Tesla C2050 and a GTX 590. The test systems also differed in the processor and PCIe bus topologies and amount of memory. ECC was disabled for all devices, since the GTX 590 did not supported it. Two experiments were conducted, the first measured bandwidth for the proposed solution with and without P2P transfers, comparing it to a baseline that employed the state-of-the-art at the time. Results of the first experiment have shown that the proposed scheme outperformed the state-of-the art by 1.4, 1.6x and 1.85x in the GTX 590, C2050 and K20 platforms, respectively. The second experiment consisted in executing a 3D 7-point stencil application based on the discretization of the Laplace equation. In that scenario, the experiments have shown that the communication latency could be effectively hidden and the proposed scheme was 58% and 40% faster for the largest problem sizes on the Tesla C2050 and GTX 590, respectively.

Chu et al. [8] implemented two novel designs for CUDA-aware MPI libraries, which improved the performance of non-contiguous data processing and movement, and explored further concurrency between communication and computation within the GPU and between CPU and GPU. Both designs take advantage of modern CUDA features, such as Hyper-Q, multiple streams and event abstractions.

The first proposed design consisted in using CUDA event abstractions for verifying operation completion and communication. Using custom MPI datatypes, (un)packing operations are started in different CUDA streams when *MPI_Isend* or *MPI_Irecv* is called, and an event is recorded after each operation is issued, using the same stream for packing and recording. Once the process calls *MPI_Wait* and enters the communication progress engine, the CUDA events previously recorded are queried, then the first one to complete is selected, since it indicates that the (un)packing operations has completed, and the communication is initiated for the matching request and polled for completion. To avoid deadlocks, all events are queried, and the first one to complete is selected by the engine. This design was recommended for GPU-only workloads.

Second design consisted in using CUDA stream callbacks, in order to achieve maximum overlap for mixed CPU-GPU (hybrid) workloads. It also utilized MPI datatypes and MPI_Isend or MPI_Irecv for (un)packing. When one of these MPI functions is called, a (un)pack operation is issued on a separate stream, a completion callback is associated with it and the matching

request information is set. After the (un)packing operation completes, a thread is dynamically spawned by the CUDA runtime for processing the callback. CUDA runtime does not allow for GPU operations to be issued inside the callback function and, since MPI often requires the use *cudaMemcpy* for data transfers from(to) the GPU, a separate helper thread is spawned inside the callback function for these kinds of transfers. This scheme allows for complete separation of communication progress and computations in the main MPI thread, while allowing CUDA context sharing. In the context of this work, only one helper thread is spawned per process, in order to reduce CPU context switching overhead and reduce the number of shared locks inside the MPI runtime.

Proposed solutions were tested in two different clusters for intra-node and a mix of inter- and intra-node communication between GPUs using benchmark applications that require distinct communication patterns. For intra-node inter-GPU ping-pong experiments using DDTBench benchmarks, the event-based design achieved up to 54%, 67% and 61% performance improvement on three distinct finite element benchmark applications. Callback-based design was not as efficient as the event-based one, because the benchmarks consisted mainly in GPU-only workloads. The proposed designs also achieved 33% improvement on total execution time of a halo-exchange based benchmark application using up to 32 GPUs on the Wilkes cluster.

3.3 HPC Framework Architecture

Godenschwager et al. [14] described an update to the previous implementation of waLBerla, with further optimizations added to improve LBM kernels' single node performance. It also adapts the kernel implementations to utilize features of current hardware architectures, which mainly consists in using SIMD vectorization and multiple cores to saturate main memory bandwidth, since Lattice Boltzmann methods are memory bound. Furthermore, the domain decomposition was rewritten, such that it requires a constant amount of memory per process, regardless of the total number of processes and the shape of the domain. Domain setup and initialization was also improved using a binary format and other optimizations, such that the simulation domain setup require less disk and memory space, even with domains with millions of cells.

A model for single node performance was established for both SuperMUC and JUQUEEN supercomputers, using the roofline model. Implementation started with a generic stream-pull kernel, then a kernel written specifically for the D3Q19 stencil and, finally a kernel that utilized SIMD vectorization and a structure of arrays (SoA) memory layout, which performed the LBM update on a by-direction manner, to circumvent prefetcher limitations with multiple concurrent load/store streams.

To test the scalability of the framework, weak scaling experiments were performed using the fastest TRT kernel in two simple scenarios: the lid-driven cavity problem and channel flow around a fixed obstacle with an obstacle to fluid ratio of less than 1%. In weak scaling experiments, the parallel efficiency of the framework was only limited by the communication network of SuperMUC. A parallel efficiency of 92% was achieved using the whole JUQUEEN machine. Complex vascular geometry processing required some modification to the way lattice cells were traversed, being a scheme similar to the one employed for sparse matrices, the one chosen to allow for SIMD vectorization. In all tests, the performance measurements agreed well with the predictions obtained from performance models (roofline and ECM). waLBerla reached up to 6638 LBM time steps per second in strong scaling settings, with a resolution of a trillion cells, that could be used for practical simulations.

Recently, a Python extension [3] was coupled to the waLBerla C++ multi-physics simulation framework. While performance critical parts were kept in C++, higher level functionality, like domain setup, simulation control, and evaluation results are possible using Python. Advantages of the proposed extension were two-fold: ease the burden of using the C++ language, which domain experts might not have familiarity or experience with, and increase the flexibility for the setup, control and analysis of simulations.

Chapter 4

A Scalable GPU Communication Framework for Stencil Applications

In this chapter, solutions to hide communication latency and improve performance in the waLBerla framework are presented. State-of-the-art GPU communication solution for waLBerla does not explore concurrency, therefore a potential increase in performance can be obtained through concurrency mechanisms, potentially hiding all the communication latency. Furthermore, advancements in MPI implementations, such as support for GPU memory buffers are utilized in a novel GPU communication scheme.

4.1 Architecture Overview

Proposed solutions aim to hide communication latency, by exploring CUDA concurrency mechanisms, and the use of inner and outer computation kernels. Through the use of advanced programming techniques and best practices in software engineering, the proposed implementations maintain flexibility and usability while maintaining performance.

Extensions to the waLBerla framework were implemented, either as new classes or modifications and capabilities to existing functionalities. The *GPUPackInfo* communication class is extended to support *zyxf* memory layout (array-of-structures) and CUDA streams. A new class, *GPUBufferedScheme* is introduced to allow communication using GPU buffers, since waLBerla supported only host memory MPI buffers. GPU buffers are supported by most modern MPI implementations that offer CUDA-aware support, and several optimizations exist. An interface to invoke inner and outer kernels was also developed.

The *Kernel* class was modified to support CUDA streams, and the boundary handling and computational kernels of the lattice Boltzmann simulation were also modified to support and use streams, in order to utilize maximum concurrency within the application.

Details about these and other proposed solutions are described in the next sections of this chapter. Current communication architecture is presented in Section 4.2.1. Extensions to the current GPU communication architecture are presented in Section 4.2.2. Time loop setup is described in Section 4.3. Configurable memory alignment for GPU fields is described in Section 4.4.

4.2 GPU Communication Engineering

The state-of-the-art scheme for GPU communication in waLBerla, *GPUPackInfo*, improved considerably overall GPU communication efficiency with respect to previous existent solutions [10]. However, it did not make any use of concurrency mechanisms, such as CUDA streams, which were hence implemented.

In this thesis, the implementation of *GPUPackInfo* that makes use of concurrency mechanisms is also referred to as *asynchronous GPUPackInfo*. A reduction in memory copy overhead was also possible, by reducing the number of calls to the CUDA runtime API (*cudaMemcpy3D*), through the use of packing and unpacking kernels.

In order to reduce the communication overhead further, it is also possible to use GPU memory buffers, since modern MPI implementations can use them to perform communication directly, making use advanced CUDA features, such as GPUDirect RDMA when it is available. Since most of waLBerla communication algorithms rely on CPU buffers, a novel communication infrastructure is introduced that makes use of GPU buffers.

This section presents solutions to improve GPU communication performance and hide communication latency in waLBerla, including design and implementation details.

4.2.1 Base Communication Architecture

Extending on the work of Carvalho *et. al*, the *GPUPackInfo* data packing and unpacking mechanism is augmented. Support for CUDA streams is introduced, so that asynchronous memory copies in all stencil directions are possible. Array-of-structures (zxyf) memory layout is also implemented and tested. Finally, an extra option is added to the packing and unpacking mechanisms, data can be packed and unpacked with single calls to specialized CUDA kernels, instead of relying on *cudaMemcpy3DAsync*.

