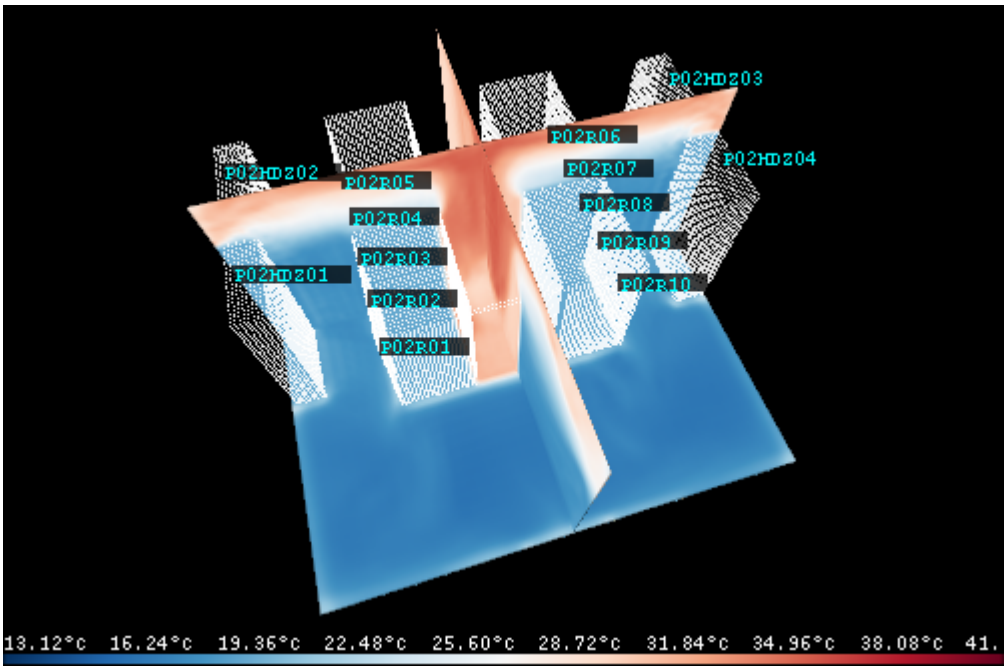
DEGREE PROJECT

Real-time Thermal Flow Predictions for Data Centers

Using the Lattice Boltzmann Method on Graphics Processing Units for Predicting Thermal Flow in Data Centers



Johannes Sjölund

Computer Science and Engineering, master's level
2018

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Abstract

The purpose of this master thesis is to investigate the usage of the Lattice Boltzmann Method (LBM) of Computational Fluid Dynamics (CFD) for real-time prediction of indoor air flows inside a data center module. Thermal prediction is useful in data centers for evaluating the placement of heat-generating equipment and air conditioning.

To perform the simulation a program called RAFSINE was used, written by Nicholas Delbosc at the University of Leeds, which implemented LBM on Graphics Processing Units (GPUs) using NVIDIA CUDA. The program used the LBM model called Bhatnagar-Gross-Krook (BGK) on a 3D lattice and had the capability of executing thermal simulations in real-time or faster than real-time. This fast rate of execution means a future application for this simulation could be as a predictive input for automated air conditioning control systems, or for fast generation of training data sets for automatic fault detection systems using machine learning.

In order to use the LBM CFD program even from hardware not equipped with NVIDIA GPUs it was deployed on a remote networked server accessed through Virtual Network Computing (VNC). Since RAFSINE featured interactive OpenGL based 3D visualization of thermal evolution, accessing it through VNC required use of the VirtualGL toolkit which allowed fast streaming of visualization data over the network.

A simulation model was developed describing the geometry, temperatures and air flows of an experimental data center module at RISE SICS North in Luleå, Sweden, based on measurements and equipment specifications. It was then validated by comparing it with temperatures recorded from sensors mounted in the data center.

The thermal prediction was found to be accurate on a room-level within ± 1°C when measured as the average temperature of the air returning to the cooling units, with a maximum error of ± 2°C on an individual basis. Accuracy at the front of the server racks varied depending on the height above the floor, with the lowest points having an average accuracy of ± 1°C, while the middle and topmost points had an accuracy of ± 2°C and ± 4°C respectively.

While the model had a higher error rate than the ± 0.5°C accuracy of the experimental measurements, further improvements could allow it to be used as a testing ground for air conditioning control or automatic fault detection systems.

Preface

Thanks to the people at RISE SICS North for helping me during my master thesis work, in particular my external supervisor Jon Summers. Also thanks to my internal supervisor Johan Carlson at Luleå University of Technology (LTU), Anna-Lena Ljung at LTU, as well as fellow student Rickard Nordlander for proofreading and comments on the thesis work. I would also like to thank Emelie Wibron for providing measurements and drawings of the data center modules at RISE SICS North.

Johannes Sjölund

Contents

Chapter 1

Introduction

An important part of the physical design and construction of a data center is determining optimal air flows for cooling the computer equipment at the correct air temperature and humidity. Since physical experiments in building designs are very costly to construct, computer simulations can be used to analyze how the distribution of air flows behave under different physical conditions. Such simulations are based on the theory of Computational Fluid Dynamics (CFD), a branch of fluid mechanics which uses numerical analysis and data structures to solve the dynamics of fluid behaviors.

The CFD software used in this thesis project, called RAFSINE, was written by Nicolas Delbosc as part of his PhD work at the University of Leeds and documented in his thesis *Real-Time Simulation of Indoor Air Flow using the Lattice Boltzmann Method on Graphics Processing Unit* [4].

There are many different CFD software packages available on the market, such as COMSOL which is based on a computational technique called the Finite Element Method (FEM), or ANSYS CFX which uses a hybrid of FEM and the Finite Volume Method (FVM). In addition, there exist free and open-source solutions such as OpenFOAM which uses FVM. These finite methods are based on numerical solutions for Partial Differential Equations (PDEs), more specifically the Navier-Stokes equations briefly described in chapter 2.1. Unlike these software packages, RAFSINE is based on the Lattice Boltzmann method (LBM) which is a computational method for solving the discrete Boltzmann Transport Equation (BTE). Chapters 2.3 and 2.4 introduces LBM and the RAFSINE code respectively.

The main advantage of the LBM compared to FVM, FEM and similar is that it is highly parallelizable, which makes it suitable for execution on a general-purpose Graphics Processing Unit (GPU). According to a benchmark between CFD softwares performed by the original author, in which the temperatures inside a small data center were simulated, the COMSOL package took 14.7 hours to converge on a solution, while RAFSINE had a convergence time of 5 minutes [4, pg.168]. This fast execution rate makes it possible to perform the CFD simulation in real-time or faster than real-time. Such a high rate

of execution of the simulation model could theoretically allow it to be used not only for testing different air conditioning control systems, but also for integration into closed loop control systems. In this case, the predictive model could be used by a control algorithm as a reference point for example when setting the speed of the fans in a cooling unit. Another use case of the model is for fast training data set generation for sensor based automatic fault detection of cooling equipment by machine learning algorithms. The effect of a malfunction could be simulated and temperatures recorded, without the risk of damaging any real data center equipment.

Since many modern office environments at the time of writing do not have computer workstations equipped with powerful GPUs, but laptops with somewhat weaker integrated graphics card solutions, it is advantageous to allow the execution of such software on remote networked servers. One of the first goals of this master thesis project was to deploy RAFSINE to a remote server running the UNIX operating system Ubuntu[1] and equipped with an NVIDIA GeForce GTX 1080 Ti GPU. This involved configuring an improved build system for the source code using CMake and is described in chapter 3.1.

While the server was equipped with a GPU and therefore had graphics rendering capability, it was headless, meaning no monitor was attached to the GPU and the only way to access it was through remote access systems such as Virtual Network Computing (VNC) or Secure Shell (SSH). Since RAFSINE featured user-interactive graphical visualization using Open Graphics Library (OpenGL), a feature normally not possible to use over VNC, the server had to be configured to use a toolkit called VirtualGL. This allowed the OpenGL graphics to be rendered by the server GPU instead of the client, thus overcoming this limitation and allowing low-latency streaming of visualization data over the network. The theory behind VirtualGL is described in chapter 3.2.

When executing RAFSINE over a VirtualGL enabled VNC session, it was found that the computational overhead from VirtualGL limited the rate of execution for the simulation. This was solved by making the application multi-threaded, to decouple the visualization part of the application from the LBM simulation kernel execution. Also, to be able to perform simulation of time-dependent behavior such as transient heat generation for servers and varying flow rate from cooling fans in the simulated data center, it was necessary to add support to RAFSINE for real-time boundary condition modification. Chapter 4 describes the changes made to the original RAFSINE source code.

After the application had been deployed on the server, and necessary changes had been made to the source code, a simulation model of a data center module called POD 2 at Research Institutes of Sweden, Swedish Institute of Computer Science, North (RISE SICS North) was developed. It described the physical geometry and boundary conditions of the heat and air flows inside it based on equipment specifications and measured data from sensors. Chapter 5 describes how this data was used when constructing the model.

From earlier heating experiments in the data center, a log file of recorded temperatures, power usages and air mass flow rates was available for comparison and model validation.

[1] https://www.ubuntu.com/

Chapter 6 discusses the simulation results and the validity of the model. Finally, the overall results and conclusions of the thesis work are discussed in chapter 7.

Chapter 2

Background

2.1 Fluid Dynamics

Fluid mechanics, as implied by its name, is a branch of physics dealing with the mechanics of fluids such as gases, liquids and plasmas, and how forces interact with them. Its applications range from mechanical engineering, such as the propagation of fluid in motors and pumps, to civil engineering, such as how wind and water interact with buildings, as well as chemical, biomedical engineering, and many more fields.

Fluid mechanics can be divided into the subdisciplines of *fluid statics* which describes fluids at rest and *fluid dynamics* which describes the behavior of fluids in motion. In itself, fluid dynamics can be divided into the field of hydrodynamics, which is the study of liquids in motion, and aerodynamics which describes the motion of air and other gases. While these two fields are practical disciplines, fluid dynamics are their common underlying structure since it contains the laws for how to solve practical problems from flow measurements such as velocity, pressure, density, temperature and how they change over space and time.

The basis for the field of fluid dynamics are the laws for the *conservation of mass*, *conservation of linear momentum* and *conservation of energy*. For mathematical models of fluid dynamics, it is also often assumed that fluids can be described as a *continuum*, which means that they are not seen as discrete molecules and that properties such as temperature, velocity, density and pressure vary continuously between all points in space. This is called the *continuum assumption* and is one of the models under which the famous Navier-Stokes equations operate.

2.1.1 The Navier-Stokes Equations

Fluids such as air, water and thin oils, can be modeled as *Newtonian fluids*, which means they have the property that viscous stresses from their flow have a linear relationship to their local change of deformation over time. For Newtonian fluids which are dense enough to fit the continuum assumption, the laws which govern their momentum are the *Navier-Stokes equations*. In the case of compressible fluids such as air, it can be stated as the equation [9, pg.46]

$$\rho\left(\frac{\partial\vec{u}}{\partial t}+(\vec{u}\cdot\nabla)\vec{u}\right)=-\nabla\vec{p}+\mu\nabla^2\vec{u}-\left(\frac{1}{3}\mu+\zeta\right)\nabla^2\vec{u}, \tag{2.1}$$

where $\vec{u}$ is the velocity vector, p the pressure, ρ the density, μ and ζ are viscosity coefficients of the fluid. The left hand side of the equation correspond to inertial forces in the fluid, the first term on the right side the pressure forces, second the viscous forces and the last represents internal forces.

While air is a compressible fluid, indoor air flows, as opposed to for example pressurized air, can be modeled as an incompressible fluid. The Navier-Stokes equation can then be simplified so that the divergence of the velocity field $\nabla^2\vec{u}=0$, which yields [9, pg.47]

$$\frac{\partial\vec{u}}{\partial t}+(\vec{u}\cdot\nabla)\vec{u}=-\frac{1}{\rho_0}\nabla p+\nu\nabla^2\vec{u}-F, \tag{2.2}$$

where ρ_0 is a uniform fluid density, $\nu=\mu/\rho_0$ is the kinematic viscosity, and F is an external force exerted on the fluid.

While the Navier-Stokes equations represent the conservation of linear momentum and pressure, they are always used together with the *continuity equation* representing the conservation of mass,

$$\frac{\partial\rho}{\partial t}+\nabla\cdot(\rho\vec{u})=0. \tag{2.3}$$

When considering an incompressible fluid, the continuity equation can be simplified as

$$\nabla\cdot\vec{u}=0. \tag{2.4}$$

While the Navier-Stokes equations and the continuity equation describe the behavior of fluids as represented by a continuum, in order to apply them to any particular problem they also need models of conditions and constraints for the problem to be solved. Chapter 2.2 describes how they can be applied to practical problems.

2.1.2 Turbulence

The flow of a fluid is considered *laminar* when all parts of it move in a parallel uniform and regular fashion, such that its internal shear stress is proportional to its velocity. As the shear stress increases so that velocities at different points in space fluctuate randomly over time, the flow becomes *turbulent*. Turbulence is mathematically described by a dimensionless number called the *Reynolds number* which is given by

$$\mathrm{Re} = \frac{\vec{U}\ell}{\nu} \tag{2.5}$$

where $\vec{U}$ is the mean fluid velocity of a region, ν is kinematic viscosity and ℓ is a characteristic length scale usually determined by the problem domain in which the turbulence is modeled.

The ordinary Navier-Stokes equations are only valid for laminar flows [1, pg.639]. For modeling the dynamics of turbulent flows, the time-averaged equations of motion called the Reynolds Averaged Navier-Stokes (RANS) equations are often used. These equations decompose the fluid flow model into a fluctuating part and another part which only considers the mean values of fluid properties such as velocity and pressure.

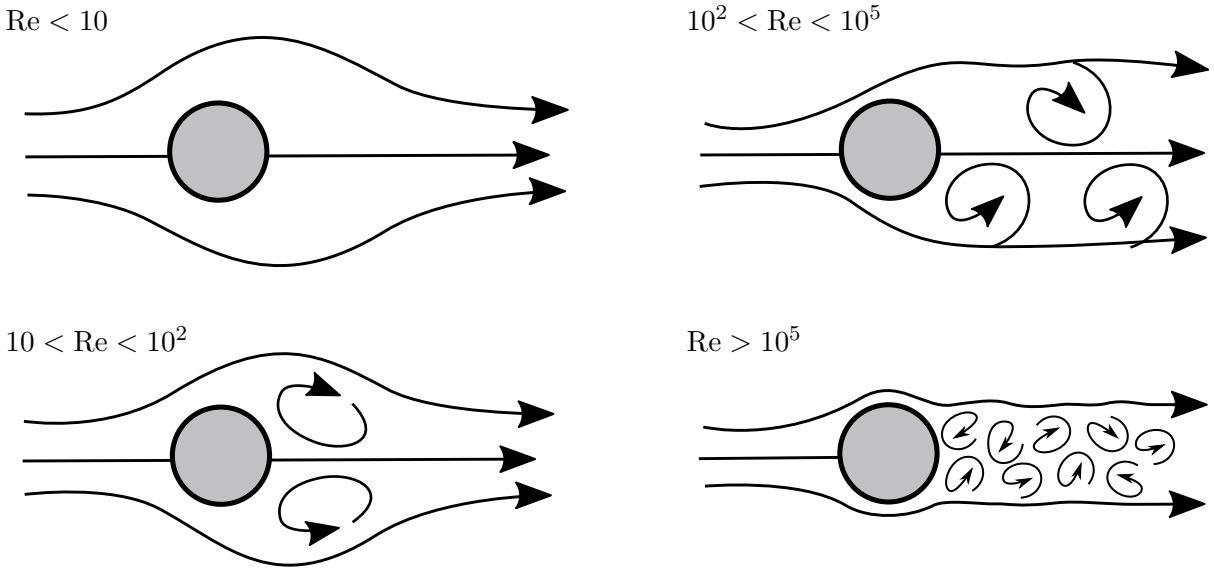


Figure 2.1: Flow conditions for different Reynolds numbers.

2.2 Methods in Computational Fluid Dynamics

Fluid dynamics, and continuum mechanics in general, is the attempt to formulate a particular physical problem using sets of general PDEs, for example the Navier-Stokes and continuity equations, constrained by *initial-* and *boundary-conditions* which are unique for a particular problem. Together they form an Initial Boundary Value Problem (IBVP) which has a unique solution, that can be found either by analytical (exact) or numerical (approximated) methods [1, pg.6].