These two options for packing and unpacking data are available to the users of *GPUPackInfo* through the boolean flag *usePackKernels* in *GPUPackInfo*'s constructor. The default option is to use packing and unpacking kernels, however users can also choose to use *cudaMemcpy3DAsync* instead. Since *GPUPackInfo* relies on host memory allocated MPI buffers, a memory copy is required to transfer data between host and device memories in packing and unpacking operations.

A new communication scheme is also implemented, which makes use of buffers allocated on the GPU memory. The scheme is contained in a new class, *GPUBufferedScheme*, and it implements the same communication algorithm as *UniformBufferedScheme*. All the buffers passed in *GPUBufferedScheme* reside on the GPU, and are implemented in the *GpuSendBuffer* and *GpuRecvBuffer* classes, for MPI send and receive buffers, respectively.

In order to contextualize how the communication mechanism works, it is important to understand at a high-level, how the *UniformBufferedScheme* class works. Algorithm 1 demonstrates a high-level representation of how the *UniformBufferedScheme::startCommunication()* method works, and its interactions with the MPI buffers. This method implements most of the communication algorithm.

4.2.2 Communication Architecture Extensions

Concurrency in CUDA is achieved through the use of streams. Streams can be used throughout the life of the application, either in the framework internal routines or when the user calls a function or kernel. Therefore, a new singleton class is introduced, *GPUManager*, which

Algorithm 1 Pseudo-code for UniformBufferedScheme communication algorithm [10]

```
uniformPackInfo : communication::UniformPackInfo
sendBuffer : mpi::SendBuffer
recvBuffer : mpi::RecvBuffer
for block ∈ structuredBlockForest do
   for dir ∈ Stencil do
      neighbor ← block.getNeighbor(dir)
      if block and neighbor are owned by the same process then
         uniformPackInfo.communicateLocal(block, neighbor)
      else
         uniformPackInfo.packData(block, dir, sendBuffer)
         mpiExchangeBuffers() // perform MPI communication between neighbors
         uniformPackInfo.unpackData(block, dir, recvBuffer)
      end if
   end for
end for
```

initializes and manages CUDA streams for the entire framework. Stream initialization is tied with the application stencil, such that one stream is provided for each direction of the stencil. This scheme allows the use of concurrent streams to pack and unpack data in different directions, as well as to issue concurrent outer kernels, one for each stencil direction.

In order to explore CUDA concurrency mechanisms and enable communication latency hiding in the simulation, *GPUPackInfo* class is extended to utilize streams and asynchronous operations, specifically *cudaMemcpy3DAsync* and specialized kernels for packing and unpacking. Implementation of packing and unpacking kernels is presented in Listing 4.1.

Packing and unpacking kernels are set to be used by default in *GPUPackInfo*. In this case, data is (un)packed (from)into the buffer using a single kernel call, regardless of direction. The kernel is executed in the stream that corresponds to the direction of the field being packed. In order to perform the operation, a staging buffer is allocated in GPU memory, so that when the kernel completes, only a single memory copy is required from GPU to host to insert the data into the buffer, and vice-versa in the receive case.

Listing 4.1: GPU packing and unpacking kernels

```
template< typename T > __device__ __forceinline__
T & getBuffer( unsigned char * __restrict__ bufferBasePtr, size_t index )
{
   return *(T*)( bufferBasePtr + index * sizeof(T) );
}

template< typename T > __device__ __forceinline__
const T & getBuffer( const unsigned char * __restrict__ bufferBasePtr,
   size_t index )
{
   return *(T*)( bufferBasePtr + index * sizeof(T) );
}

template< typename T > __global__
void packDataKernel( cuda::FieldAccessor<T> src,
   unsigned char * __restrict__ buf )
{
   src.set(blockIdx, threadIdx);
```

```
19    const uint_t bufferIndex = src.getLinearIndex(blockIdx, threadIdx,
20      gridDim, blockDim);
21
22    if ( src.isValidPosition() )
23    {
24      getBuffer<T>(buf, bufferIndex) = src.get();
25    }
26  }
27
28  template< typename T > __global__
29  void unpackDataKernel(cuda::FieldAccessor<T> dst,
30    unsigned char * __restrict__ buf)
31  {
32    dst.set(blockIdx, threadIdx);
33    const uint_t bufferIndex = dst.getLinearIndex(blockIdx, threadIdx,
34      gridDim, blockDim);
35    if ( dst.isValidPosition() )
36    {
37      dst.get() = getBuffer<T>( buf, bufferIndex );
38    }
39  }
40
41  template< typename T > __global__
42  void localCommunicationKernel(cuda::FieldAccessor<T> dst,
43  cuda::FieldAccessor<T> src)
44  {
45    dst.set(blockIdx, threadIdx);
46    src.set(blockIdx, threadIdx);
47    // Copy data directly from source to destination field
48    dst.get() = src.get();
49  }
```

Support for GPU memory buffers is introduced in *GPUBufferedScheme*. It utilizes the same communication algorithm as *UniformBufferedScheme* and implements the same interface, the only difference is the type of buffers involved. *GpuSendBuffer* and *GpuRecvBuffer* classes implement the GPU memory buffers, which can be supplied directly to MPI implementations that support CUDA pointers, also known as CUDA-aware MPI implementations. These buffer implementations follow the same interface as their CPU counterparts, *mpi::SendBuffer* and *mpi::RecvBuffer*.

4.3 Timeloop Setup

Hiding communication latency requires overlapping computation with communication. Communication in Lattice Boltzmann method simulations require the exchange of boundary points from blocks, however the boundary points must be updated prior to the communication step. Usually, the domain points are updated all at once, in a single kernel. To eliminate this dependency, the LBM kernel is split into inner and outer kernels, with the inner kernel computing the inner block points and outer kernel computing the boundary block points.

Separating the LBM kernel into inner and outer kernels allows the overlap of computation and communication, since the communication does not have to wait for the computation of all domain points to complete. Timeloop setup in waLBerla has to be changed, to carry out these new operations, instead of the classic LBM setup. Figure 4.1 depicts how the timeloop is executed to overlap computation of inner points and communication. Since these operations