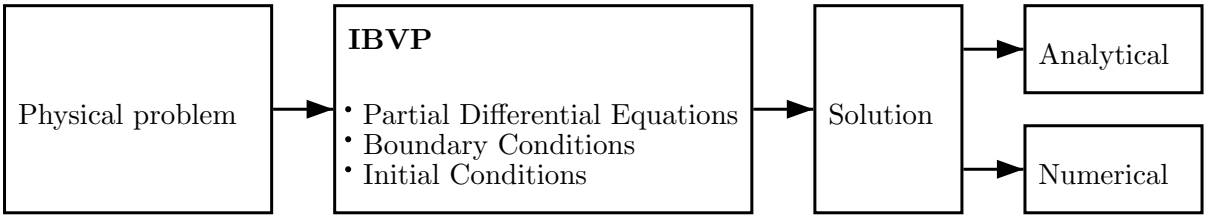


Figure 2.2: The Initial Boundary Value Problem.

The study of CFD is mainly concerned with the numerical methods of solving the IBVP. Since it may be very difficult to find a solution directly from the PDEs and the constraints of the IBVP, due to complex geometry and nonlinearity, several schemes have been developed to reduce the problem into systems of discrete points, volumes or elements governed by simple algebraic equations modeling their interactions. These can approximate the original problem and be more easily solved by iterative algorithms until they converge on a solution.

- The *Finite Volume Method (FVM)* is a common computational technique in CFD. It involves solving PDEs by calculating the values of variables averaged across a volume in the domain. One of its advantages is that it is not confined to a regular grid or mesh [11]. The software package ANSYS CFX uses a hybrid of FVM and FEM.

- The *Finite Element Method (FEM)* is mostly used in the field of solid mechanics. It is based on discretizing the domain into subdomains where the governing equations are valid [1, pg.7]. It is used in the CFD software COMSOL.

- The *Finite Difference Method (FDM)* involves dividing the continuum of the problem domain into discrete points where the governing equations can be applied [1, pg.7].

- In the *Boundary Element Method* (BEM), only the domain boundary is discretized

into elements and is useful when working with semi-infinite or infinite problem domains [1, pg.7].

In general, these methods involve discretizing the IBVP both by its spatial properties (such as displacement, volume and pressure) as well as their rates of change (e.g. velocity).

As mentioned previously, the Navier-Stokes equations and CFD methods based on them are useful when a fluid is modeled as a continuum instead of individual particles, or the so called *macroscopic scale* instead of the *microscopic scale*. There exists however a scale between these two, called the *mesoscopic scale*. Here the particles are considered to be neither a continuum nor individual units. Instead, fluids are modeled by their behavior in collections of particles, with properties modeled by statistical *distribution functions* [12, pg.3]. Figure 2.3 shows the relationship between these representations. The mesoscopic scale is the one on which the Lattice Boltzmann method (LBM) operates.

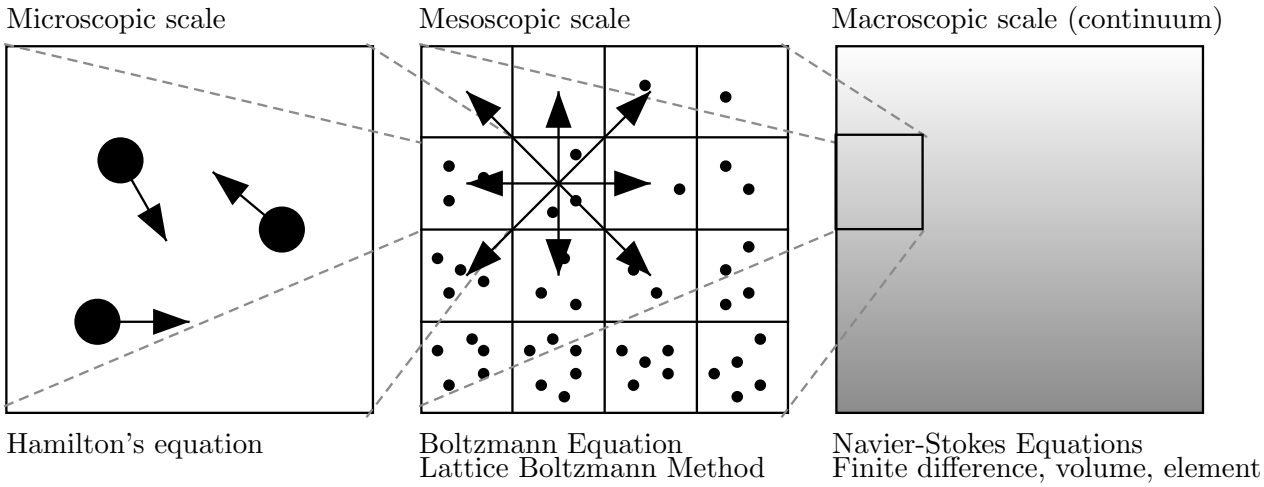


Figure 2.3: Techniques of simulations for different scales of fluid representations.

2.3 The Lattice Boltzmann Method

The CFD technique called the Lattice Boltzmann method (LBM) is based on the concept of *cellular automaton*, which is a discrete computational model based on a regular grid (also called *lattice*) of cells (lattice sites). The grid can have any finite number of dimensions and each site has a finite number of states, in the simplest models only the states *true* or *false*. At time $t = 0$ the state of each site is initialized to some predetermined value. As time progresses in discrete steps such that $t = t + \Delta t$ the state of each site is changed according to some fixed rule, depending on their own current state and those of

their neighbors on the lattice.

In the case of LBM as used for CFD modeling, the lattice can have either two or three dimensions, the states of the lattice sites are called *distribution functions*, and the rule of the automaton is the Discrete Lattice Boltzmann Equation.

2.3.1 The Boltzmann Equation

The Boltzmann equation is based on a concept called *kinetic theory*. The idea behind it is that gases are composed of particles following the laws of classical mechanics, but that considering how each particle interacts with its neighboring particles is not necessary. Instead a statistical representation can describe how groups of particles affect adjacent groups by the notion of streaming behavior between each other in a billiard-like fashion. This system can be described by the velocity distribution function

$$f^{(N)}(\vec{x}^{(N)}, \vec{p}^{(N)}, t) \tag{2.6}$$

which gives the probability of finding N number of particles with the displacements $\vec{x}$ and momentums $\vec{p}$ at the time t. In reality however, the first order particle distribution function $f^{(1)}$ is sufficient to describe all properties of non-dilute gases [15, pg.28].

When an external force F acts on the particles, their future positions and momentum can be described by

$$f^{(1)}(\vec{x} + d\vec{x}, \vec{p} + d\vec{p}, t + dt) \tag{2.7}$$

as long as no collisions occur between particles. This is called the particle streaming motion and models fluid advection (or convection) which is the transport of fluid properties by bulk motion. When taking particle collisions into account however, the evolution of this distribution function by particle interactions over time can be described by the *Boltzmann Equation*

$$\left(\vec{u} \cdot \frac{\partial}{\partial \vec{x}} + F \cdot \frac{\partial}{\partial \vec{p}} + \frac{\partial}{\partial t}\right) f^{(1)}(\vec{x}, \vec{p}, t) = \Gamma^{(+)} - \Gamma^{(-)}. \tag{2.8}$$

The left hand side describes the streaming motion introduced by the external force F during time dt. The right hand side contains two so called *collision operators* which act on the velocity $\vec{u}$. $\Gamma^{(-)}$ represents the number of particles starting at $(\vec{x},\ \vec{p})$ and not arriving at $(\vec{x} + d\vec{x},\ \vec{p} + d\vec{p})$ due to particle collisions. Conversely, $\Gamma^{(+)}$ is the number of particles not starting at at $(\vec{x}, \vec{p})$ but ending up at $(\vec{x} + d\vec{x},\ \vec{p} + d\vec{p})$ [15, pg.29]. This step models the diffusion of properties in the fluid.

Together the streaming and collision motion models what is called *advection-diffusion*, which describes how physical quantities such as temperatures and particle velocities are transferred in a fluid system.

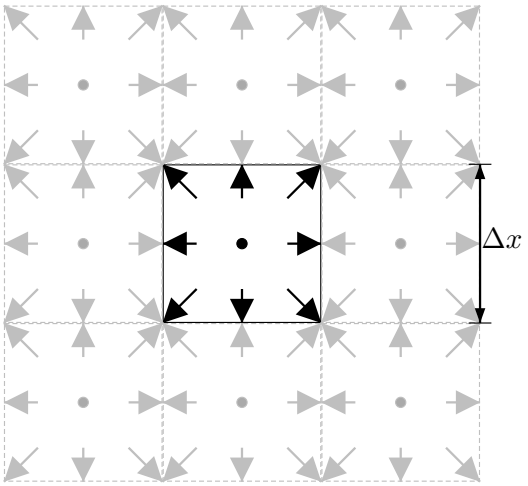
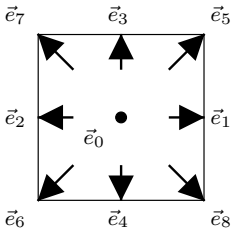


Figure 2.4: D2Q9 discretization lattice grid sites. Each arrow corresponds to a particle distribution function, and corresponds to nine potential movement directions. A particle population can either move to a neighboring site or remain in the current one.



$\vec{e}_0 = (\ 0,\ 0)$ $w_0 = 4/9$

$\vec{e}_1 = (\ 1,\ 0)$ $w_1 = 1/9$

$\vec{e}_2 = (-1,\ 0)$ $w_2 = 1/9$

$\vec{e}_3 = (\ 0,\ 1)$ $w_3 = 1/9$

$\vec{e}_4 = (\ 0, -1)$ $w_4 = 1/9$

$\vec{e}_5 = (\ 1,\ 1)$ $w_5 = 1/36$

$\vec{e}_6 = (-1, -1)$ $w_6 = 1/36$

$\vec{e}_7 = (-1,\ 1)$ $w_7 = 1/36$

$\vec{e}_8 = (\ 1, -1)$ $w_8 = 1/36$

Figure 2.5: D2Q9 lattice site. Direction vectors $\vec{e}_i$ are lattice velocities, with their corresponding weight w_i.

2.3.2 Discrete Lattice Boltzmann

The goal of LBM programs is to provide a numerical solution to the Boltzmann Equation, by an approximation method called the *Discrete Lattice Boltzmann Equation* which can be written as [4, pg.58]

$$f_i(\vec{x} + \vec{e_i}\Delta t, t + \Delta t) = f_i(\vec{x}, t) + \Gamma(f_i(\vec{x}, t)). \tag{2.9}$$

Therefore the basis of the LBM algorithm is the discretization of space and time into a lattice of suitable number of dimensions. The unique solution to the equation varies depending on these properties and the initial- and boundary-conditions of the problem domain.

Figures 2.4 and 2.5 show examples of a simple two dimensional lattice of square sites, where each lattice site has a set number of possible directions in which particle populations, or so called distribution functions, can move. The width of each site is exactly one lattice unit (Δx) in all directions on a square grid. For a specific problem domain, the conversion of length in meters is done by defining a physical length reference in meters L_{phys} and an equivalent length in number of lattice sites L_{lbm}. The conversion factor C_L from distance d in meters to number of lattice sites n is then

$$n = \frac{d}{C_L} = d \cdot \frac{L_{lbm}}{L_{phys}}. \tag{2.10}$$

A grid spacing Δx on a 3D lattice can be expressed from L_{phys} and the number of lattice sites in the domain $N = N_x \cdot N_y \cdot N_z$ with

$$\Delta x = \frac{L_{phys}}{\sqrt[3]{N}} \tag{2.11}$$

The conversion factor C_U from physical speed u in m/s to speed in lattice units is [4, p. 115]

$$U = \frac{u}{C_U} = u \cdot \frac{V_{lbm}}{V_{phys}}. \tag{2.12}$$

Time in the simulation domain is not continuous as in nature, but is measured in constant time steps Δt, so $t \in \Delta t n \mid n \in \mathbb{N}$. A time step is completed after all sites in the lattice have been updated once. This means that for each update, a constant time period of

$$\Delta t = \frac{C_L}{C_U} = \frac{L_{phys}}{C_U \sqrt[3]{N}} \tag{2.13}$$

seconds in simulated time has passed. Obviously, if Δt is equal to or greater than the time it took to compute the lattice update, the simulation can be performed in real-time or faster than real-time.

Each direction vector e_i seen in figure 2.5 is called a *lattice velocity* vector and is scaled such that during a time step Δt a particle can move exactly from one site to an adjacent one. When velocity is measured in lattice units per time steps ($\Delta x\ \Delta t^{-1}$) the magnitude of the lattice velocity is $\sqrt{2}\ \Delta x\ \Delta t^{-1}$ for diagonal lattice velocities and $1\ \Delta x\ \Delta t^{-1}$ otherwise. The particular type of lattice in the figures is called a D2Q9 discretization, which corresponds to the number of dimensions and lattice velocities respectively. Figure 2.6 shows a three dimensional lattice site with 19 lattice velocities.

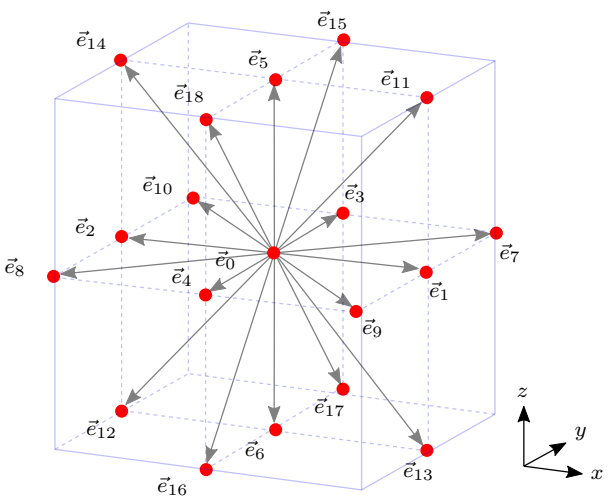
2.3.3 The LBM Algorithm

The collision operator Γ in the Boltzmann Equation (equation 2.8) can be implemented in multiple ways, the simplest being Bhatnagar–Gross–Krook (BGK) [15, pg.35]. Another common collision model is called *multiple-relaxation-time* (MRT). It calculates collision in terms of velocity moments instead of distribution functions and has superior accuracy and stability compared to BGK [4, pg.25]. However, since the RAFSINE application studied in this thesis uses the BGK model, the workings of MRT is outside the scope of this thesis.

In BGK the collisions of particle velocity distribution functions f_i are modeled by

$$f_i(\vec{x} + \vec{e}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t)}{\tau}. \tag{2.14}$$

Like the Boltzmann Equation, it contains a streaming part, represented by $f_i(\vec{x}+\vec{e}_i\Delta t, t+\Delta t) = f_i(\vec{x}, t)$ and a collision term $(f_i(\vec{x}, t) - f_i^{eq}(\vec{x}, t))/\tau$, where $\vec{x}$ is the particle displacement. The function f_i^{eq} is called the *equilibrium distribution function* and is defined



$\vec{e}_0$	= (0, 0, 0)	$w_0 = 1/3$
$\vec{e}_1$	= (1, 0, 0)	$w_1 = 1/18$
$\vec{e}_2$	= (-1, 0, 0)	$w_2 = 1/18$
$\vec{e}_3$	= (0, 1, 0)	$w_3 = 1/18$
$\vec{e}_4$	= (0, -1, 0)	$w_4 = 1/18$
$\vec{e}_5$	= (0, 0, 1)	$w_5 = 1/18$
$\vec{e}_6$	= (0, 0, -1)	$w_6 = 1/18$
$\vec{e}_7$	= (1, 1, 0)	$w_7 = 1/36$
$\vec{e}_8$	= (-1, -1, 0)	$w_8 = 1/36$
$\vec{e}_9$	= (1, -1, 0)	$w_9 = 1/36$
$\vec{e}_{10}$	= (-1, 1, 0)	$w_{10} = 1/36$
$\vec{e}_{11}$	= (1, 0, 1)	$w_{11} = 1/36$
$\vec{e}_{12}$	= (-1, 0, -1)	$w_{12} = 1/36$
$\vec{e}_{13}$	= (1, 0, -1)	$w_{13} = 1/36$
$\vec{e}_{14}$	= (-1, 0, 1)	$w_{14} = 1/36$
$\vec{e}_{15}$	= (0, 1, 1)	$w_{15} = 1/36$
$\vec{e}_{16}$	= (0, -1, -1)	$w_{16} = 1/36$
$\vec{e}_{17}$	= (0, 1, -1)	$w_{17} = 1/36$
$\vec{e}_{18}$	= (0, -1, 1)	$w_{18} = 1/36$

Figure 2.6 & Table 2.1: D3Q19 lattice site with its lattice velocities $\vec{e}_i$ and weights w_i.

as [15, pg.35]

$$f_i^{eq}(\vec{x}) = w_i \rho(\vec{x}) \left(1 + \frac{\vec{e}_i \cdot \vec{u}}{c_s^2} + \frac{(\vec{e}_i \cdot \vec{u})^2}{2c_s^4} - \frac{\vec{u}^2}{2c_s^2}\right), \tag{2.15}$$

where the vector product is defined as the inner product. Figure 2.11 shows the lattice velocities $\vec{e}_i$ and corresponding weights w_i for a D2Q9 lattice type, and $c_s = \frac{1}{\sqrt{3}}$ is the speed of sound on the lattice.