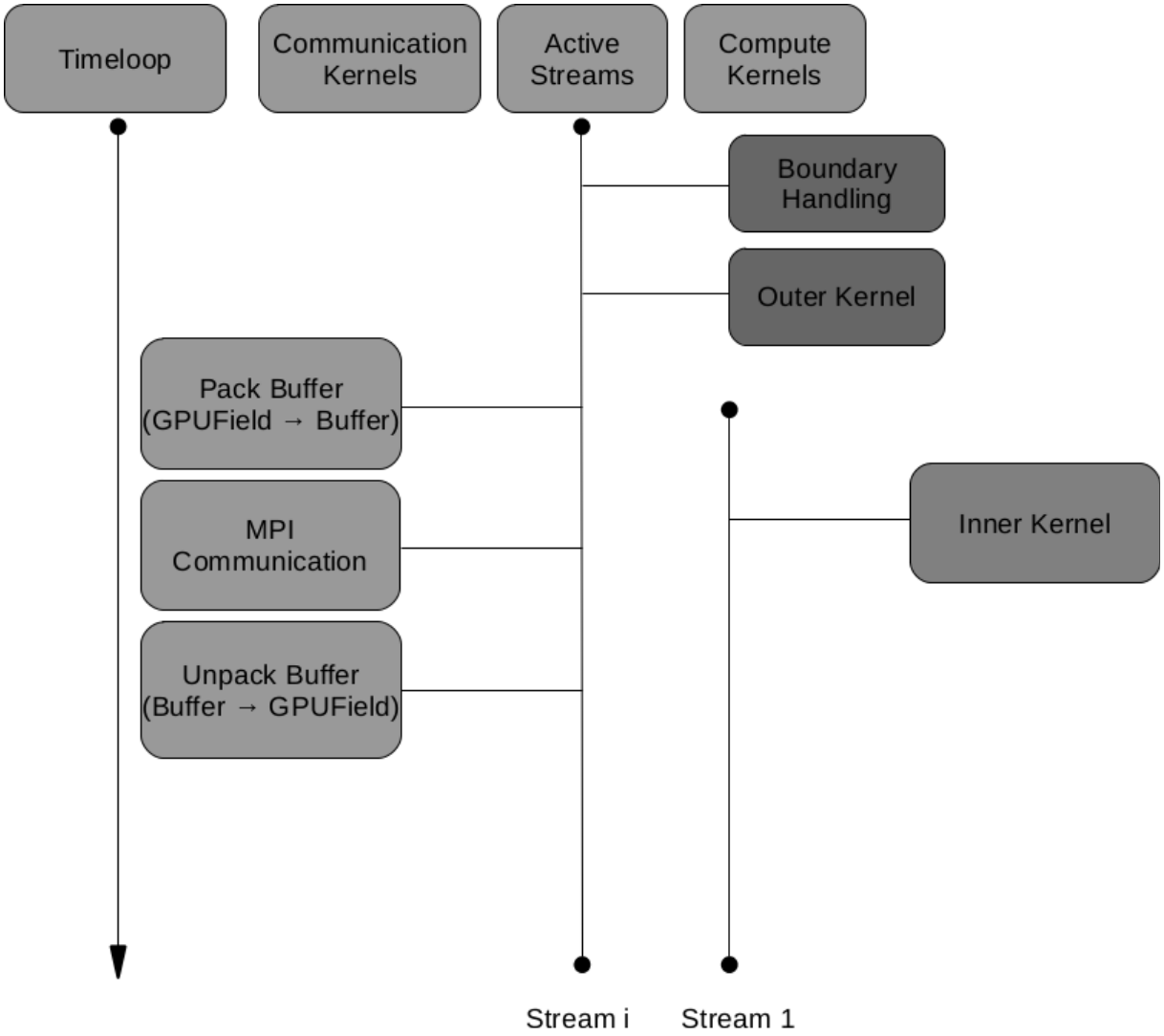

Figure 4.1: Timeloop setup with inner and outer kernels. Outer kernel compute boundary points of the block, while inner kernel computes inner points of the block. One CUDA stream is used for the inner kernel, and a stream is used for each neighboring process, which is represented by Stream *i*

are performed in separate CUDA streams, they are executed concurrently. A synchronization operation on all streams is performed at the end of the timeloop, to ensure all operations have completed before starting the next time step.

4.4 Configurable GPU Field Memory Alignment

Configurable memory alignment for GPU fields is introduced in the *GPUField* class. Through handcrafted memory allocation, different alignments are supported, and *cudaMalloc3D* is no longer necessary. Padding in the innermost dimension is utilized in the *GPUField* class constructor, by checking if the innermost dimension is aligned to the desired alignment.

In order to enable the custom alignment, the *usePitchedMem* flag must be set to true in the constructor. Listing 4.2 demonstrates how the allocation works.

Listing 4.2: GPUField custom alignment allocation

```
cudaExtent extent;
if ( layout_ == zyxf )
{
   extent.width = (_fSize ) * sizeof(T);
   if ( usePitchedMem_ )
   {
      // extent.width is already in bytes
      const size_t nrOfMisalignedBytes = extent.width % alignmentSize;
      const size_t nrOfPadBytes = alignmentSize - nrOfMisalignedBytes;
      extent.width = extent.width +
         (nrOfMisalignedBytes > 0 ? nrOfPadBytes : 0);
   }
   extent.height = (_xSize + 2 * _nrOfGhostLayers );
   extent.depth = (_ySize + 2 * _nrOfGhostLayers ) *
      ( _zSize + 2 * _nrOfGhostLayers );
}
else
{
   extent.width = (_xSize + 2 * _nrOfGhostLayers ) * sizeof(T);
   if ( usePitchedMem_ )
   {
      // extent.width is already in bytes
      const size_t nrOfMisalignedBytes = extent.width % alignmentSize;
      const size_t nrOfPadBytes = alignmentSize - nrOfMisalignedBytes;
      extent.width = extent.width +
         (nrOfMisalignedBytes > 0 ? nrOfPadBytes : 0);
   }
   extent.height = (_ySize + 2 * _nrOfGhostLayers );
   extent.depth = (_zSize + 2 * _nrOfGhostLayers ) * _fSize;
}

pitchedPtr_ = make_cudaPitchedPtr( NULL, extent.width, extent.width,
   extent.height );
cudaMalloc( &pitchedPtr_.ptr, extent.width * extent.height * extent.depth );
```

Chapter 5

Materials and Methods

This chapter describes hardware, software, methods and goals to demonstrate the validity and performance of the proposed solutions. Validation experiments are performed to verify that the solutions produce the expected outputs. Performance experiments are performed to evaluate the performance of proposed solutions in worst case scenarios, as well as their scalability. Proposed solutions are compared with the state-of-the-art communication solution for waLBerla [10].

5.1 Hardware and Software

Two different systems of the C3HPC cluster [6] were utilized to run the validation and performance tests. These systems share the same CPU, memory and network adapter specifications, since they are all part of the C3HPC cluster. GPU specifications for the nodes are presented in Table 5.1. Other relevant specifications are presented in the following list:

- CPU: Intel Xeon CPU E5-4627 v2 Ivy Bridge EP @ 2.6 GHz (with max clock rate of 3.3 GHz)
- Cores: 8
- Sockets: 4
- RAM per node: 256 GB DDR3
- PCIe: v3.0 x16
- Network controller: Mellanox Technologies MT27500 Family [ConnectX-3]
- OS: Linux 4.4.35
- Compiler: g++ v4.9.2, nvcc 7.5.17
- MPI: OpenMPI 1.10.2, with CUDA support
- Location: DINF, Federal University of Paraná

GPU characteristics for each type of node are described in table 5.1.

	Kepler	**Fermi**
Number of nodes	2	1
GPUs per node	2	2
GPU	NVIDIA Tesla K40m	NVIDIA Tesla C2075
CUDA driver version	7.5	7.5
CUDA compute capability	3.5	2.0
CUDA cores	2880	448
Core clock	745 MHz	1150 MHz
Total memory	11520 MB	5375 MB
Memory clock	3.0 GHz	1.5 GHz
Memory bandwidth	288 GB/s	144 GB/s
ECC (error-correcting code)	Enabled	Enabled
NVCC flags	-std=c++11 -O3 -arch=sm_35	-std=c++11 -O3 -arch=sm_20

Table 5.1: Hardware and software characteristics for GPU nodes with Kepler and Fermi devices.

Currently, the CUDA toolkit requires a compute capability of at least 2.0, since support for previous architectures will be discontinued in the next release. However, the compute capability 3.5 is recommended, specifically for nodes equipped with NVIDIA Tesla K40m GPUs. CUDA version 7 or newer is required to compile the proposed solutions [10], since they make extensive use of C++11 template features, and so does the waLBerla framework.

All solutions were developed based on commit *b96c6485-2016-07-14* of the cuda-comm branch, which was based on a commit from the topic/cuda branch [10], from the official waLBerla repository. Experiments performed in this work were performed using the *Release* mode from waLBerla's build system, which enables the highest levels of compiler optimizations (*e.g.* -O3 flag). Support for CUDA and Python extensions were enabled on the build system, *i.e.* WALBERLA_BUILD_WITH_CUDA and WALBERLA_BUILD_WITH_PYTHON.

5.2 Validation

In this section, the experiments to validate the proposed solutions are presented. Simulation setups are similar to the ones presented by Carvalho *et. al* [10]. A Lid-driven cavity simulation is utilized with changing parameters to verify the validity and performance of the proposed solutions. Results are compared with the ones obtained by Carvalho *et. al* [10], for the same simulation and scenarios.

To verify the correctness of the simulation, the velocity profiles for the x and z components are obtained for the y mid-plane. Then, these values are compared with reference values documented in the literature [10, 34]. Streamlines are also obtained to verify if a vortex is formed around an axis in the y direction.

Validation tests are performed with a 3-D lid-driven cavity simulation, D3Q19 stencil lattice Boltzmann implementation with SRT/BGK collision operator, using 10000 iterations, contiguous (linear) memory, structure-of-arrays memory layout (fzyx layout), ECC enabled and *cuda::FieldIndexing* as the indexing scheme. There is an exception, though, the validation with the reference values required more iterations to converge to the reference values, and 20000 iterations were used to guarantee the convergence in the Fermi node. These extra number of iterations are attributed to an issue with the Lattice Boltzmann method simulation, particularly in the inner and outer kernel setup. Since there was not enough time to find the exact cause, the investigation of the cause of the issue is suggested as future work.