In equation 2.14 the distribution function f_i is relaxed towards the equilibrium function f_i^{eq} at a collision frequency of $1/\tau$. The relaxation time τ is chosen to correspond to the correct kinematic viscosity ν of the fluid in question. In the real world, it expresses the ratio of dynamic viscosity to the density of the fluid and is measured in $\mathrm{m^2 s^{-1}}$. For a D2Q9 model

$$\nu = \frac{1}{3}\left(\tau - \frac{1}{2}\right) \tag{2.16}$$

and is measured in $\Delta x^2\ \Delta t^{-1}$ [15, pg.39].

The density of the cell can be calculated as the sum of the velocity distribution functions,

$$\rho = \sum_i f_i, \tag{2.17}$$

and momemtum can be similarly calculated by multiplication with the lattice velocities

$$\vec{p} = \rho\vec{u} = \sum_i f_i \vec{e}_i. \tag{2.18}$$

Temperature T of a lattice site can be calculated as the second order moment [4, pg.40]

$$\rho e = \frac{1}{2}\sum_i (\vec{e}_i - \vec{u})^2 f_i, \tag{2.19}$$

$$e = \frac{3k}{2mT}, \tag{2.20}$$

where e is internal energy, k is called *Boltzmann factor* and m is mass, but this is inaccurate because of discretization errors. Instead, the temperature of a lattice site is stored in an independent set of *temperature distribution functions* T_i. The evolution of temperature in the system can be modeled using a smaller set of distribution functions, such as D2Q4 for a two dimensional domain, or (as was done in this thesis project) a D3Q6 lattice in three dimensions. In BGK, this is modeled by the equation

$$T_i(\vec{x} + \vec{e}_i\Delta t, t + \Delta t) = T_i(\vec{x}, t) - \frac{T_i(\vec{x}, t) - T_i^{eq}(\vec{x}, t)}{\tau_T}, \tag{2.21}$$

and the equilibrium distribution functions

$$T_i^{eq}(\vec{x}) = \frac{T}{b}\left(1 + \frac{b}{2}\vec{e}_i \cdot \vec{u}\right), \tag{2.22}$$

where $b = 7$ for a D3Q6 lattice. Relaxation time τ_T is related to thermal diffusivity

$$\alpha = \frac{2\tau_T - 1}{4} \cdot \frac{\Delta x^2}{\Delta t}, \tag{2.23}$$

and the temperature of a site can then recovered as [4, pg.41]

$$T = \sum_i T_i. \tag{2.24}$$

When under the effects of a gravity field, temperature differences between regions of particles in a fluid affect their velocity and displacement by what is known as *natural convection*.

2.3.4 Natural Convection

When a hot object is placed in a colder environment the temperature of the air surrounding the object will increase because of heat exchange. Since hot air has a lower density than cold air, it will start to rise and colder air will flow in to replace it. This phenomenon is known as *natural convection*. In its absence heat would only be transferred by *conduction*, which is a much slower process, or by *forced convection* from for example a fan blowing air on the object. When a gravitational field acts on the hot air it creates a force which pushes the hot air upwards. This is called the *buoyancy force*.

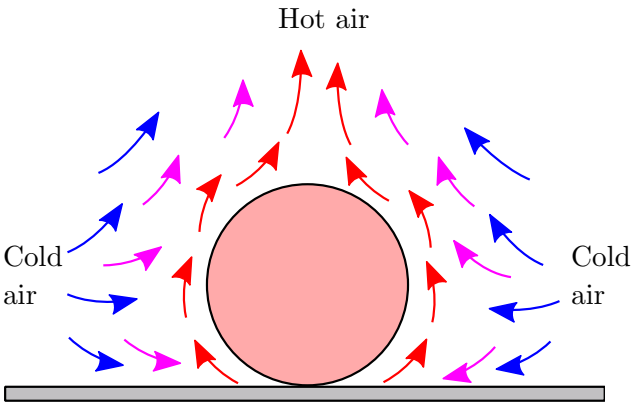


Figure 2.7: Heat and velocity flow around a sphere due to natural convection.

In fluid dynamics, natural convection can be modeled by the *Boussinesq approximation*. It assumes that variations in density have no effect on the flow field, except that they give rise to buoyancy forces. It is typically used to model liquids around room temperature such as natural ventilation in buildings [7]. The Boussinesq force can be defined as [4, pg.144]

$$\vec{F}_B = -\vec{g}\beta(T - T_0), \tag{2.25}$$

where $\vec{g}$ is the force due to gravity, $(T - T_0)$ is the thermal gradient between hot and cold fluid and β is its thermal expansion coefficient at the reference temperature T_0.

The effect of this term is included in RAFSINE by distributing it in the collision step to the two velocity distribution functions corresponding to up and down (see figure 2.6),

$$f_5(\vec{x}, t + \Delta t) = f_5^{temp}(\vec{x}, t) - \frac{f_5^{temp}(\vec{x}, t) - f_5^{eq}(\vec{x}, t)}{\tau} + \frac{\vec{g}\beta(T - T_0)}{2}, \tag{2.26}$$

$$f_6(\vec{x}, t + \Delta t) = f_6^{temp}(\vec{x}, t) - \frac{f_6^{temp}(\vec{x}, t) - f_6^{eq}(\vec{x}, t)}{\tau} - \frac{\vec{g}\beta(T - T_0)}{2}. \tag{2.27}$$

2.3.5 Turbulence modeling

As mentioned in section 2.1, a common way to model turbulence in CFD is using the RANS equations. One such model is called k – ϵ and uses two transport equations for turbulent flows, one for turbulent kinetic energy (k), and another for turbulent energy dissipation (ϵ). Since LBM is a time dependent simulation this model cannot be used directly in its RANS based time-averaged form, but adaptations for LBM exist [14].

The turbulence model used in RAFSINE is called Large Eddy Simulation (LES) and is based on the idea of ignoring the dynamics of small scale swirling motion of fluids (eddies) since large scale eddies carry more energy and contribute more to fluid motion transport. This is achieved by applying a low-pass filter to the Navier-Stokes equations.

In LBM BGK models, the filter width is that of a lattice site, which is often chosen as unity $\Delta x = 1$ [4, pg.51].

Energy created by stress from turbulence is defined in LES by the local momentum stress tensor $\bar{S}_{\alpha\beta}$. This tensor defines the flux of the αth component of the momentum vector $\vec{u}_\alpha$ across a surface with the constant coordinate x_β. For a lattice site with q lattice velocity vectors $\vec{e}_i$, the stress tensor is calculated by

$$\bar{S}_{\alpha\beta} = \frac{1}{2}\left(\frac{\partial \vec{u}_\alpha}{\partial x_\beta} + \frac{\partial \vec{u}_\beta}{\partial x_\alpha}\right) = \sum_{i=1}^{q} \vec{e}_{i\alpha}\vec{e}_{i\beta}\left(f_i - f_i^{eq}\right), \tag{2.28}$$

where f_i^{eq} are the equilibrium distribution functions as defined in equation 2.15, and f_i are the non-equilibrium ones. The local momentum stress tensor can then be used to calculate eddy viscosity as

$$\nu_t = \frac{1}{6}\left(\sqrt{\nu^2 + 18C_S^2(\Delta x)^2\sqrt{\bar{S}_{\alpha\beta}\bar{S}_{\alpha\beta}}}\right), \tag{2.29}$$

where ν is the kinematic viscosity and $C_S > 0$ is called the *Smagorinsky constant*. The turbulence is then implemented in the LBM simulation by letting the relaxation time τ in the BGK model (equation 2.14 and 2.16) vary locally in space for each lattice site, by

$$\tau = \frac{1}{2} + 3(\nu_0 + \nu_t), \tag{2.30}$$

where ν_0 is the kinematic viscosity of the fluid [4, pg.51].

For a LBM lattice of size N, the Reynolds number (see chapter 2.1.2) can be computed from

$$\mathrm{Re} = \frac{\vec{U}_{lbm}N}{\nu_{lbm}} \tag{2.31}$$

where $\vec{U}_{lbm}$ is the mean fluid velocity and ν_{lbm} is the viscosity, both defined in lattice units [4, pg.118].

2.3.6 Initialization Step

The simulation of fluid flow in LBM begins at time $t = 0$ with the *initialization step*, where initial conditions are set, usually to a zero velocity state. Formally, the initialization can be written as [4, pg.58]

$$f_i(\vec{x}, 0) = f_i^{eq}(\rho(\vec{x}), \vec{u}(\vec{x})), \tag{2.32}$$

where ρ is the initial pressure and $\vec{u}$ is initial velocity.

This happens only once at the start of simulation, followed by a repeating sequence of the *streaming*, *collision* and *boundary steps*. Each sequence of these three steps progresses time by Δt so that the total simulation time is $N\Delta t$ where N is the number of repetitions.

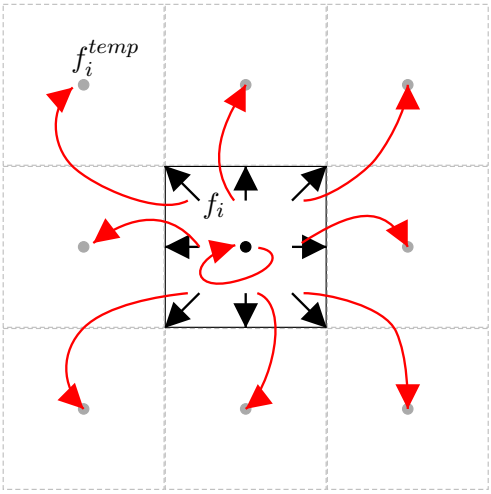


Figure 2.8: Lattice streaming step, representing the transport of distribution functions to neighboring sites. All functions are copied to the neighboring sites in a parallel fashion.

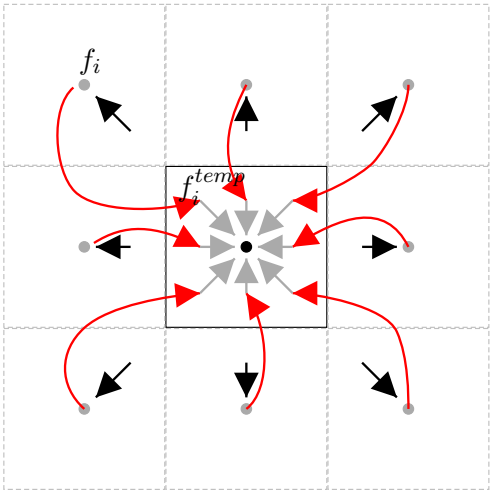


Figure 2.9: Also in the streaming step, the current site is filled with new distributions from the neighboring sites.

2.3.7 Streaming Step

The *streaming step* models transport of heat and mass in a fluid flow by motion, in a process called *advection*. Firstly, the distribution functions of a lattice site are copied into adjacent sites, or remain in the current one, depending on their lattice velocities. This is done in a parallel fashion so that each site both distributes to and receives from distribution functions in neighboring sites. The streamed functions are stored in temporary arrays f_i^{temp} which are used in the next step. Figures 2.8 and 2.9 illustrates the streaming step, which can be written as the left hand part of equation 2.14,

$$f_i^{temp}(\vec{x} + \vec{e_i}, t) = f_i(\vec{x}, t). \tag{2.33}$$

2.3.8 Collision Step

Next, the *collision step*, seen in figures 2.10 and 2.11, models how the collective motion of particles are affected by the previous advection. This process is also known as *diffusion*.

Newly arrived particle distribution functions are redistributed such that their mass and momentum is conserved. This operation is performed individually for each site, since all relevant data, f_i^{temp}, is localized to it from the streaming step, which makes it a highly parallelizable operation. The BGK collision model can be written as [4, pg.60]

$$f_i(\vec{x}, t + \Delta t) = f_i^{temp}(\vec{x}, t) - \frac{f_i^{temp}(\vec{x}, t) - f_i^{eq}(\vec{x}, t)}{\tau}. \tag{2.34}$$

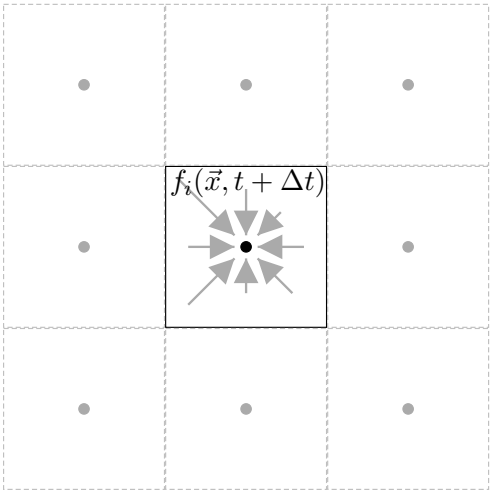


Figure 2.10: Collide step in a D2Q9 lattice. Particles from adjacent sites collide locally in the current site.

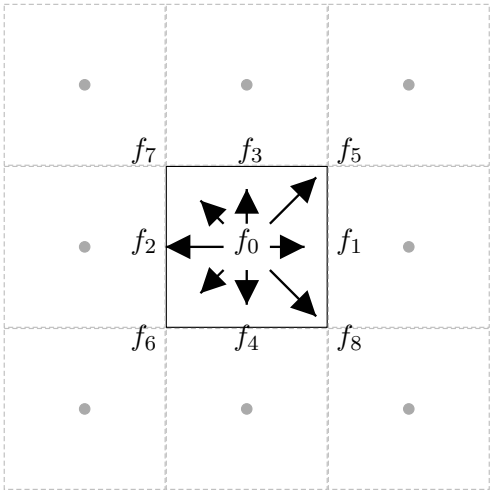


Figure 2.11: During the collide step the particle populations are redistributed. Both mass and momentum is conserved.

For most of the simulation domain no further calculations are needed and the stream and collide steps could be repeated to progress the simulation. However, at the edges of the domain there is the need to define the behavior of particles that collide with a wall or leave the domain through for example an air exhaust. Additionally, the problem might state that new particles should enter the domain, such as from air inlets.

2.3.9 Boundary Step

The final step is called the *boundary step*, and models the constraints of the simulation from the problem boundary conditions. There are many different types of boundary conditions suited to solve different fluid flow problems. One of the simplest is the *periodic* boundary condition, where distribution functions leaving the edge of the domain are streamed back to the opposite side. For a two dimensional plane periodic in all directions, this configuration can be visualized as a torus shape. It can be used to approximate an infinite domain [4, pg.63].

Bounce-Back Boundaries

Another simple boundary condition is called *bounce-back* and basically works by taking distribution functions leaving the domain, and inverting their direction so that $\vec{e_i} = -\vec{e_{\bar{i}}}$ and

$$f_i(\vec{x}) = f_{\bar{i}}(\vec{x}). \tag{2.35}$$

This is also called a *no-slip condition* and assumes that at a solid boundary, the fluid will have zero velocity relative to the solid. In other words, the adhesive force between

solid and fluid is stronger than the cohesive force in the fluid. This type of boundary condition is also called a Dirichlet condition, which specifies the values of a solution at a domain boundary (as opposed to e.g. the derivative of a solution). Bounce-back can either be implemented in what is called full-way or half-way schemes.

In the *full-way bounce-back* scheme, all distribution functions encountering this boundary condition are reverted in the opposite direction regardless of the plane normal of the boundary. The results of a collision takes place one time step after the collision, since the bounced-back distribution functions are stored in nodes on the other side of the boundary. This scheme is mostly useful when modeling a porous media where the normal of the boundary is not easily defined [4, pg.64] and is illustrated in figure 2.12.

The *half-way bounce-back* scheme is similar to full-way, except the reflection is calculated in the same time step as the collision occurs (see figure 2.13). It requires the normal of the boundary condition to be clearly defined, such as when modeling a solid wall.

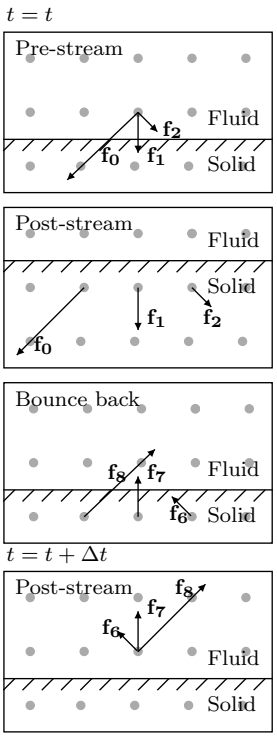


Figure 2.12: Full-way bounce-back boundary condition on a D2Q9 lattice (image source [4, pg.64]).