Two communication schemes were tested: GPUBufferedScheme (which uses GPU buffers), and GPUPackInfo with CUDA streams (asynchronous). In the list below, the validation tests are presented:

1. **Domain decomposition:** Validates the implementation of intra-GPU communication for both *GPUBufferedScheme* and asynchronous *GPUPackInfo*, the implementation of the local communication kernel, and *GPUManager*'s stream management. Simulation is setup with 128^3 cells, with the top velocity vector $v = (0.1, 0, 0)$, and $\omega = 1.4$.

2. **Multiple devices per node:** Validates the implementation of intra-node communication in a single node. The implementation of packing and unpacking kernels of both *GPUBufferedScheme* and asynchronous *GPUPackInfo* are tested. In the case of *GPUBufferedScheme*, the implementations of both *GpuSendBuffer*, *GpuRecvBuffer* and *GPUBufferPackInfo* are also tested. Particularly, *GpuSendBuffer::forward()* and *GpuRecvBuffer::skip()* methods are utilized in *GPUBufferPackInfo*. No Infiniband is tested in this test case. Simulation is setup with 4^3 blocks of 32^3 cells running on 2^3 processes. Blocks are processed on 2 devices of a single node.

3. **Multiple nodes:** Validates the implementation of inter-node communication, which includes MPI and Infiniband. Simulation setup consists of 4^3 blocks of 32^3 cells running on 2^3 processes, over 2 nodes with Kepler GPUs (*i.e.* 4 GPUs). Two processes are allocated per GPU. Processes with odd ranks are mapped to the first node, and the ones with even ranks on second node.

5.3 Performance

In this section, the experiments to evaluate the performance and scalability of the proposed solutions are presented. Performance is evaluated with respect to the overall communication time, communication direction imbalance and scalability.

A 3D lid-driven cavity simulation using the D3Q19 stencil is utilized in all experiments. All test cases utilize double precision floating-point, contiguous memory with structure-of-arrays (fzyx) layout. *FieldIndexing* is used to setup all CUDA kernels, and the device error-correction code (ECC) is always turned on.

Performance is measured based on the wall-clock time of the simulation loop, discarding the time spent in initialization and termination. Simulation is executed for 200 iterations, and repeated 10 times within the same process to mitigate warm-up interference – e.g. JIT compilation in the driver [24]. Only the smallest wall-clock time is regarded. All measurements are based on the mega lattice updates per second (MLUP/s).

5.3.1 Communication Performance

In this section, an experiment is presented to verify the communication performance of the proposed communication solutions for waLBerla, compared with the previously implemented state-of-the art communication scheme, *GPUPackInfo* [10].

Overall communication times vary between different uniform domain decompositions. Variations in these times occur because blocks have different amounts of data to communicate with a varying number of neighboring processes. In the case of the D3Q19 stencil, on a non-periodic domain, blocks that are internal to the decomposition always have to communicate

with 18 neighbors (worst case scenario), while blocks on the boundaries can have from 6 to 13 neighbors.

Communication in all directions dominates the overall simulation time. Therefore, a decomposition that explores this scenario would be of interest to demonstrate how the proposed solutions perform. In order to do that with the D3Q19 stencil, it would be necessary to decompose the domain using 3 blocks in each direction, such that the block at middle of the cube ($B_x = 2$, $B_y = 2$, $B_z = 2$), has to communicate with 18 neighbors. Doing so would require 27 GPUs, because each block is assigned to one GPU. Since the C3HPC cluster offers only up to 4 Kepler GPUs, the domain is set to be periodic in all directions, so that all blocks have to communicate with 18 neighbors in each communication step, thus circumventing the limitation on the number of GPUs.

Performance test is carried out on two Kepler nodes of the C3HPC cluster. Both *GPUBufferedScheme* and asynchronous *GPUPackInfo* are used for the communication step in separate runs, and compared with the previous state-of-the-art solution, *GPUPackInfo*. The domain is decomposed into 4 cubic blocks ($B_x = 1$, $B_y = 2$, $B_z = 2$), with one block per Kepler GPU. Simulation is repeated for 8, 16, 32, 64, 128 and 256 cubic domain sizes.

5.3.2 Communication Direction Imbalance

In this section, an experiment is presented to verify the overall communication time for each of the three directions, x (east/west), y (north/south), z (top/bottom), and demonstrate the higher communication overhead in some of these directions.

Neighboring block communication using ghost layers involves copying slices of axis-aligned data from fields. Copy performance may vary between slices, depending on the directions and memory layout of fields, mainly due to element strides. For the D3Q19 stencil, a process may have up to 18 neighbors to communicate. Since the domain is cubic, there are directions that extract planes from the cube (6 directions), which have $n_i \times n_j \times g$ elements, directions that extract lines (12 directions), which have $n_i \times g$ elements. Therefore, there are different amounts of data being copied depending on the direction.

Communication direction imbalance test is proposed to measure these discrepancies. The experiment consists in running the lid-driven cavity simulation, using a structure-of-arrays (fzyx) memory layout. Simulation domain is decomposed into 2 blocks, running on separate Kepler nodes of the C3HPC cluster, decomposing in each of the x, y and z directions. Blocks were kept cubic to prevent interference of any performance impact that could be introduced by directional changes in shape. Experiment was repeated with cubic block sizes of 16, 32, 64, 128 and 256 cells.

5.3.3 Scalability

In this section, an experiment is presented to verify the scalability of the proposed solutions. Two scalability experiments are proposed: weak and strong scaling [18].

Weak scaling consists in increasing the workload with the number of processes, in order to explore bigger domain sizes. The proposed experiment consisted in running the lid-driven cavity simulation with 1, 2, and 4 blocks, with the cubic block size fixed to 256 cells.

Strong scaling consists in reducing the time to solution (overall simulation time), by decomposing the domain further, as more processes are added. The proposed experiment consisted in running the lid-driven cavity simulation, decomposing the domain into 1, 2, and 4 blocks, using a cubic domain size of 256 cells.

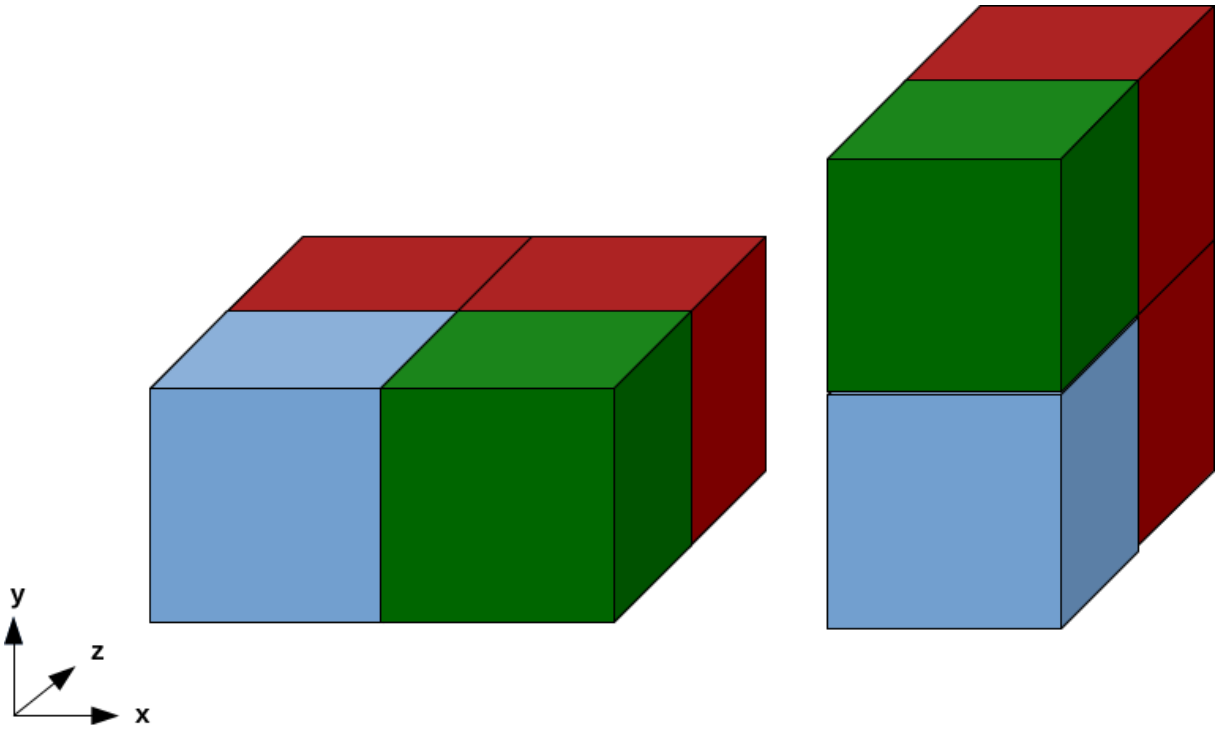

Figure 5.1: Representation of the xz and yz block decompositions, from left to right. Initial block has the blue color, the second block in the decomposition is depicted with the color green. The last two blocks for the decomposition with 4 blocks, are depicted with the color red.

Both scalability experiments were performed on Kepler nodes of the C3HPC cluster, with exactly one block per Kepler GPU. Domain was setup to be periodic in all directions, since periodic domains are the worst case scenario in terms of communication. Due to the limitation in the number of available GPUs, it was only possible to test the domain decomposition in two directions, since decomposing in three directions would require at least 8 GPUs. Therefore, two different scenarios for each type of scaling were tested to cover all domain decomposition directions:

1. **xz decomposition:** A first decomposition is performed in the x-axis and a second in the z-axis.

2. **yz decomposition:** A first decomposition is performed in the y-axis and a second in the z-axis.

The xz and yz decompositions are illustrated in Figure 5.1. In the case of weak scaling, blocks with a cubic size of 256^3 cells are added, as the number of processes increases. As for strong scaling, an initial block of 256^3 cells, is divided between processes, according to the direction of the decomposition. For example, in the first x-direction decomposition, with two processes, the block is decomposed into 2 blocks of $(128, 256, 256)$ cells, with one block per process, in the strong scaling case.

Chapter 6

Results and Discussion

6.1 Validation Results

In this section, the results from the validation experiments proposed in Chapter 5 are presented. Velocity profiles are extracted and compared with reference values from the literature [34, 10]. Figure 6.1 shows the streamlines for the converged solution of the LBM, after 20000 iterations, with a vortex forming around the y-axis, as expected.

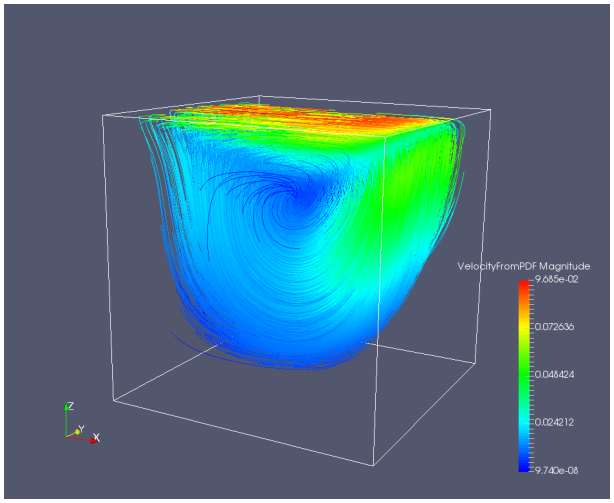

Figure 6.1: Streamlines for the converged solution of the lid-driven cavity problem.

Velocity profiles are extracted from the last time step of the LBM simulation after 20000 iterations. Simulation domain coordinates were normalized to the interval $[0,1]$. A slice of the three dimensional domain is taken in the middle of the y-axis ($y = 0.5$), and the velocity profiles of the x and z axes are taken.

In order to extract the velocity profile from the x-axis, the x coordinate is fixed at the mid-plane ($x = 0.5$), and the velocity is measured for varying values of z. The same process is done to extract the velocity in the z-axis, with z fixed at the z coordinate mid-plane.

Figure 6.2 depicts the velocities taken from the x and z axes, at the y mid-plane, which are referred as V_x and V_z, respectively. All validation simulations produce the same profile and are represented with lines. Reference values are represented with points. Results on Figure 6.2 demonstrate that the simulation results match with the reference values.

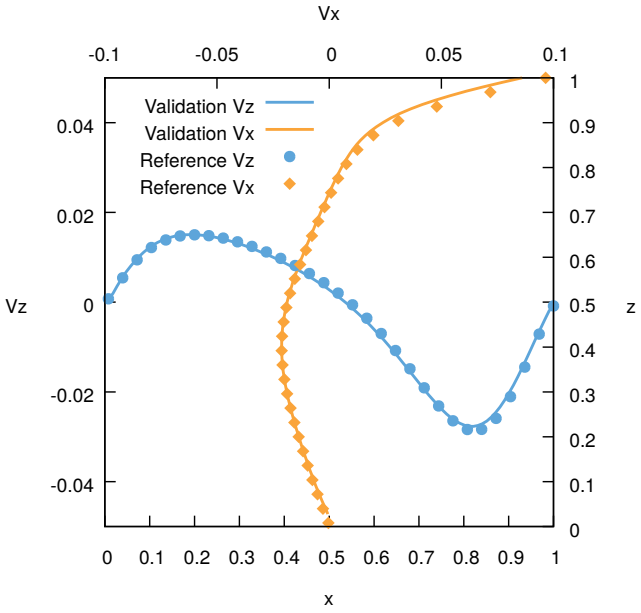

Figure 6.2: Velocity profiles for x and z components (Vx and Vz, respectively) at the y-midplane, compared with reference values from the literature [10, 34].

6.2 Performance Results

In this section, the results and analysis of the performance experiments, proposed in Section 5.3, are presented.

6.2.1 Communication Performance Results

In this section, the results and analysis of the communication performance experiment, proposed in Section 5.3.1, are presented.

As shown in Figure 6.3, both the proposed communication solutions, asynchronous *GPUPackInfo* and *GPUBufferedScheme*, outperform the state-of-the-art communication scheme, *GPUPackInfo*. Asynchronous *GPUPackInfo* has shown the best performance for the test scenario. In the same scenario, *GPUBufferedScheme* has shown a similar performance to *GPUPackInfo*.

Performance increase with asynchronous *GPUPackInfo* is more significant, as bigger block sizes are utilized, starting from 64^3. For a block size of 256^3 cells, asynchronous *GPUPackInfo* achieved 1149 MLUP/s, while *GPUBufferedScheme* and *GPUPackInfo* achieved 748 and 760 MLUP/s, respectively.

Communication latency was hidden up to 52% using asynchronous *GPUPackInfo*, with a block size of 320^3. This result comes from the fact that the measured single GPU performance on a Kepler GPU for the same block size was 550 MLUP/s, and if communication latency was fully hidden on the four Kepler nodes, a result of 2200 MLUP/s would be achieved. However, this is the worst case scenario, in which the domain is periodic in all directions, *i.e.* every process has to communicate in 18 directions. Hence, it is expected that communication can be further hidden when non-periodic domains are utilized.

The same experiment was performed with another MPI implementation, MVAPICH2 2.3, and it showed that the performance *GPUBufferedScheme* is highly dependent on the performance of the MPI implementation. Using MVAPICH2, *GPUBufferedScheme* performed considerably better than using OpenMPI. This is very likely related to the several optimizations

that MVAPICH2 employs, which were described in more detail in Chapter 3. Since MVAPICH2 with CUDA support was not available in the C3HPC cluster, the experiment was conducted on the VRI group's server [1], using only intra-node GPU communication with 2 NVIDIA GeForce Titan X GPUs. Results of this experiment are presented in Figure 6.4. Therefore, it is highly recommended that users rely on the MVAPICH2 MPI implementation to improve communication performance in GPU simulations, specially with larger block sizes.

These results show that the proposed communication solutions improve communication performance with respect to the state-of-the art communication solution available in waLBerla. Furthermore, it is important to choose a proper MPI implementation in order to take the maximum advantage of GPU buffer based communication.