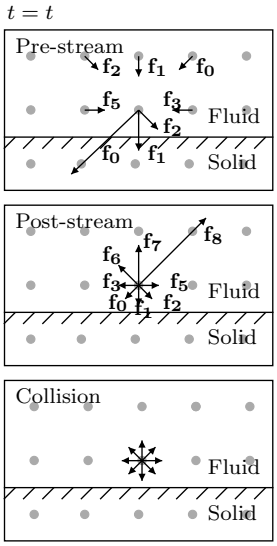


Figure 2.13: Half-way bounce-back boundary condition on a D2Q9 lattice (image source [4, pg.64]).

Von Neumann Boundaries

For modeling increase in fluid velocity and pressure, for example air passing through a server rack cooled by integrated fans, various types of *von Neumann boundary conditions* can be used. Von Neumann differs from Dirichlet boundary conditions by specifying the value of the derivative of a solution instead of the value of a solution directly. In the case of server rack fans it uses the particle velocity gradient in the direction normal to the boundary and increases particle momentum (and temperature in this case) by a set value. This results in a decreased pressure on the inlet side and and increase on the outlet side. It can also be used to model constant static pressure, such as from a heat exchange air cooler taking hot air out of the domain and introducing cold air into the domain.

The design of this type of boundary condition varies depending on what it is meant to model. Chapter 5.1 details how they were used when modeling a data center.

2.4 RAFSINE

RAFSINE is a CFD application which implements LBM using the Graphics Processing Unit (GPU) parallel processing toolkit NVIDIA CUDA (CUDA). It was written by Nicolas Delbosc during his Ph.D study in the School of Mechanical Engineering at the University of Leeds, England.

CUDA is an Application Programming Interface (API) which allows programmers to utilize NVIDIA GPUs to perform massively parallel general purpose calculations instead of graphics rendering. The program makes use of the highly parallelizable nature of the LBM algorithm to perform the *streaming*, *collision* and *boundary* steps calculations concurrently for a large number of lattice nodes.

When executed on the gaming-oriented graphics card NVIDIA GeForce GTX 1080 Ti, the parallel execution of the LBM simulation program allows simulations to be performed in real-time or faster than real-time depending on the size of the domain and complexity of boundary conditions. According to a benchmark between CFD softwares performed by the original author, in which the temperatures inside a small data center were simulated, the COMSOL CFD package took 14.7 hours to converge on a solution, while RAFSINE had a convergence time of 5 minutes [4, pg.168]. The real-time execution of the simulation could open up the possibility of integration into a cooling unit control system, which could use the LBM model to optimize the temperature and air flow rates. This is not possible with CFD software which uses the various finite method such as FVM because of a much slower convergence time.

2.4.1 GPU Programming

Computer programs running on a modern UNIX operating system is generally seen as a single *process*. Processes are composed of one or more *threads*, which is a sequence of programming instructions independently managed by a thread scheduler. The scheduler is part of the operating system and responsible for allocating memory and executing the threads. On a Central Processing Unit (CPU) with multiple processor cores, multiple threads can be executed simultaneously on each core, while processors with single cores generally handle multiple threads by time slicing (regularly switching between thread executions). At the time of writing, workstation CPUs generally have between 2-8 cores, while modern server hardware can have up to 32 cores.

In a modern computer, the CPU handles most tasks such as executing the operating system and scheduling the execution of applications using them, they also have an additional processing unit for rendering the graphics displayed on the screen. While CPUs are specialized at executing instructions in a sequential fashion at a very high frequency, while also having very fast control units for switching between different thread contexts, GPUs are specialized at executing a large number of threads in parallel (albeit at a lower rate of instructions per thread compared to the CPU). For comparison with a CPU, the NVIDIA GeForce GTX 1080 Ti has 28 Streaming Multiprocessors (SM), each capable of the parallel execution of a maximum of 2048 threads at any instance. A limitation of the GPU cores however is that it they cannot switch between tasks.

GPUs cannot function independently of CPUs, but rely on the CPU to feed it instructions. The execution of a CUDA program, called a *kernel*, is done in three steps. The data and instructions to be executed first has to be sent by the CPU to the GPU (also called *host* and *device* respectively in CUDA terms) through the PCI-Express bus. The GPU receives the instructions, executes them, then sends the results back to the CPU.

2.4.2 RAFSINE LBM Implementation on GPU

The application code for running the LBM algorithm was implemented using the C++ programming language (C++) and the CUDA C language. Figure 2.14 shows the basic structure of the program.

At program initialization, the memory required for the distribution functions f and f^{temp} is allocated as arrays on the GPU. Since they hold the distribution functions for all lattice sites a 3D grid of size N using the D3Q19 model needs a length of $19N$ each. Their size also depend on which physical quantities they hold. For modeling air velocity, three floating point values are needed to represent the axis directions, while temperature needs only one scalar. The arrays are initialized to some preset value such as the average room temperature in a data center.

After initialization, the program enters the simulation main loop, which consists of a CUDA kernel which performs both the streaming, collision and boundary steps. Figure 2.14 shows them as separate kernels for clarity. Integrated into the CUDA kernel are

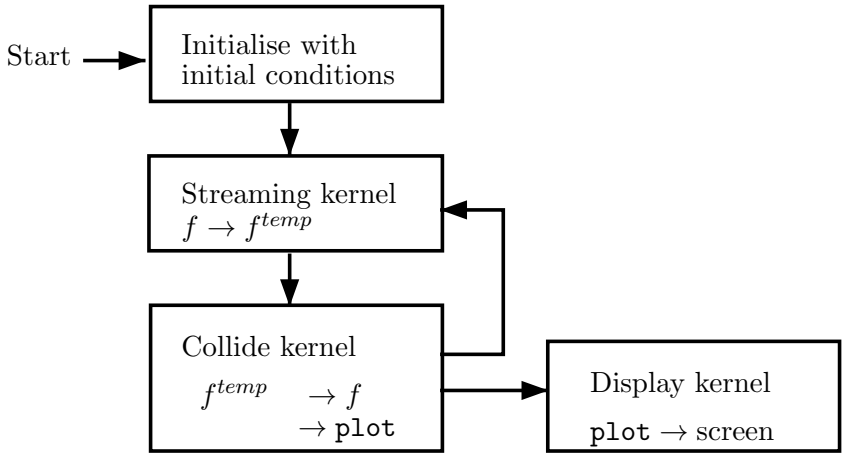


Figure 2.14: Structure of the LBM program (image source [4, pg.79]).

also routines for creating the OpenGL visualization. It consists of an array (called `plot`) in which values to be visualized are written depending on which physical quantity is to be displayed.

The `plot` array, which is stored on GPU memory, can be used directly by another CUDA kernel which generates a texture mapping representing the values. This texture can then be rendered in 3D space using OpenGL.

The original code included a few example simulations, one of which modeled the heat flow in a data center module with so called *cold aisle containment*. A screen capture image of this model can be seen in figure 2.15, showing temperature in vertical slices along each axis. These slices can be moved using keyboard input to examine different regions of the simulation lattice.

In order to change the simulation boundary conditions to represent different thermal models, RAFSINE uses C++ code generation with the Lua scripting language (Lua). Lua code describing the model geometry, physical quantities and boundary conditions are executed at compile time, which generates C++ header libraries. These libraries are linked into the main source code, allowing the compiler to optimize the generated code as best it can. Chapter 5.1 describes how this code generation was used to simulate another data center and perform model validation.

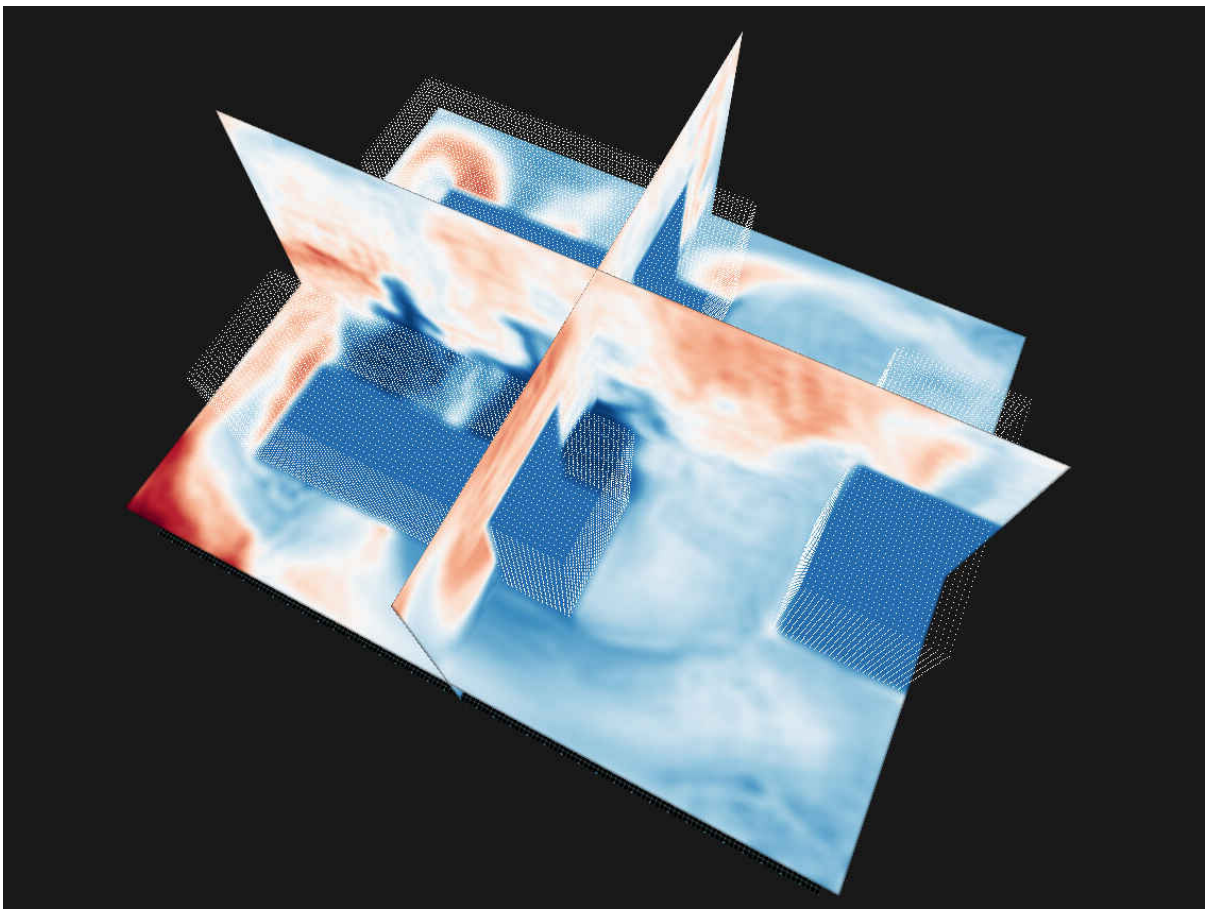


Figure 2.15: OpenGL visualization of an example CFD problem in RAFSINE, featuring heat flow in a data center with cold aisle containment.

2.5 Related works

As mentioned in chapter 2.4.2, the original author of the RAFSINE program included a simulation model of a small data center using cold aisle containment as a way of demonstrating how its LBM implementation could be applied to solve actual CFD problems [4, pg.163]. A graphical representation of this model can be seen in figure 2.15. The model was validated using other CFD applications such as COMSOL and OpenFOAM [4, pg.170], which showed the accuracy of RAFSINE was comparable to other solutions for this particular problem. However, this model was never validated against actual temperature measurements in a real-world data center.

In 2018, Emilie Wibron at Luleå University of Technology created CFD models of another data center module at RISE SICS North using the commercial CFD software ANSYS CFX. This work set out to investigate different configurations of air conditioning equipment, the performance of different turbulence models in ANSYS CFX, and the accuracy of these models by comparing them to experimental measurements. The results were published in the licentiate thesis *A Numerical and Experimental Study of Airflow in Data Centers*[16]. While the validation of these models showed an accuracy for temperatures within a few degrees [16, pg.32], the simulation did not take transient power usage of the servers into account. Similarly, the air conditioning was set to output a constant temperature and volumetric air flow. It should also be mentioned that ANSYS CFX does not have the capability of performing CFD simulations in real-time.

As for the evaluation of LBM as a general CFD solver, many validations have previously been performed. One example is the 2016 paper by Tamás István Józsa et al. called *Validation and Verification of a 2D Lattice Boltzmann Solver for Incompressible Fluid Flow*[8], in which the authors used their own CUDA LBM application to compare a general fluid channel flow with an analytical solution. They also simulated the so called lid-driven cavity problem in LBM and compared it to other CFD packages. In addition, they simulated the fluid flow over a step geometry and validated it with experimental data.

The specific problem of performing real-time CFD simulations of indoor thermal flows in data centers and validating them against an experimental setup does not seem to have been addressed in the literature studied during this thesis project.

Chapter 3

Remote Server Deployment

This chapter describes the process of deploying the RAFSINE program onto the remote GPU enabled headless servers at RISE SICS North. Section 3.1 describes how the CMake build system was used to automate compilation and linking of the code. The steps taken to perform hardware accelerated visualization using OpenGL on the servers over remote access systems is described in section 3.2.

3.1 CMake Build System

The original RAFSINE source code was written using a combination of C++, CUDA and Lua. The program was compiled using a Makefile, basically a shell script which performs code compilation and linking when executed. In this Makefile, file system paths were specified to shared dynamic libraries such as the OpenGL toolkit, X Window System (X11), OpenGL Extension Wrangler Library (GLEW), and the Joint Photographic Experts Group (JPEG) image codecs.

While Makefiles ease the code compilation step by automatically running the correct sequence of commands to compiler and linker, they can be hard to deploy on other platforms than the one for which they were developed. For example, different versions and distributions of operating systems can specify different file system paths for dynamic libraries, which the Makefile build system must account for.

This is a common problem in software development, and many alternative build systems have been developed, such as Waf, Gradle, Scons, QMake and CMake. These systems can automate the process of specifying file system paths for compiling a software source code. For this project, the CMake[1] build system was chosen, since it had native support for all the languages and dynamic libraries used by the RAFSINE source code, as well as having a large user base and documentation.

[1] https://www.cmake.org/

CMake is an open-source, cross-platform software build tool which has the ability to automatically search the file system for common dynamic libraries for linking into the compiled binary, and also set default linker flags to the compiler based on which libraries are used [2].

Another feature of CMake is the ability to run shell script commands, which in this project could be used to execute the CUDA code generation through Lua scripts. This made it possible to automatically execute the code generation of the simulation environment, such as the data center model described in chapter 5.

3.2 Remote OpenGL Visualization through VirtualGL

While the CUDA framework used by RAFSINE did not need any graphics rendering capability to perform CFD processing, the user interactive part of the program, such as keyboard and mouse event handling and graphical image rendering required an X11 server with hardware accelerated OpenGL functionality.

Since the RAFSINE simulation program needed the ability to execute on remote GPU enabled headless servers, accessed through remote control systems such as SSH or VNC, certain steps had to be taken to make use of the user interactive part of the program.

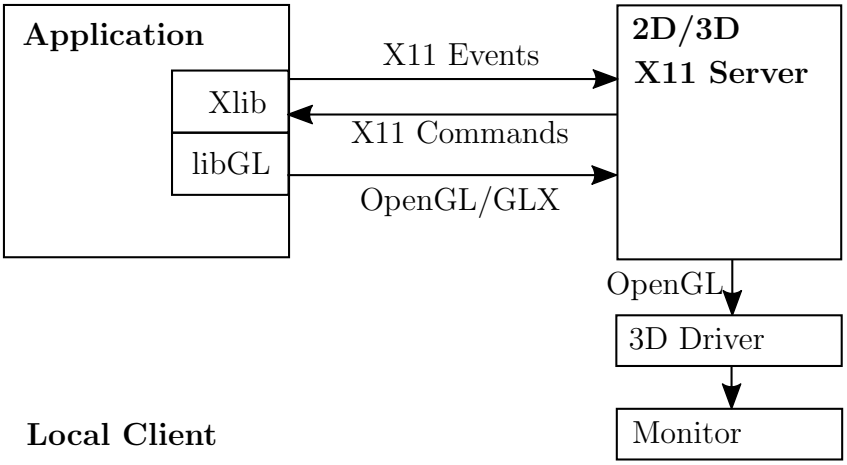


Figure 3.1: Direct OpenGL rendering using OpenGL extension to the X Window System (GLX) on a local GPU with a monitor attached.[2]

On a local UNIX machine with a monitor attached to the GPU, an OpenGL based

[2] Derived from the VirtualGL Project, https://virtualgl.org/About/Background, under the terms of the Creative Commons Attribution 2.5 License.

program can access the graphics rendering context used by X11 through the LibGL library. This library implements the GLX interface, which means it provides an interface between OpenGL and the X11 server.