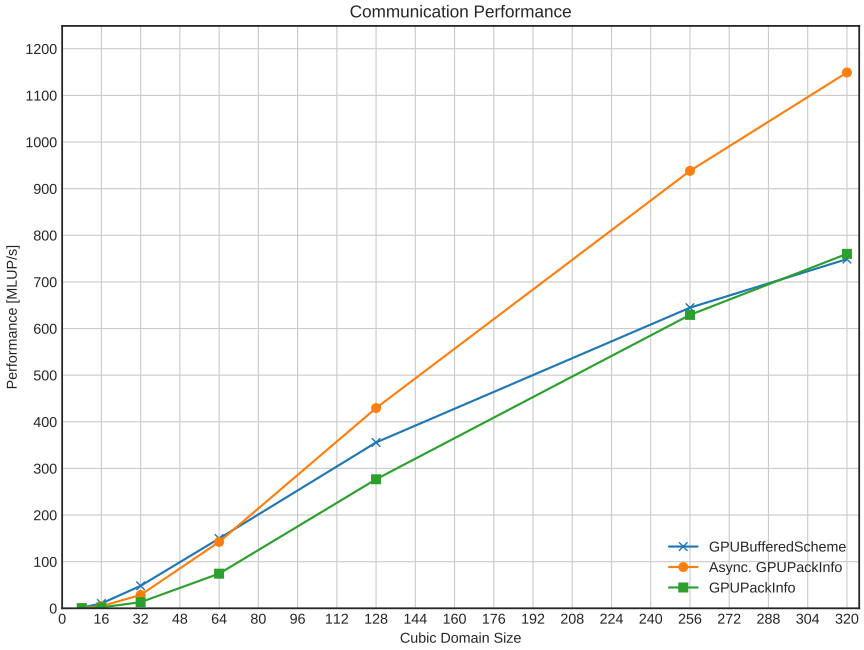

Figure 6.3: Performance comparison of the state-of-the art communication scheme, with the proposed solutions. Results are measured for increasing cubic block sizes. The experiment was performed on 4 Kepler nodes and the domain decomposition in this case is $(x = 1, y = 2, z = 2)$.

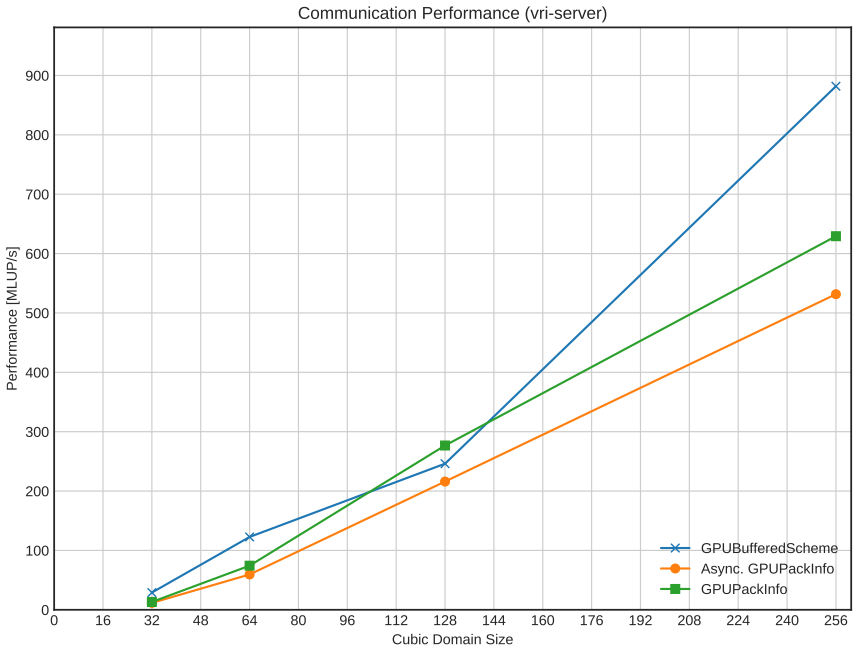

Figure 6.4: Performance comparison of the state-of-the art communication scheme, with the proposed solutions, using MVAPICH2 2.3 with CUDA support. Results are measured for increasing cubic block sizes. The experiment was performed on 2 NVIDIA GeForce Titan X GPUs of the VRI group's server (single node) and the domain decomposition in this case is $(x = 1, y = 1, z = 2)$. In this case, only intra-node GPU communication is performed.

6.2.2 Communication Direction Imbalance Results

In this section, the results and analysis of the communication direction imbalance experiment, proposed in Section 5.3.2, is presented. Previous results for the communication direction imbalance were reported by Carvalho *et. al* [10], with a bigger impact when the communication happened at the x-axis

Although the simulation performance when the domain is decomposed along the x-axis still affects the performance of *GPUPackInfo*, the same trend is not verified in *GPUBufferedScheme*. The discrepancy between the performance of the x-axis communication and the communication in y and z axes is still significant in asynchronous *GPUPackInfo*. In *GPUBufferedScheme*, there is still a gap in the same scenario, however the performance does not degrade as severely as in the case *GPUPackInfo*.

Therefore, the use of *GPUBufferedScheme* is recommended for simulations that require a higher ratio of decomposition in the x direction. While for *GPUPackInfo*, the same advice remains from [10], *i.e.* the domain should be further decomposed in the y and z directions, in comparison with the x direction, to reduce the amount of data that is communicated in that direction.

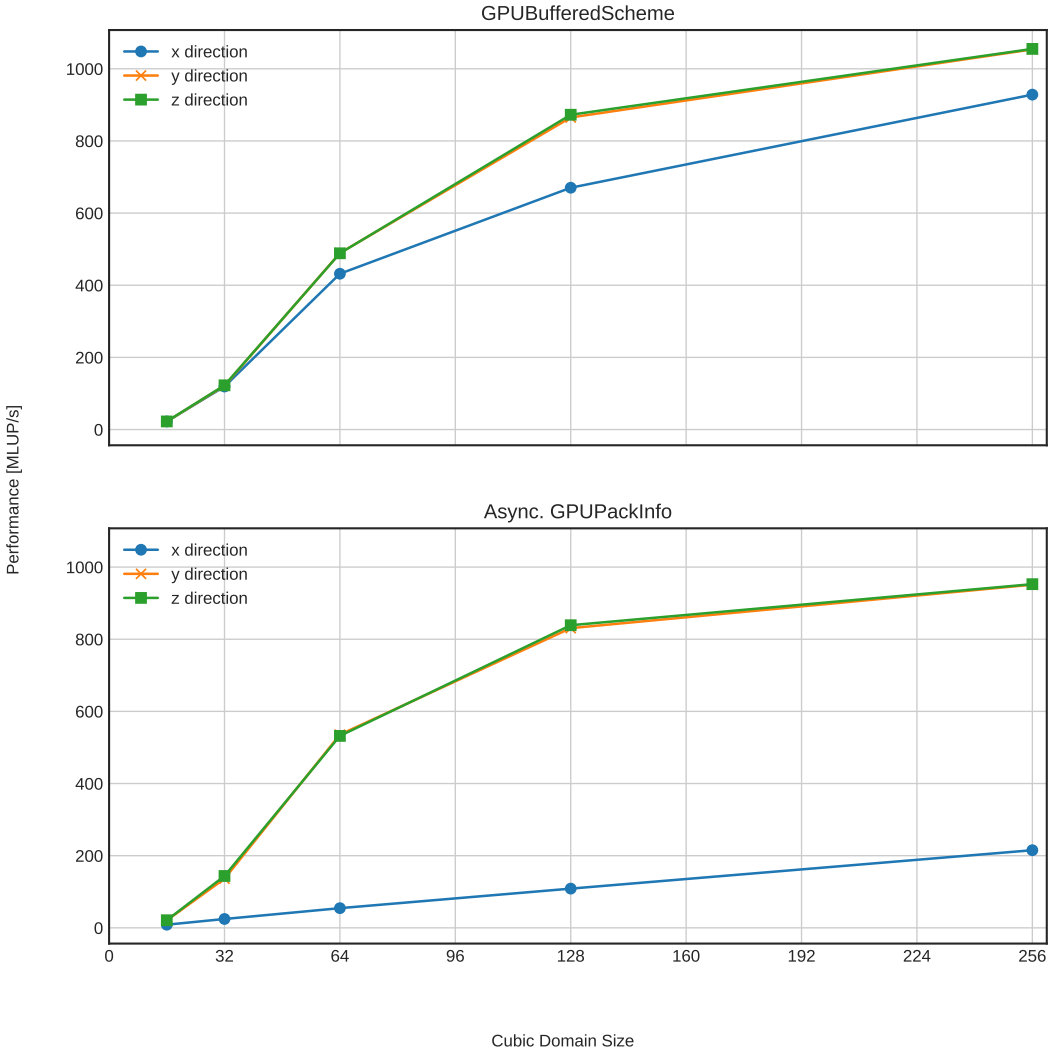

Figure 6.5: Communication direction imbalance results for the proposed communication solutions.