When an application wants to draw 3D graphics inside a window, LibGL loads the appropriate 3D driver, in this case the NVIDIA GPU driver, and dispatches the OpenGL rendering calls to that driver [5]. X11 events such as mouse and keyboard input from the user and X11 commands such as opening and closing windows and image update requests is handled by Xlib [6]. Figure 3.1 shows a schematic of this process. In this configuration, the application is allowed to directly access the video hardware, the GPU, through the Direct Rendering Interface (DRI), allowing for hardware accelerated graphics rendering. This process is called *direct rendering*.

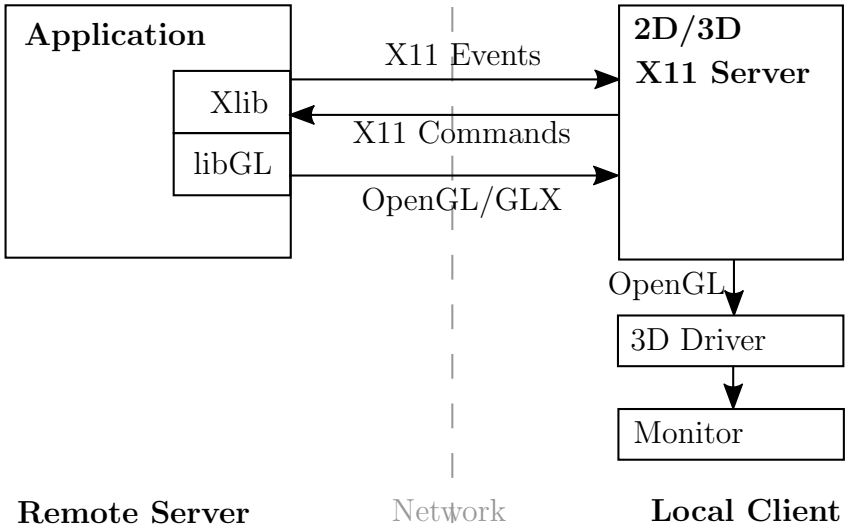


Figure 3.2: Indirect OpenGL rendering over a network using GLX.[3]

When an OpenGL application running on a remote headless server is accessed through a remote access system such as VNC or X11-forwarding through SSH, LibGL creates GLX protocol messages and sends them to the local client X11 server via a network socket. The local client then passes the messages on to the local 3D rendering system for rendering on a monitor [5]. The local 3D rendering may or may not be hardware accelerated. This process is called *indirect rendering*, and a schematic representation can be seen in figure 3.2.

There are two main problems with this approach. Firstly, in the case where the application (RAFSINE in this case) is executed through X11-forwarding and rendered on a local client, the problem is that some OpenGL extensions require that the application has

[3] Derived from the VirtualGL Project, https://virtualgl.org/About/Background, under the terms of the Creative Commons Attribution 2.5 License.

direct access to the GPU, and can thus never be made to work over a network. Secondly, 3D graphics data such as textures and large geometries can take up a relatively large amount of space, several megabytes in many cases. Since an interactive 3D application requires tens of frame updates per second to be free of lag, indirect rendering requires an extremely high bandwidth and latency [13].

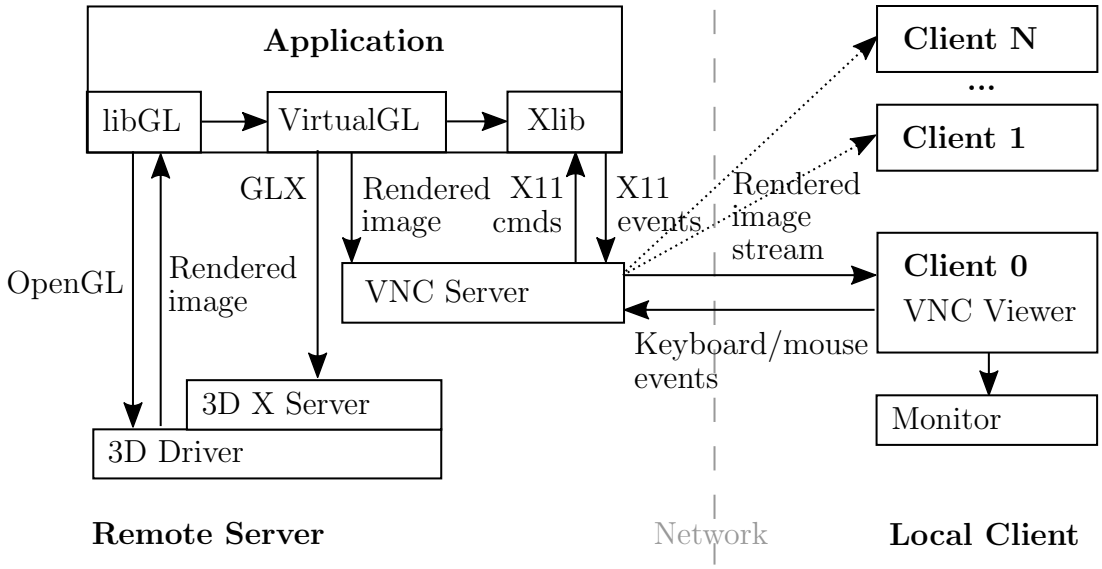


Figure 3.3: In-Process GLX Forking with an X11 proxy over a network, in the form of a VNC server and client.[4]

A solution to the problem of OpenGL rendering on a remote server can be found in the VirtualGL[5] software toolkit, which features two modes of operations to solve this problem. One solution is to introduce a GLX interposer which ensures OpenGL calls are directed to the server GPU, encodes the rendered 3D images inside of the server application process, and sends the them through a dedicated TCP socket to a VirtualGL client application running on the client machine. This network socket connection is called the *VGL Transport*. The client then decodes the images and draws them in the appropriate X11 window [13].

While this is a much more efficient solution than *indirect rendering*, and allows for seamless window integration in a UNIX client running a X11-forwarding through SSH, it requires the client to actually run an X11 server.

The other mode of operation is more cross-platform and can be made to work on Microsoft Windows machines through a VNC client. This mode is called *in-process GLX*

[4] Derived from the VirtualGL Project, https://virtualgl.org/About/Background, under the terms of the Creative Commons Attribution 2.5 License.

[5] https://www.virtualgl.org/

forking and also involves interposing application GLX calls and redirection to the server GPU. However, instead of using the *VGL Transport* stream, the rendered images can be sent to an X11-proxy such as a VNC server [13]. The local client can then connect to it using a VNC client software, such as TurboVNC[6] (which is built and optimized by the VirtualGL team for this purpose), or TigerVNC[7]. Figure 3.3 shows a schematic for this mode of operation.

For the work described in this thesis, the *in-process GLX forking* mode was chosen because of its compatibility with Windows client machines. For details about the VirtualGL installation process on the remote GPU enabled servers at RISE SICS North, see appendix A.1.

[6] https://www.turbovnc.org/
[7] https://www.tigervnc.org/

Chapter 4

Implementation

While the original RAFSINE application performed very well when running on a local system, certain code modifications had to be done in order to achieve the same performance on a remote system accessed by VirtualGL through VNC. These modifications mostly involved making the application multi-threaded and are described in chapter 4.1.

For the purpose of validating the result of a simulation, a table of measurement data was used. This table included the air temperatures and velocities measured during an experiment in the real world data center, as well as the server power usage and the rotational speeds of the integrated server cooling fans. In the original RAFSINE application, the last two conditions were modeled as constants, with no way of modifying them while a simulation was running. Since the data used to validate the model contained transient behavior for power usage and fan speeds, the code had to be modified to support changing simulation boundary conditions in real-time. Chapter 4.2 describes these changes.

4.1 Multithreading

The original code was written as a single-threaded application, where basically all code was executed in a single loop which performed the visualization, checked the timing of regularly scheduled tasks such as averaging, and displayed statistical outputs. The only thing which could interrupt this loop was handling user keyboard and mouse events. While this is a simple and effective way of executing simple applications, it does limit the utilization of modern multi-threaded CPUs.

One situation where single-threading was discovered to limit the performance of the RAFSINE application was in the graphical OpenGL visualization part. Even though VirtualGL allowed excellent performance for remote OpenGL rendering through VNC, it did introduce a certain overhead from the process of interposing GLX calls and transporting the rendered image to the VNC X11-proxy. When running on a local GPU, the time it took to render the OpenGL visualization was negligible, but when adding the

overhead from VirtualGL it was discovered that several CUDA kernel simulations steps could have been performed during this period. Since the application was single-threaded this meant that when rendering the OpenGL frames the CUDA simulation kernel could not be executed, even though this was theoretically possible. Figure 4.1 shows a time line representation of this situation. While a simulation step was being executed, multi-threading would allow the GPU to render the visualization of the previous simulation step.

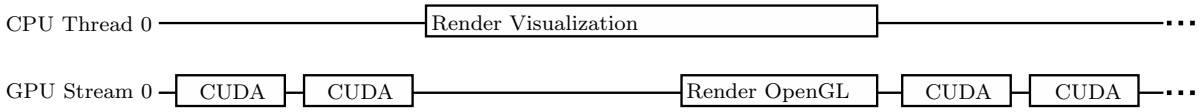


Figure 4.1: Single threading in the original RAFSINE application.

The application needed to run both the CUDA simulation and OpenGL rendering, perform regular calculations on measurements, while also being able to respond to user input. In order to facilitate these requirements in a multi-threaded context, the POSIX Threads (pthreads) support in the Linux operating system was used in conjunction with the threaded timer support in the Fast Light Toolkit (FLTK) library. pthreads is a programming interface which handles the creation and destruction of CPU threads, as well as inter-thread communication for signaling suspension and resuming of their execution. The FLTK library implements timers for scheduling future execution of specified code. These timers can expire at any point in the regular execution of the application code, triggering the execution of specified callback functions.

While threads on the CPU are used for concurrent execution of CPU code, CUDA applications manage concurrency by executing asynchronous GPU commands in *streams* [10]. This is accomplished by specifying an index number which is used as an identifier for a particular stream. Streams may execute their commands concurrently with each other and can perform memory copies of GPU device memory independent from each other. In the case of the RAFSINE application, different CUDA kernels were used for CFD simulation and OpenGL rendering, allowing concurrent simulation and rendering to be implemented.

A common problem when programming multi-threaded applications is to ensure memory which is shared between threads is handled in a correct way. For example, if one thread starts to manipulate memory and gets interrupted by another thread also accessing this memory there is the risk of the second thread using incorrect or incomplete data. This situation is called a *race-condition*, and the parts of the thread both accessing shared memory is called a *critical section*. A common way to prevent race-conditions is using the thread synchronization primitive called Mutual Exclusion (mutex), which

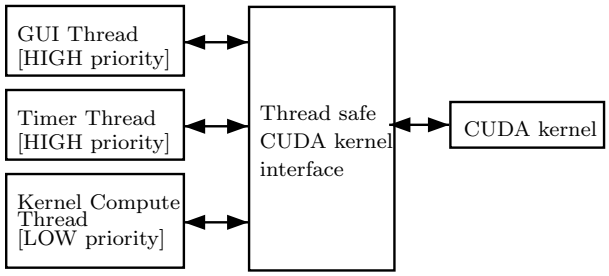


Figure 4.2: Thread-safe interface for communicating with the CUDA kernel from multiple CPU threads.

acts as a locking mechanism, allowing only one execution thread at a time to access the critical section. In the RAFSINE application, this type of synchronization was needed for communicating with the single CUDA kernel from multiple threads. A common communication interface was constructed using mutex locking for thread synchronization. Figure 4.2 shows a diagram of this model.

The goal of the multi-threading support was to allow the CUDA kernel to execute as often as possible, while also allowing the user to modify the execution parameters of it (such as simulation boundary conditions). A single CPU thread was constructed as an infinite loop, always trying to execute the kernel again as soon as the previous execution had finished. Through the common kernel interface, other threads could signal suspension and resuming of kernel execution as well as reading simulation data and setting simulation boundary conditions. Thread access to the kernel was protected by mutex locking to ensure no race conditions could occur.

One problem with this execution model however was that the C++ standard does not make any guarantees about the order in which mutex locked access to critical sections is granted. In this situation, it was discovered that only the infinite kernel execution loop was granted execution access to the kernel while all other thread had to wait indefinitely for access. This was solved by granting different threads privileged access through a relatively simple process using three different mutex locks, as seen in figure 4.3.

The CUDA kernel execution was scheduled as a low priority access thread running as often as possible, while threads for setting simulation boundary conditions and regularly reading simulation outputs were given high priority access. The low priority thread has to first acquire a low-priority access mutex L, then a next-to-access mutex N. After both these locks have been acquired, a final data mutex M has to be acquired, after which the N mutex can be released (allowing another thread to secure access after it). With M acquired, the memory in the critical section can be safely accessed, after which both the M and L mutex can be released in the order in which they were acquired. High-priority threads do not need to first access the low-priority lock L but can take N immediately.

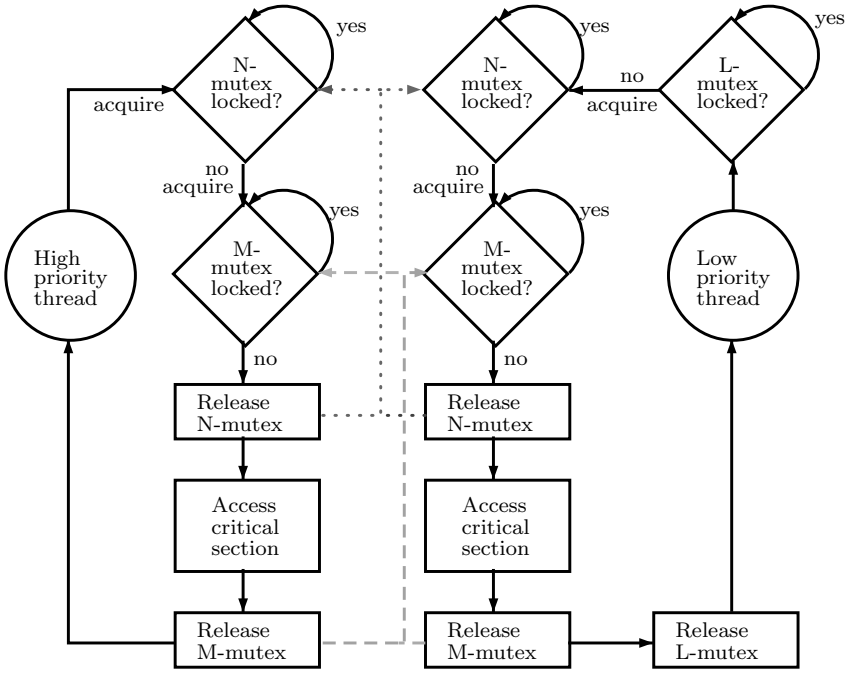


Figure 4.3: Simple thread priority scheme using triple mutex locking.

While this type of prioritized thread synchronization is simple to implement and has very low overhead, it does not support other priorities than high and low. If more priority levels were required, some sort of queue ordered by thread priority could be implemented using for example an array of function pointers.

With multi-threading implemented, the overhead from VirtualGL rendering could be completely eliminated with respect to the amount of CUDA kernel executions performed during a certain time period. The low priority simulation kernel execution thread tried to run its code in its dedicated GPU stream as often as possible. The CPU thread responsible for rendering the OpenGL visualization was set to copy the simulation output from this thread and stream when a new visualization frame needed to be drawn (according to a set frame rate). Copying was done asynchronously using device-to-device copy (between GPU memory banks) to a memory buffer, meaning the rendering could be done independent of simulation kernel execution. Of course, there was still a slight overhead from performing the visualization compared to not performing it. Disabling the visualization by minimizing or hiding the drawing window did slightly improve simulation performance.