6.2.3 Scalability Results

In this section, the results and analysis of the weak and strong scaling experiments, proposed in Section 5.3.3, are presented. Weak scaling experiment results for different domain decompositions are depicted in Figure 6.6. Likewise, strong scaling results are depicted in Figure 6.7.

Results in the weak scaling experiment, demonstrated that *GPUBufferedScheme* outperformed the asynchronous *GPUPackInfo* on almost all domain decompositions. For the xz decomposition, *GPUBufferedScheme* completely outperforms *GPUPackInfo*, achieving up to 854 MLUP/s on 2 Kepler nodes, while *GPUPackInfo* achieves only 100 MLUP/s.

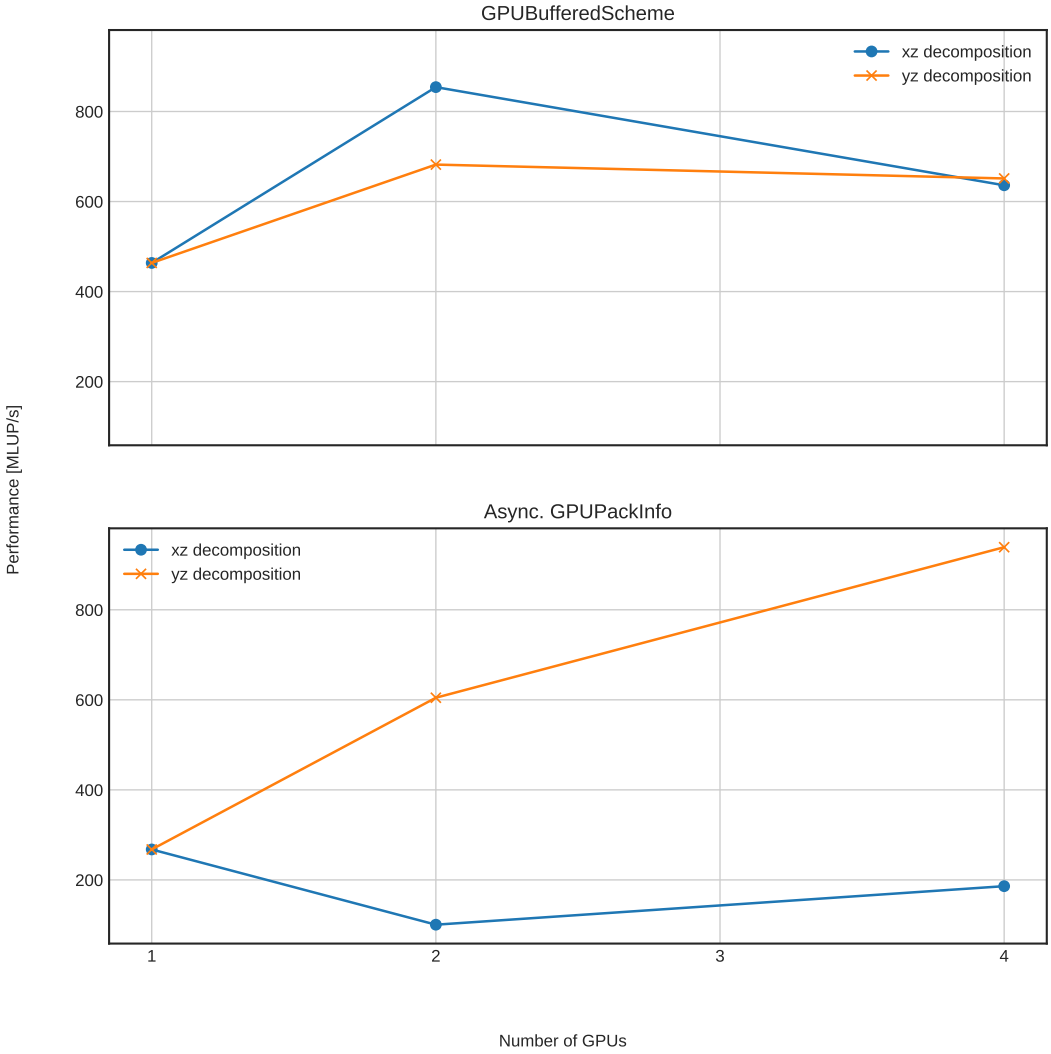

Figure 6.6: Results of the weak scaling experiment for a cubic domain size of 256 cells, for the xz and yz domain decomposition directions.

Note that the yz decomposition presents the best scalability curve, with the performance increasing almost linearly as more nodes are added, specially for asynchronous *GPUPackInfo*. However it is not possible to precise where the scalability curve stabilizes, since there are only 4 GPUs available in the cluster. Finally, *GPUBufferedScheme* performed more consistently for different domain decompositions, even when the domain was decomposed in the x-direction.

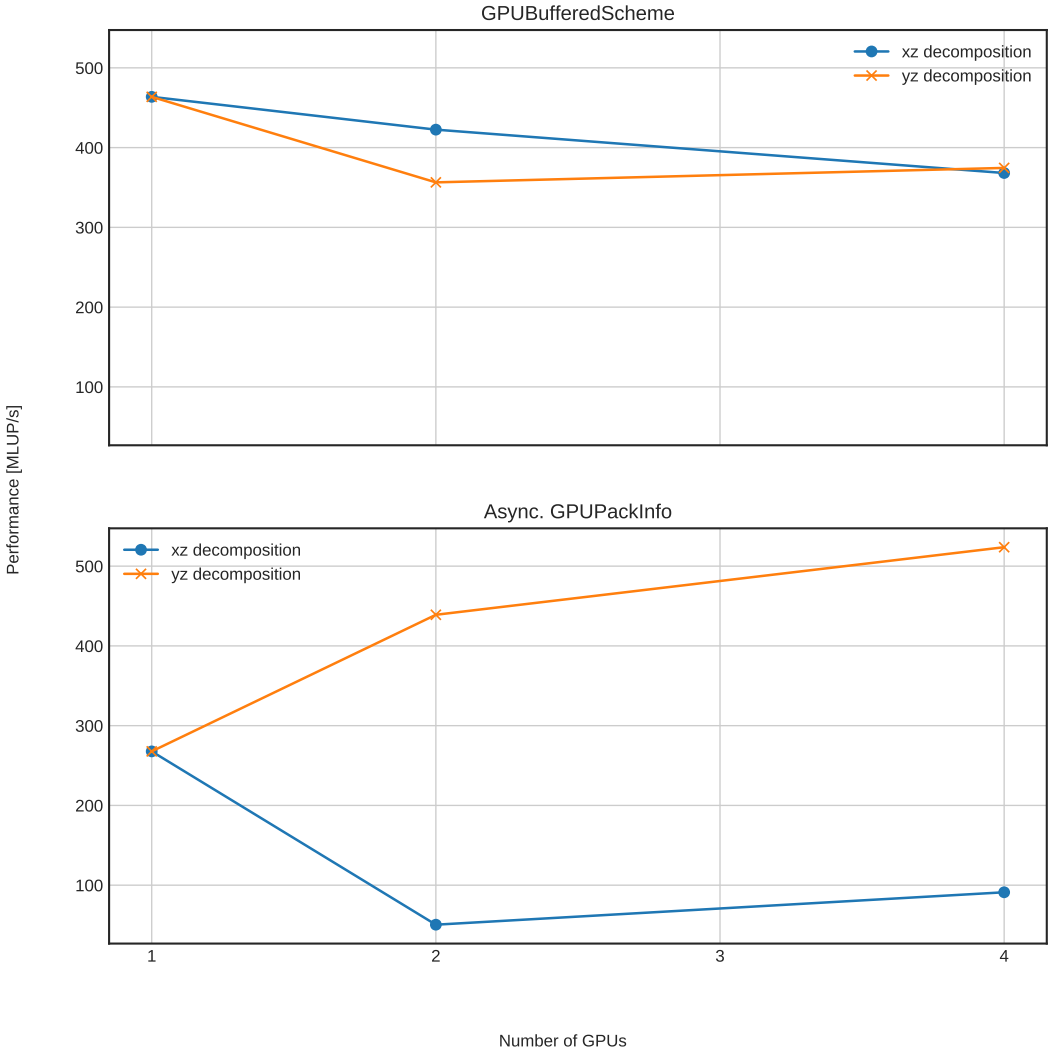

Figure 6.7: Results of the strong scaling experiment for a cubic domain size of 256 cells, for the xz and yz domain decomposition directions.