Other CPU threads were used, one responsible for reading simulation output data such as averaged temperatures and velocities, while another was used to allow either time dependent or user event based manipulation of simulation boundary conditions. These functions required *device-to-host* and *host-to-device* memory transactions respectively. While it would be possible to use asynchronous memory copy for these functions, the

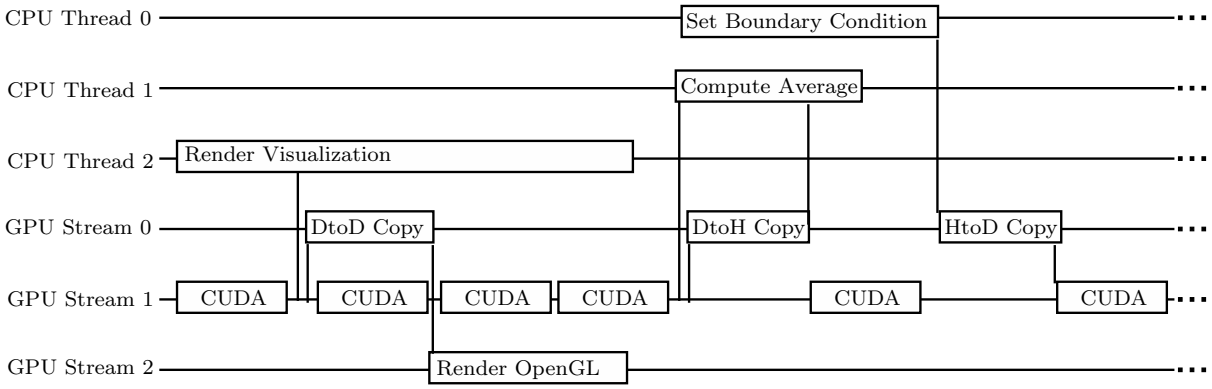


Figure 4.4: Example of CPU and GPU multi-threading in RAFSINE. The CUDA kernel is executed as often as possible (triggered by a CPU thread not displayed). The use of multi-threading allows OpenGL rendering to run concurrently with kernel execution by performing asynchronous *device-to-device* (DtoD) copying of visualization data from the kernel execution stream to an OpenGL rendering stream. *Device-to-host* (DtoH) copying and vice versa (HtoD) is synchronous in order to ensure correct execution order.

amount of memory copied was so small it was decided to use synchronous transfers in this case to simplify the programming. A schematic showing an example of these executions and memory transactions over a period of time can be seen in figure 4.4.

4.2 Real-time Boundary Condition Updates

As mentioned in chapter 2.4.2, RAFSINE used code generation techniques to define and compile the geometry and boundary conditions of the simulation domain. These properties were defined in a Lua script, which when executed produced C++ compatible code in the form of a source code file. This code could then be compiled into the CUDA kernel program using the C++ `#include` directive.

The source file contained conditional statements for how to calculate the temperature and velocity distribution functions, which the compiler could convert into machine code. This technique made the boundary condition evaluation very fast, but it also meant their definitions could not be changed during simulation runtime. To accommodate transient behavior such as increased server load and cooler air flow over a period of time, it was necessary to move the calculation of these from the code generation into the simulation kernel.

The types of boundary conditions modeling fluid inlets and exhausts required parameters for setting various properties, such as

- Numerical identifier of the lattice sites implementing the boundary condition.

- Plane normals of the lattice sites implementing the boundary condition.
- Type of boundary condition, e.g. inlet, exhaust.
- Desired fluid velocity.
- Temperature (constant or relative to other sites) in the case of exhausts.

While the memory footprint of these variables were quite small, using them as parameters for the CUDA simulation kernel meant that the GPU had to read them from the so called *global* memory, whereas before they were read from *register* memory. Although performance of different memory types varies between CUDA architectures and generations, according to the CUDA Programming Guide [3], global memory can more than 100 times slower than register memory in certain situations. A performance decrease was seen after this modification was made, and an important future work on the RAFSINE code would be to optimize this functionality for faster kernel execution.

In order to validate the changes to the kernel, a simple framework for reading the temperatures and flow rates set by the boundary conditions was added to the existing simulation model code generation. The framework calculated the lattice array indices for planes and lattice sites adjacent to the boundary conditions. This allowed different positions on the lattice to be sampled at regular intervals and display thermal conditions and volumetric flow rates as they were changed.

Chapter 5

Modeling a Data Center Module

This chapter describes how the simulation model of the experimental data center POD 2 at RISE SICS North was constructed using the Lua-to-CUDA code generation API built into RAFSINE.

The server racks in the data center were configured in a so called hot aisle configuration, so that the backside of each row of server racks were facing one another. As hot air gathered in the aisle, it pooled along the ceiling, after which it was drawn into the inlets of four Computer Room Air Conditioners (CRACs) along the sides of the room. The CRACs were configured to cool the air from inlets on top of them using heat exchange elements, and then blow the cold air out into the room. Figure 5.1 illustrates this setup.

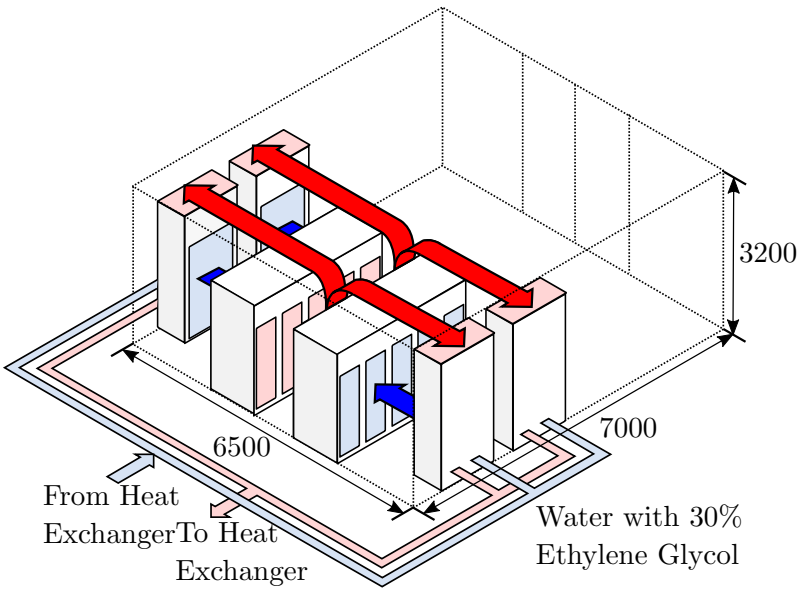


Figure 5.1: Theoretical heat flow in the data center POD 2 at RISE SICS North.

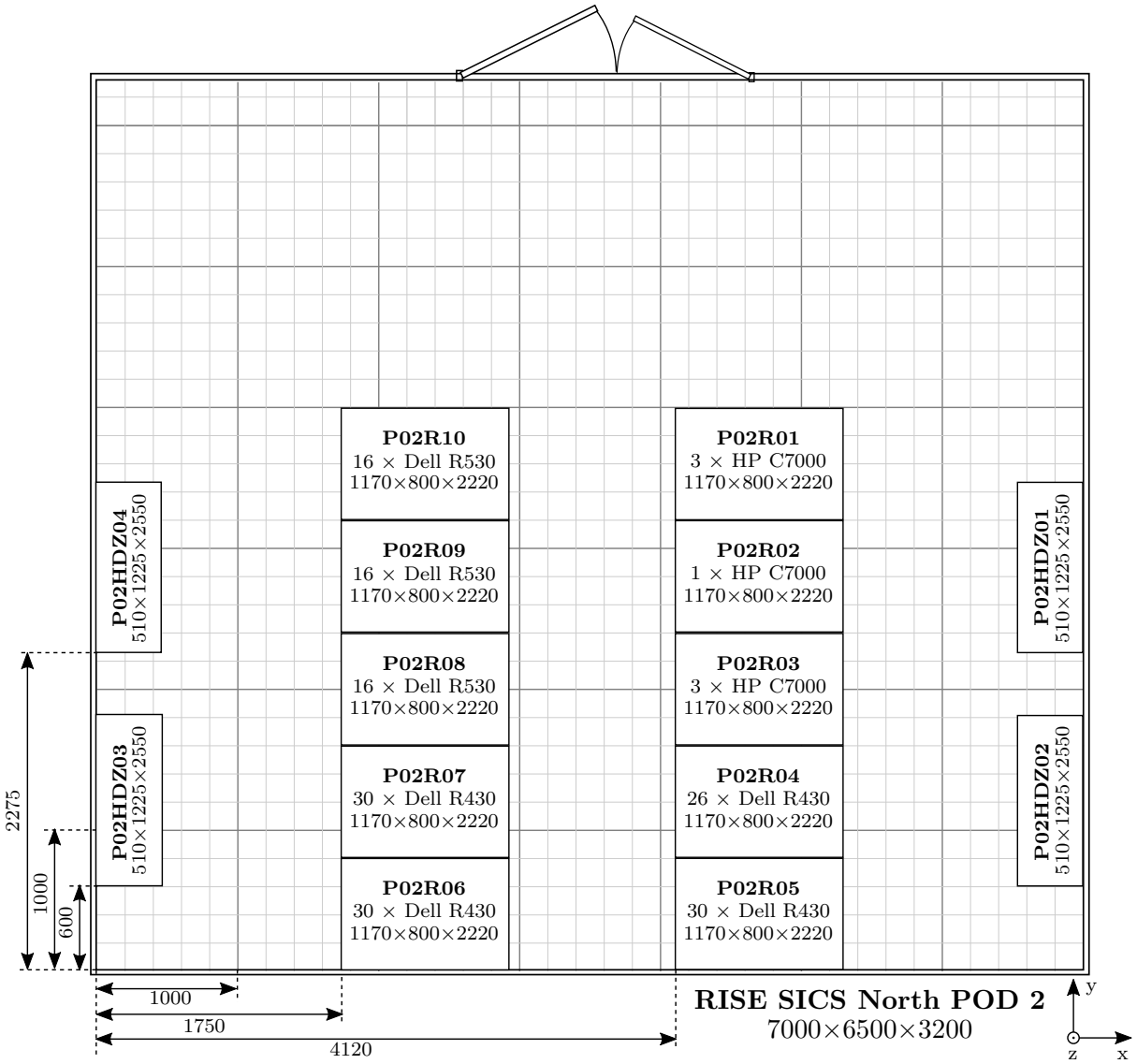


Figure 5.2: Schematic of the data center module POD 2 at RISE SICS North.

5.1 Data Center CFD Model

The basic geometry of the boundary conditions in RAFSINE were defined in a Lua script file, based on length measurements of the physical data center, as seen in figure 5.2. LBM is based on a discreet lattice of cells, and the program automatically discretized the lengths from meters into lattice units (lu).

For simplicity, the model used the same lattice types as the example simulation model included in the original RAFSINE code, which was D3Q19 for the velocity distribution functions and D3Q6 for the temperature.

Since the domain of LBM is based on a regular grid with Euclidian coordinates for each cell, modeling sloped surfaces is problematic unless a very high resolution is used. Figure 5.3 shows an approximation of how the CRACs were modeled. As can be seen, the geometry was simplified to remove slopes, while the areas of the topside inlet and sideways exhaust kept the same dimensions.

The floor, walls and ceiling of the room were modeled using the half-way bounce-back scheme explained in chapter 2.3.9, which basically defines zero air velocity along these boundaries. Likewise, the temperature distribution functions were also implemented with bounce-back which meant no heat transfer took place at the walls. The boundary conditions for the CRACs and servers required more specialized definitions.

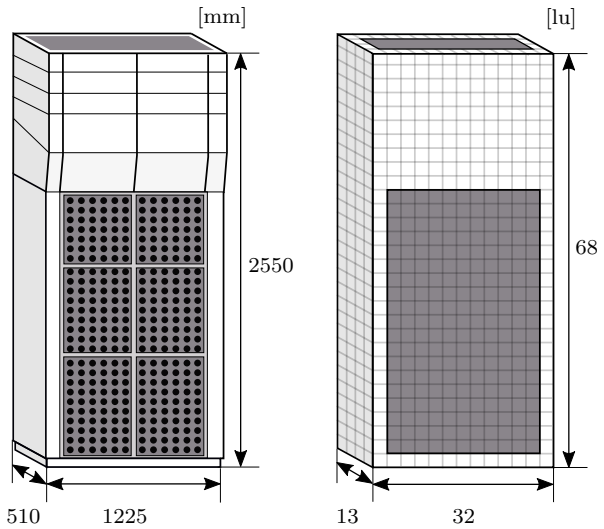


Figure 5.3: Heat exchange cooler SEE Cooler HDZ-2.

CRACs

Firstly, the air exhausts at the sides of the CRACs were set to blow cold air at a constant temperature T_{supply} and flow rate Q_{supply}. Secondly, the topside inlets were set to create a constant static pressure p_{return}, which meant setting the gradient of the velocity and temperature fields to zero, that is [4, pg.164]

$$\frac{\partial \vec{u}}{\partial z} = 0, \ \frac{\partial T}{\partial z} = 0. \tag{5.1}$$

This von Neumann boundary condition is called *zero-gradient*. ∂z is the gradient component corresponding to the normal of the inlet plane. Since these two boundary conditions modeled a circulating air flow, the inlet flow rate $Q_{return} = Q_{supply}$.

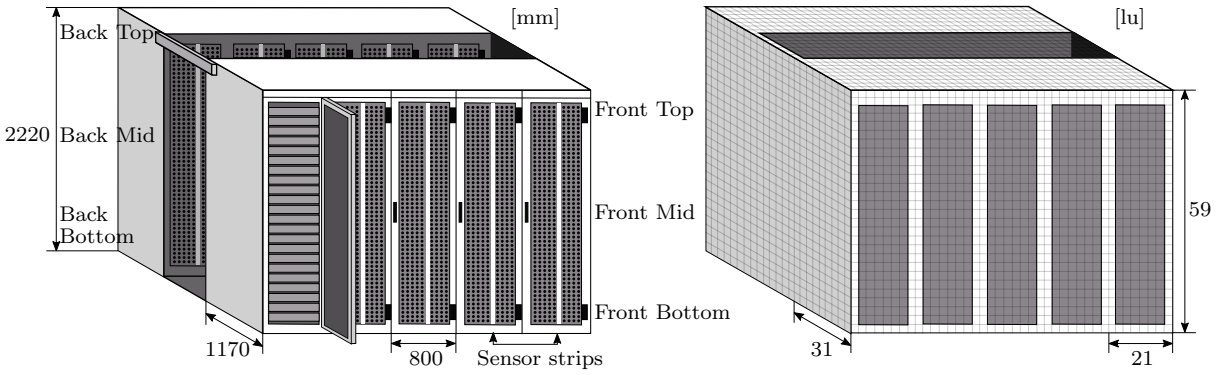


Figure 5.4: Two rows of five Minkels server racks positioned in hot aisle configuration. The entrance is separated by a curtain of flexible plastic sheets.

Server racks

Although in reality, each server rack contained between 16-30 servers, their power consumption was only measured on a per-rack basis. Since their individual power usages were unknown specific boundary conditions for each server could not be derived. Instead each rack was modeled using one inlet boundary condition on the front of the rack and one exhaust on the back. Figure 5.4 illustrates the modeling of the racks.

The air inlet on the front of the racks were modeled using a similar zero-gradient boundary condition as the CRAC inlets, with a constant flow rate Q_{in}.

Each server contained case mounted fans which provided cooling to the components depending on their temperatures. When the server components consumed more power and created more heat, the fans automatically increased their flow rates Q_{out}. The temperatures on the back of the server racks T_{out} was therefore based on the temperatures on the

front T_{in} plus a temperature increase ΔT which depended on server power consumption. This created the relations [4, pg.166]

$$T_{out} = \frac{\int T_{in} dA}{\int dA} + \Delta T, \tag{5.2}$$

$$\vec{u}_{out} = \frac{Q_{out}}{\int dA} \vec{n}, \tag{5.3}$$

which describes the boundary condition for the temperature and velocity distribution functions on the back of the racks. The first equation is the average of the integration of inlet air T_{in} over an area A, which in this case was simply the six temperature distribution functions of an inlet lattice cell in the D3Q6 model. The second shows the exhaust air velocity along the normal $\vec{n}$ of each cell on the server backside.

Temperature increase ΔT is affected by the server workload, which relates directly to its power consumption P (measured in kilowatts) and fan flow rate Q_{out}. This was modeled using the relation [4, pg.167]

$$\Delta T = \frac{P \cdot \nu}{Q_{out} \cdot k \cdot Pr}, \tag{5.4}$$

where the constants $\nu = 1.568 \cdot 10^{-5}$ m^2/s is the kinematic viscosity, $k = 2.624 \cdot 10^{-5}$ kW/m is the thermal conductivity and $Pr = 0.707$ is the Prandtl number of air at 30°C[1].

Figure 5.5 shows a screenshot of the geometry of the final data center model as implemented in RAFSINE. The ceiling was removed for better visibility but exists in the actual simulation model.

5.2 Simulation Input Data

As described in chapter 5.1, the data center LBM model had four unknown input parameters - the volumeteric flow rates and air exhaust temperatures for the four CRAC units, $Q_{supply,i}$ and $T_{supply,i}$ as well as server rack flow rate $Q_{out,j}$ and temperature increase ΔT_j for $i = 1, \ldots, 4$ and $j = 1, \ldots, 10$. Temperature increase depends on rack power consumption P_j.

To provide these values, sensor data recorded from an earlier experiment in POD 2 was used, which took place on the 23 of January 2018. The sampling rate of the data was once per minute, and extended a period of 36 hours starting at 9:00.

[1] The Engineering ToolBox: Dry Air Properties https://www.engineeringtoolbox.com/dry-air-properties-d_973.html

CRACs

The CRAC air flow Q_{supply} rate was measured using a mass flow sensor, which reported flow rate in the unit kg/s. Conversion to m^3/s was done using the density of air $\rho = 1.177$ kg/m^3 at 30°C and 1 atm pressure[1]. CRAC air supply temperature T_{supply} was simply measured by a thermometer in °C and required no conversion.

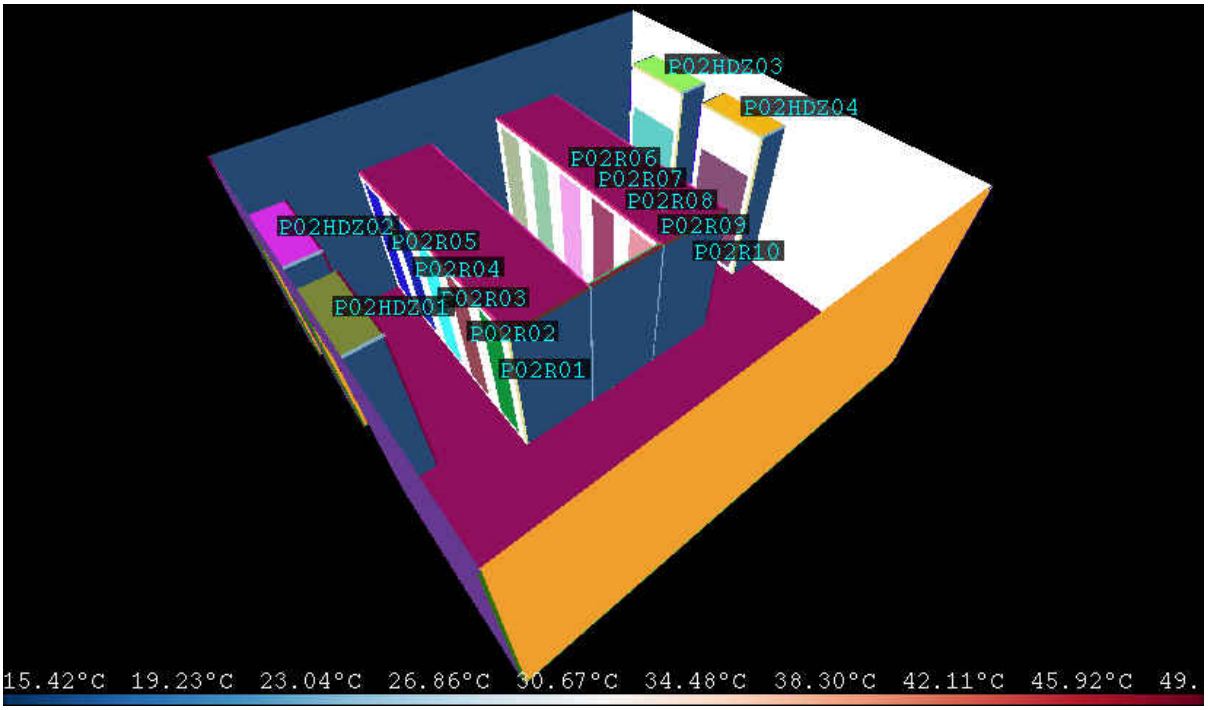


Figure 5.5: Geometry of data center module POD 2.

Server racks

The server racks were powered by a three-phase power supply, and the recording listed the power consumption of each phase for each rack. The total consumption for each rack P_j was calculated as the sum of the three phases.

Unfortunately, the recording contained no direct measurement of the air flow rate through the server racks, except the rotational speed of the integrated fans measured in rotations per minute (RPM). This meant the flow rate had to be approximated using specifications of the fans inside the racks. Table 5.1 shows the server equipment installed in each rack, while table 5.2 shows the specifications of the integrated fans.

According to the *affinity laws*, the power consumption of a fan is proportional to the

Table 5.1: Equipment of the server racks in data center POD 2.

Rack	**Equipment**	**Amount** $n_{servers}$
P02R01	HPC7000	3
P02R02	HPC7000	1
	Voltaire 4700	1
P02R03	HPC7000	3
P02R04	Dell R430	26
P02R05	Dell R430	30
P02R06	Dell R430	30
P02R07	Dell R430	30
P02R08	Dell R530	16
P02R09	Dell R530	16
P02R10	Dell R530	16

cube of the shaft speed. This means the ratio between maximum input power P_{max} and an operational point P_{op} is equal to the cube of ratio maximum fan speed ω_{max} over operational speed ω_{op}. Around this operating point, the power ratio can be assumed to be proportional to the volumetric flow rates Q_{max} and Q_{op}, so

$$\frac{P_{max}}{P_{op}} = \left(\frac{\omega_{max}}{\omega_{op}}\right)^3, \tag{5.5}$$

$$\frac{P_{max}}{P_{op}} = \frac{Q_{max}}{Q_{op}}. \tag{5.6}$$

Solving for Q_{op} yields

$$Q_{op} = Q_{max}\left(\frac{\omega_{op}}{\omega_{max}}\right)^3. \tag{5.7}$$

As seen in table 5.2, each Dell R430 and R530 server had $n_{fans} = 6$ integrated fans. While the experimental data log recorded the average fan speed of each individual server, this model simplified the boundary conditions for server air flow by using the average fan speed ω_{rack} in RPM of all servers in a rack. This meant the total flow rate in CFM for a server rack was

$$Q_{out} = \omega_{rack} \cdot n_{fans} \cdot n_{servers} \cdot \frac{Q_{op}}{\omega_{op}} \tag{5.8}$$

$$= \omega_{rack} \cdot n_{fans} \cdot n_{servers} \cdot Q_{max}\frac{(\omega_{op})^2}{(\omega_{max})^3}. \tag{5.9}$$

When calculating air flow for a Dell R430 server at $\approx 50\%$ speed, or 7500 RPM, the total flow $Q_{out} = 26$ CFM or 0.0123 m^3/s.

A more accurate model would implement a unique boundary condition for each server depending on its own fan speed. Even if the individual power consumption of each server is not recorded, the total consumption of the rack could be divided by the number of servers to get an average for each server.

Because the rotational speeds in racks 1–3 was not recorded, the fan specifications for these servers was not investigated. Instead, the flow rate through these servers was set to scale directly with power usage, by a factor approximated by trial and error.

Table 5.2: Fan specifications of the servers.

Equipment	**Fan type**	**Amount** n_{fans}	**Max power** P_{max} **[W]**	**Max speed** ω_{max} **[RPM]**	**Max flow** Q_{max} **[CFM]**
Dell R430	Delta Electronics GFB0412SHS-DF00	6	13.2	14300	30.23
Dell R530	Delta Electronics PFR0612DHE-SP00	6	19.2	14500	65.95
HP C7000	Unknown				
Voltaire 4700	Unknown				

5.3 Simulation Output Data

The fronts and backs of the server racks at the data center module POD 2 were fitted with temperature sensor strips which held three sensors at different heights above the floor. Figure 5.4 shows the positions of the sensors, which were fastened to the racks by magnets and connected to a sensor network so their readings could be recorded. The CRAC units contained integrated sensors which recorded the temperature of the air at the intake and exhaust.

In a similar fashion, the lattice sites corresponding to the positions of the three temperature sensors on the racks were sampled during simulation runtime. For measuring CRAC intake and exhaust temperatures, the average temperature of the lattice sites adjacent to the sites containing the boundaries were sampled. Figure 5.6 shows a schematic of how simulation input and output values were handled, while figure 5.7 shows a screenshot of RAFSINE during execution of the simulation. Blue areas correspond to a low

temperature, while red ones are hotter.

Readings in both the experimental data and simulation was averaged over one minute, after which they were recorded in a file using the Comma-Separated Values (CSV) format. Figures B.3 to B.6 shows a comparison between the simulated temperatures and the experimental data for the CRACs, while B.7 to B.16 shows the temperatures of the racks.

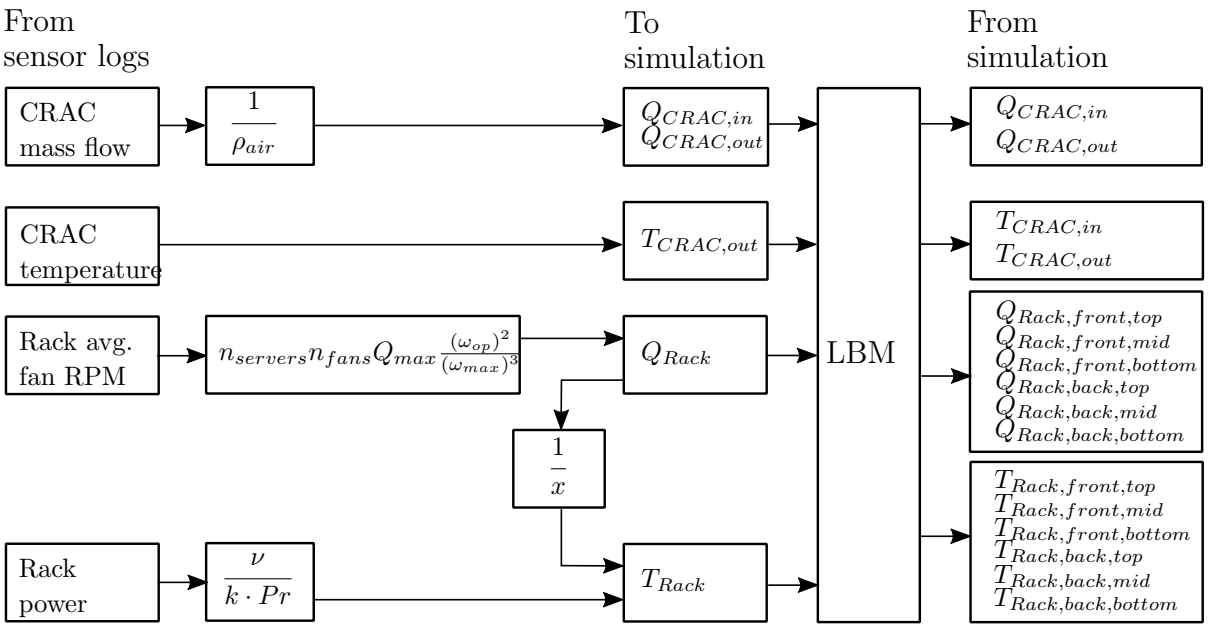


Figure 5.6: Overview of simulation inputs and outputs.

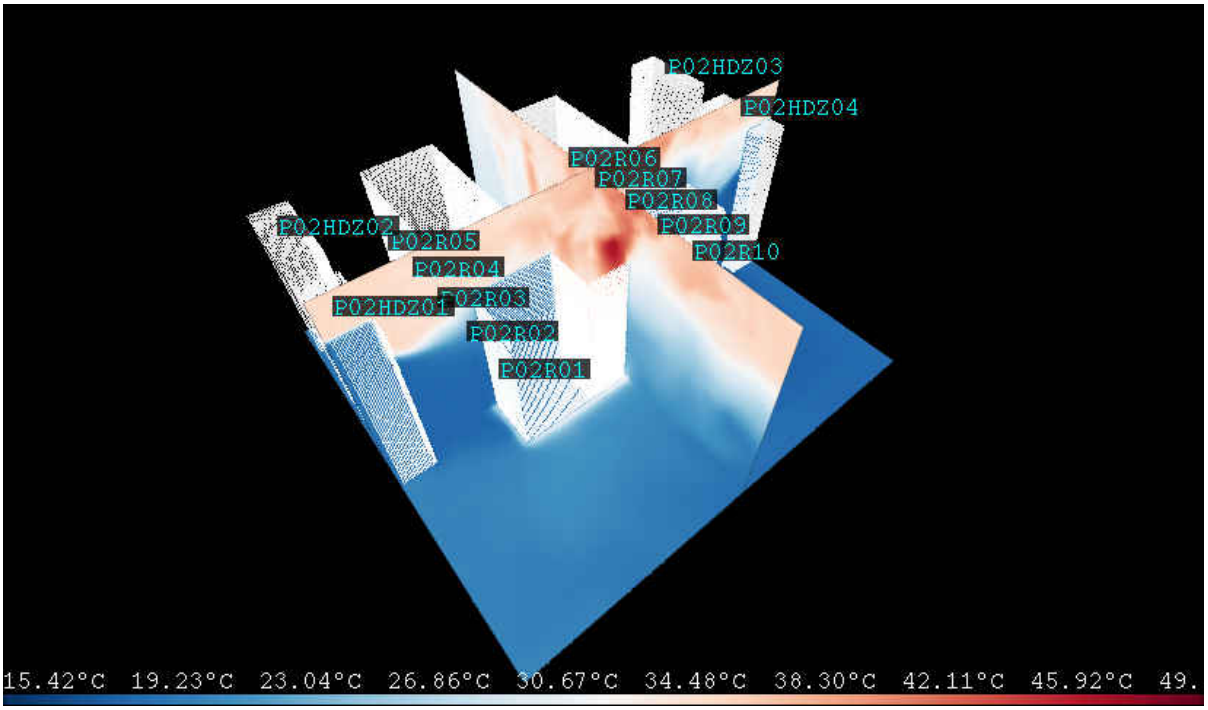


Figure 5.7: Simulating the thermal model of data center module POD 2.

Chapter 6

Model Validation

In order to know how accurate the CFD model of POD 2 described in chapter 5 was, it had to be compared against experimental values using data recorded during the physical experiment.

The temperature measurements on the front and back sides of the server racks were performed using the Microchip temperature sensor MCP9808 which according to specifications had a ± 0.5 °C maximum accuracy. Unfortunately, no experimental measurements of air velocity or flow rate were available for the server racks. While the CRACs had integrated sensors for temperature and mass flow rate (which can be accurately converted to volumetric flow rate using air density at 1 atm pressure), the model and accuracy of these sensors were unknown.

During initial testing of the model a lattice resolution of approximately 27 lu per meter, or 3.75 cm per lu, was used based on previous simulation domains created by the original author [4, pg.163]. When discretizing the model this resulted in approximately $1 \cdot 10^6$ lattice sites. It was however discovered that the accuracy of this resolution when setting and measuring volumetric flow rates was unacceptably poor, giving a maximum error of around 0.5 m^3/s when setting a flow rate of 2.0 m^3/s for the boundary conditions on the CRACs.

Increasing the resolution to 36 lu per meter, or 2.7 cm per lu, resulted in more accurate flow rates of less than half of the previous error. The discretized model the had approximately $7 \cdot 10^6$ lattice sites, which naturally slowed down the rate at which the model could be simulated relative to real time. Likely, a finer resolution would have given even better results but because of time constraints this was not investigated.

Because of the large number of measurement points the complete record of the thermal simulation along with the experimental values is available in appendix B.

CRACs

Table 6.1 shows the Root Mean Square (RMS) of the differences between a series of simulated temperatures and flow rates, $\vec{T}_{sim}$ and $\vec{Q}_{sim}$, and experimental, $\vec{T}_{exp}$ and $\vec{Q}_{exp}$, for a simulation running for $n = 2161$ minutes. Each sample was averaged over one minute, resulting in n values, so the RMS could be calculated as

$$T_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(T_{sim,i} - T_{\,exp,i})^2} \tag{6.1}$$

and similarly for the Q value series.

Table 6.1: The RMS of the difference between simulated and experimental temperatures in °C, and volumetric flow rate in m^3/s at the inlets of the four CRAC units.

CRAC	**Inlet temp.**	**Exhaust temp.**	**Inlet flow**	**Exhaust flow**
1	0.709	0.479	0.194	0.0705
2	0.836	0.331	0.132	0.0543
3	1.91	0.0759	0.0233	0.0322
4	0.302	0.0909	0.0298	0.0338

As can be seen in table 6.1 the temperature difference at the inlets was on average just over 1 °C off from experiments. Of particular note however is the simulated temperature of CRAC number 3, which according to the non-RMS average shown in plot B.1 was 1.91 °C higher than the experimental value. The reason for this is not clear, but could have to do with the fact that CRAC 1 and 2 had slightly lower volumetric flow rate at the inlets than expected (plots B.3 and B.4), likely because of discretization errors in the intake boundary conditions. Since they were lower than expected, they would contribute less to the cooling of the air, leaving temperatures at other CRACs slightly higher. This is not apparent in the result for the intake at CRAC 4 however, so the reason remains unknown.