Chapter 7

Conclusion

In this thesis, solutions for improving communication performance, hiding communication latency, and usability of GPUs in the HPC field, considering different communication scenarios were proposed. Two GPU communication solutions were evaluated.

Due to the increasing role of GPUs in supercomputers, GPU support in HPC software frameworks becomes an important feature. Therefore, adding the proposed solutions into an HPC software framework so that they can be easily utilized by all users of the framework, consists in a valuable contribution.

Previous state-of-the-art communication strategy for the waLBerla framework did not explore concurrency mechanisms. In order to improve its performance, CUDA streams and (un)packing kernels were introduced. A novel communication scheme was also introduced, that makes use of GPU memory buffers, which are supported by modern MPI implementations.

Proposed solutions achieved up to 50% improvement over the state-of-the-art GPU communication solution, on the worst case scenario for communication, using a periodic domain in all directions. In the case of the GPU memory buffer based solution, performance was also shown to be dependent on the underlying MPI implementation and communication direction imbalance in the x-direction did not impose a big performance penalty, in comparison with previous solutions. Configurable memory alignment for GPU fields was also introduced, allowing different memory alignments to be used in simulations.

Finally, there are still many challenges for utilizing the full potential of GPUs on supercomputers. Improving communication performance brings a valuable benefit, however the biggest challenge is to perform load balancing in processes that are performing GPU computations, which stems from adaptive grid refinements with GPU fields (also, not yet possible). Load balancing of processes is still a subject for future research.

7.1 Future Work

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

- Investigate the problem in the validation experiment, in which 20000 iterations were required for convergence. Also, the results did not fit all the reference values. In a previous work [10], 10000 iterations were required for convergence.
- Allow for GPU structures to be used from waLBerla's Python interface, which is supported for CPU fields. This would allow GPU simulations to be setup and performed using Python instead of C++ code.

- Testing the proposed domain decomposition strategy to verify the communication 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.
- Adaptive grid refinement for GPU fields, which is supported for CPU fields. An implementation could be based on the work performed for CPU fields [36].
- Load balancing for non-uniform grids on GPUs, which is supported for CPUs in waLBerla, but not on GPUs. An implementation could be based on the work performed for CPU fields [35].

Bibliography

[1] *VRI – Visão, Robótica e Imagem*, May 2017 (accessed on May 29, 2017). `http://web.inf.ufpr.br/vri`.

[2] Stefan Albensoeder and Hendrik C Kuhlmann. Accurate three-dimensional lid-driven cavity flow. *Journal of Computational Physics*, 206(2):536–558, 2005.

[3] 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*, 31(6):529–542, 2016.

[4] Massimo Bernaschi, Mauro Bisson, and Davide Rossetti. Benchmarking of communication techniques for gpus. *Journal of Parallel and Distributed Computing*, 73(2):250–255, 2013.

[5] 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.

[6] C3SL. *C3HPC – C3SL*, September 2017. `http://www.c3sl.ufpr.br/c3hpc/`.

[7] Shiyi Chen and Gary D Doolen. Lattice boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.

[8] Ching-Hsiang Chu, K Hamidouche, A Venkatesh, DS Banerjee, H Subramoni, and Dhabaleswar K Panda. Exploiting maximal overlap for non-contiguous data movement processing on modern gpu-enabled systems. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 983–992. IEEE, 2016.

[9] Intel Corporation. *Intel MPI Library*, May 2017 (accessed on May 02, 2017). `https://software.intel.com/en-us/intel-mpi-library`.

[10] Paulo Roberto de Carvalho Junior. Gpu communication performance engineering for the lattice boltzmann method. Master's thesis, Programa de Pós-Graduação em Informática - Universidade Federal do Paraná, Curitiba – PR, June 2016.

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

[12] Christian Feichtinger, Stefan Donath, Harald Köstler, Jan Götz, and Ulrich Rüde. Walberla: Hpc software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105–112, 2011.

[13] MPI Forum. *MPI: A Message-Passing Interface Standard – Version 3.0*, September 2012. http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[14] 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.

[15] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction*. SIAM, 1998.

[16] Johannes Habich, Christian Feichtinger, Harald Köstler, Georg Hager, and Gerhard Wellein. Performance engineering for the lattice boltzmann method on gpgpus: Architectural requirements and performance results. *Computers & Fluids*, 80:276–282, 2013.

[17] Johannes Habich, Thomas Zeiser, Georg Hager, and Gerhard 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.

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

[19] IBM. *IBM Platform MPI*, May 2017 (accessed on May 02, 2017). http://www.ibm.com/systems/platformcomputing/products/mpi/.

[20] Dana Jacobsen, Julien Thibault, and Inanc Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522, 2010.

[21] Jiri Kraus. *An Introduction to CUDA-Aware MPI*, May 2017 (accessed on May 02, 2017). http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/.

[22] Network-Based Computing Laboratory. *MVAPICH2: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE*, May 2017 (accessed on May 02, 2017). http://mvapich.cse.ohio-state.edu/.

[23] John D. McCalpin. *STREAM Benchmark*, April 2016 (accessed on April 26, 2017). https://www.cs.virginia.edu/stream/.

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

[25] NVIDIA. *Developing a Linux Kernel Module using GPUDirect RDMA*, April 2017 (accessed on May 05, 2017). http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[26] NVIDIA. *GPUDirect*, April 2017 (accessed on May 05, 2017). https://developer.nvidia.com/gpudirect.

[27] NVIDIA. *CUDA Zone*, May 2017 (accessed on May 29, 2017). https://developer.nvidia.com/cuda-toolkit.

[28] Christian Obrecht, Frédéric Kuznik, Bernard Tourancheau, and Jean-Jacques Roux. Multi-gpu implementation of the lattice boltzmann method. *Computers & Mathematics with Applications*, 65(2):252–261, 2013.

[29] 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.

[30] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[31] DP Playne and KA Hawick. Asynchronous communication for finite-difference simulations on gpu clusters using cuda and mpi. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)', number PDP2793, Las Vegas, USA*, 2011.

[32] 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.

[33] 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.

[34] Pablo R Rinaldi, EA Dari, Marcelo J Vénere, and Alejandro Clausse. A lattice-boltzmann solver for 3d fluid simulation on gpu. *Simulation Modelling Practice and Theory*, 25:163–171, 2012.

[35] Florian Schornbaum and Ulrich Rüde. Massively parallel algorithms for the lattice boltzmann method on nonuniform grids. *SIAM Journal on Scientific Computing*, 38(2):C96–C126, 2016.

[36] Florian Schornbaum and Ulrich Rüde. Extreme-scale block-structured adaptive mesh refinement. *arXiv preprint arXiv:1704.06829*, 2017.

[37] 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, Curitiba – PR, August 2015.

[38] Mohammed Sourouri, Tor Gillberg, Scott B Baden, and Xing Cai. Effective multi-gpu communication using multiple cuda streams and threads. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 981–986. IEEE, 2014.

[39] Erich Strohmaier, Jack Dongarra, Horst Simon, Martin Meuer, and Hans Meuer. *TOP500 Supercomputers*, November 2017 (accessed on May 28, 2017). `http://www.top500.org/`.

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

[41] The MPICH Development Team. *MPICH: High-Performance Portable MPI*, May 2017 (accessed on May 02, 2017). `http://www.mpich.org/`.

[42] The Open MPI Development Team. *Open MPI: Open Source High Performance Computing*, May 2017 (accessed on May 02, 2017). `http://www.open-mpi.org/`.

[43] 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.

[44] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters. *Computer Science-Research and Development*, 26(3-4):257, 2011.

[45] Wang Xian and Aoki Takayuki. Multi-gpu performance of incompressible flow computation by lattice boltzmann method on gpu cluster. *Parallel Computing*, 37(9):521–535, 2011.

[46] QinGang Xiong, Bo Li, Ji Xu, XiaoJian Fang, XiaoWei Wang, LiMin Wang, XianFeng He, and Wei Ge. Efficient parallel implementation of the lattice boltzmann method on large clusters of graphic processing units. *Chinese Science Bulletin*, 57(7):707–715, 2012.