The reason that the error is not equal on all CRAC inlets could likely be explained by inaccuracies in the volumetric air flow set at the CRAC boundary conditions, which can also be seen in the table. As mentioned earlier, a higher resolution lattice would probably help mitigate this. Other sources of error could relate to the air turbulence caused by air flows of some of the server racks.

In any case, the results show that the LBM model tested was accurate within approximately ±2 °C when the room temperature was measured as the average of the area

adjacent to the CRAC air inlets. Figures B.3 to B.6 in appendix B shows the complete record of the CRAC temperatures over time.

Server Racks

For the temperatures at the inlets (fronts) and exhausts (backs) of the server racks shown in table 6.2, the RMS of the difference between the simulation and experiment was calculated in the same way as those of the CRACs. The table shows that on average, the temperatures measured at the lowest point on the server racks was accurate within ±1 °C, the middle within ±2 °C and at the top ±4 °C. There are likely multiple sources for this error.

The simulation modeled the fronts of every rack as one uniform boundary condition which was geometrically identical to all others of the same type. In reality, the structure of the racks was more complex. Each rack had a perforated door in front of the servers on which the temperature sensor strips were placed, behind which the vertical positioning of the servers was not uniformly spaced. Some racks were only half full, resulting in a non-uniform air flow through the surface of the rack fronts. This was not taken into account by the model.

Table 6.2: The RMS of the difference between simulated and experimental temperatures in °C at different positions on the server racks.

Rack	Front Bottom	Front Mid	Front Top	Back Bottom	Back Mid	Back Top
1	0.316	3.06	3.19	2.25	2.55	7.4
2	0.812	1.78	4.36	3.51	2.83	3.42
3	1.09	1.47	4.13	6.04	3.26	2.69
4	0.599	1.97	5.29	2.21	1.4	1.04
5	0.761	2.2	2.95	0.85	2.43	1.85
6	1.08	1.24	1.6	1.42	1.04	5.06
7	0.221	1.32	1.17	2.53	1.94	5.66
8	1.14	1.2	4.47	1.45	2.45	0.719
9	0.64	1.66	4.75	0.969	1.55	0.728
10	0.653	3.11	4.48	2.45	3.84	1.87

Another source of error could be in the vertical positioning of the simulation measuring points related to the positioning of the experimental temperature sensor strips. Inaccuracies could also come from the modeling of viscosity (see section 2.3.5) and thermal

expansion coefficient of air (see section 2.3.4). These variables affect how the heat of air spreads by turbulence and gravity respectively.

Figures B.7 to B.16 in appendix B shows the complete record of the rack temperatures over time. Of particular note are the temperatures of racks P02R01, P02R02 and P02R03, where the fan speed and flow specifications were missing. They are therefore not a good indicator of the accuracy of the thermal model.

As for the temperatures at the backs of the servers, the comparison shows the temperatures being accurate within approximately ± 6 °C. Here the same sources of errors apply as for the fronts of the servers, but since the air temperature and velocity is higher and than at the front, the turbulence is more extreme and air currents more erratic. As mentioned in chapter 5, the average RPM of all servers in a rack was used to calculate the air flow through the rack. A more detailed model could set the air flow individually for each server. As also mentioned in this chapter, the air flow through some of the racks was completely unknown because of missing data for fan speeds.

Overall, a more accurate model of the positioning and air flow calculation of the servers in the racks, as well as a higher resolution lattice would likely improve these results.

Chapter 7

Conclusions

7.1 Results

In this thesis, the Computational Fluid Dynamics (CFD) application RAFSINE which implemented the Lattice Boltzmann method (LBM) was deployed on a remote GPU enabled server at RISE SICS North. The VirtualGL toolkit was configured on the server to access the OpenGL based visualization and user interface over a local network through the remote access system VNC.

VirtualGL introduced a slight performance penalty from its in-process GLX forking, which was improved by modifying the application to use multi-threading. This decoupled the OpenGL based visualization from the execution of the CFD simulation CUDA kernel and allowed the kernel to utilize a larger amount of the available computational resources.

The Lua-to-CUDA code generation API built into RAFSINE was used to create a simulation model of the experimental data center POD 2 at RISE SICS North. In order to simulate transient behavior of server power usage, the CUDA simulation kernel was modified to allow real-time updates of boundary conditions while the simulation was running. Code was also added to calculate lattice array indices of positions adjacent to the boundary conditions, to sample temperatures and volumetric flow rates during simulations.

In order to validate the model with experimental measurements, the simulation was set to update its boundary conditions representing server power usage and CRAC air conditioning at the same rate as the experiment. On a lattice resolution of 36 sites per meter, the model was found to be accurate within ± 1°C on a room level when measured as the average error of inlet return air to the CRACs, with a maximum error of ± 2°C.

When comparing the temperatures on a server rack level using the sensor strips on the front of the racks, an error of approximately ± 1°C, ± 2°C and ± 4°C was observed for the bottom, middle and top sensors respectively. The temperatures on the back of the racks were on average accurate within ± 6°C. The error on the back temperature sensors, and to an extent the front sensors was expected because of lack of air flow data for some of

the servers, as well as simplifications made when modeling the boundary conditions.

7.2 Conclusion

As mentioned in the introduction of this thesis, one of the main use cases for this type of simulation is for testing different physical configurations and placements of heat generating servers and their air cooling units. While further refinement of the model is needed to get the error of the simulated temperatures closer to the experimental error, different placements of equipment could be quite easily tested by adjusting the coordinates of their boundary conditions in the Lua code generation script (mentioned in chapter 2.4.2).

Another situation in which the simulation could be used is in testing air conditioning control systems. While the model created in this project set the boundary conditions for temperatures and air flows based on recorded data for the purposes of model validation, it would be possible to set them based on the output of an external control system program. This program could theoretically use the simulated temperature sensors as control inputs, as long as the required rate of measurements is lower than the rate of simulated constant time steps (see chapter 2.3.2).

The final use case this project set out to investigate was for producing data sets for use in sensor based automatic fault detection systems built on machine learning. Although no actual testing of such systems was done in this project, the program could easily be adjusted to produce machine learning data sets of temperatures and air flows in situations where some of the CRACs had malfunctioned. This situation would simply be a matter of setting the boundary conditions representing the CRACs to produce no air flow through their heat exchange inlets and exhausts. The record of the simulated temperatures could then be used as an input data set, from which the algorithm could learn to detect the signs of malfunctions from sensor values.

Finally, it should be said that while the LBM algorithm is not the only solution for CFD simulations of the thermal flow in data centers, its capacity to produce predictions in real-time or faster allows it to be used directly in automated monitoring and control systems. Its fast convergence could allow it to update its thermal model directly from sensor values in the real world data center, predict future conditions faster than real time, and set real world control signals accordingly. However, it should also be said that a complete CFD simulation of thermal conditions might not be necessary in the case of control systems. Simpler models which require lower amounts of computational resources might suffice.

7.3 Future Work

During initial experiments with simulating the data center model, a higher resolution grid was found to improve the accuracy of the predictive simulation. Because of time

constraints, the effect of lattice resolution on accuracy was not investigated fully, since a larger lattice significantly reduced the rate at which the simulation could be performed relative to real time. In his thesis, the RAFSINE author Nicolas Delbosc investigated executing the simulation on two different GPUs cooperating using the NVIDIA Peer-to-Peer framework and found that the simulation performed only 3.6% slower than twice as fast [4, pg.100]. While hardware limitations in the form of maximum PCI-Express bandwidth might limit the performance gain, further investigation in this area might allow for higher resolution lattices and larger domains when multiple GPUs can cooperate to execute the simulation.

Another improvement Delbosc suggested in his thesis [4, pg.195] is the development of a better graphical user interface for geometry construction and parameter modification at runtime. While the work done during this master thesis did implement the latter, a noticable performance decrease was added because of the way it was implemented. This code change for modifying boundary conditions during runtime could use further optimization.

The Lua-to-CUDA code generation API was used for geometry construction, but the integration of a 3D graphics toolkit such as OpenSceneGraph[1] could allow for the creation of a type of CAD interface like the ones seen in commercial CFD packages such as ANSYS[2] . The interface could also allow the user to set custom boundary conditions functions using code generation techniques similar to the existing ones.

Finally, in this thesis work the BGK simulation model was used for validation but, as Delbosc mentioned in his thesis, there exist other models such as Multiple Relaxation Time (MRT) and Cascaded LBM. These models can provide a higher degree of numerical stability for simulating turbulent fluid flow BGK [4, pg.196]. Implementing these models would require knowledge of fluid dynamics, LBM, the existing RAFSINE application and the programming techniques and languages used to create it.

[1] https://www.openscenegraph.org/
[2] https://www.ansys.com/

Appendix A

Deploying RAFSINE on Ubuntu 16.04 LTS

This appendix describes the steps for installing and running the RAFSINE application on remote GPU enabled servers running Ubuntu 16.04, with the aim of achieving remote visualization of the OpenGL user interface through VNC and/or X11-forwarding.

A.1 VirtualGL Installation and VNC Configuration

First, install the latest NVIDIA drivers. If not using the latest version of Ubuntu, add the *Personal Package Archives* for proprietary NVIDIA drivers[1]. At the time of writing, the latest version was 387:

```
# add-apt-repository ppa:graphics-drivers
# apt-get update
# apt-get install nvidia-387
```

Follow any configuration steps required by this installation. Reboot and check that the drivers were loaded:

[1] https://launchpad.net/~graphics-drivers/+archive/ubuntu/ppa

```
$ lsmod | grep nvidia
nvidia_uvm            688128  0
nvidia_drm             49152  2
nvidia_modeset        897024  2 nvidia_drm
nvidia              13971456  79 nvidia_modeset,nvidia_uvm
drm_kms_helper        155648  1 nvidia_drm
drm                   364544  5 drm_kms_helper,nvidia_drm
```

Optionally, the latest CUDA can now be installed by following the instructions on the NVIDIA downloads website[2]. Check which PCI bus ID is used by the GPU:

```
$ nvidia-xconfig --query-gpu-info
Number of GPUs: 1

GPU #0:
  Name      : GeForce GTX 1080 Ti
  UUID      : GPU-bfe3f861-3df7-0e3b-6c57-ea4addd8030c
  PCI BusID : PCI:130:0:0

  Number of Display Devices: 0
```

If using a GPU featuring a dedicated general-purpose compute mode, such as the NVIDIA Tesla series, enable graphics rendering by:

```
# nvidia-smi --gom=0
GOM changed to "All On" for GPU 0000:03:00.0.
All done.
Reboot required.
```

VirtualGL requires an X11 server to be running. To this end, install a desktop environment, display manager, and X11. Then enable the display manager and set the default runlevel to graphical:

[2] https://developer.nvidia.com/cuda-downloads

```
# apt-get install xfce4 lightdm xorg
# systemctl enable lightdm.service
# systemctl set-default graphical.target
```

Download and install the latest version of VirtualGL[3] as well as TurboVNC[4] and install them with:

```
# dpkg -i virtualgl_2.5.2_amd64.deb
# dpkg -i turbovnc_2.1.2_amd64.deb
```

If the display manager is running, stop it before configuring VirtualGL. Follow the instructions in the configuration script:

```
# systemctl stop lightdm.service
# vglserver_config
```

Next, configure X11 to not use the GPU as a display device, and supply the PCI bus ID from before:

```
# nvidia-xconfig --use-display-device=none --busid="PCI:130:0:0"
```

Reboot, and check that everything is working:

[3] https://www.virtualgl.org/
[4] https://www.turbovnc.org/

```
$ ps aux | grep Xorg | grep -v grep
root  2498  0.3  0.0 170224 48784 tty7  Ss+  08:23  1:25
    /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth
    /var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch
$ systemctl status lightdm.service
    lightdm.service - Light Display Manager
    Loaded: loaded (/lib/systemd/system/lightdm.service;
    enabled; vendor preset: enabled)
    Active: active (running) ...
```

Start the VNC server:

```
$ vncserver -geometry 1920x1200 -3dwm
```

Connect to the first screen of the VNC server using e.g. `109.225.89.135:1`. This should start the desktop environment (Xfce4 in this case). To check that VirtualGL is working, try running the OpenGL example application `glxgears`. Since the VNC server was started with the `-3dwm` option, OpenGL applications can be started normally by for example typing their name in a terminal. Without this option, the application name has to be prepended with `vglrun` to indicate 3D support is required. The environment variable `VGL_LOGO=1` displays the VirtualGL logo:

```
# apt-get install mesa-utils
$ VGL_LOGO=1 vglrun glxgears
```

The result should look like figure A.1. When executing the program through a debugger such as The GNU Debugger (GDB), VirtualGL can be loaded indirectly by setting an environment variable inside GDB:

```
set environment LD_PRELOAD=/usr/lib/libvglfaker.so
```

Since the VNC server is only protected by a password as default, it can be a potential security hazard. This can be mitigated by only allowing local connections to the VNC server, tunneled from a client by SSH. Start the remote VNC server with

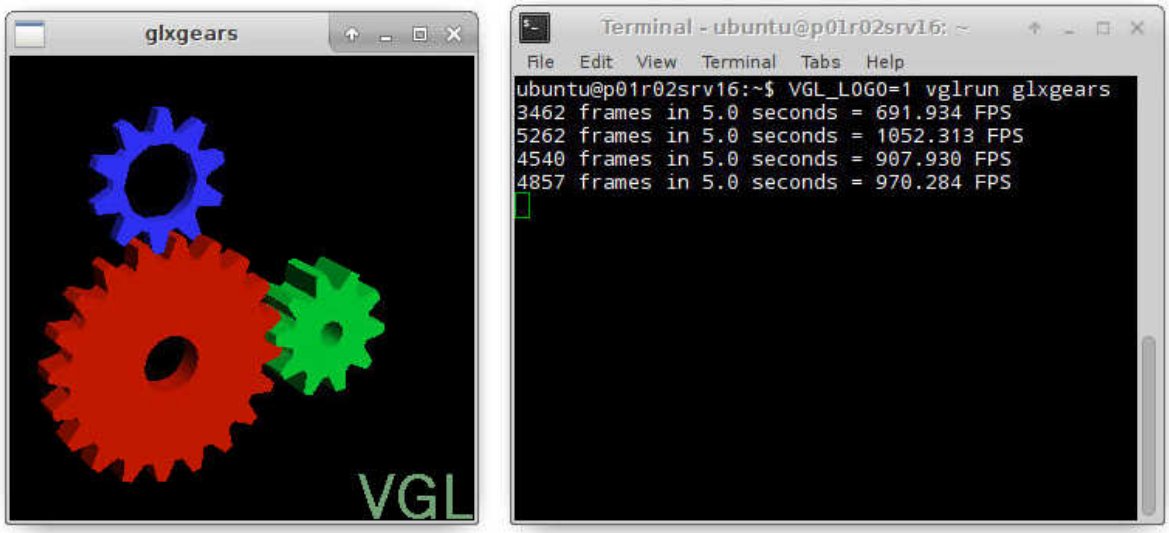


Figure A.1: The OpenGL example application `glxgears` using hardware accelerated 3D rendering through VirtualGL over VNC.

```
$ vncserver -geometry 1920x1200 -localhost -3dwm
```

and an SSH-tunnel on the client by:

```
$ ssh username@$REMOTE_IP -x -e none
        -L $LOCAL_VNC_PORT:127.0.0.1:$REMOTE_VNC_PORT
        -p $REMOTE_SSH_PORT
```

for example

```
$ ssh ubuntu@109.225.89.135 -x -e none
        -L 5901:127.0.0.1:5901 -p 31761
```

The `-x` switch disables X11-forwarding, while `-e none` disables escape characters, neither of which is useful when connecting through VNC. Now the client can connect to `localhost:1` in a VNC viewer. All VNC network traffic between client and server will be encrypted, and the only way to access the VNC server is by connecting and tunneling

through SSH. Using the TurboVNC client is advisable since it was developed by the VirtualGL team for this purpose.

A.2 VirtualGL through SSH X11-forwarding

VirtualGL can also be made to work through SSH X11-forwarding, as long as the client is running an X11-server. This solution has the advantage of seamless integration with the X11 window manager on the client, and slightly less CPU usage on the server side. Follow the steps in chapter A.1 to install and configure VirtualGL.

On Ubuntu, X11-forwarding is enabled on the SSH server by default. Check that normal X11-forwarding is working by connecting with the `-X` option

```
$ ssh username@$REMOTE_IP -X -e none -p $REMOTE_SSH_PORT
```

and running a non-OpenGL application such as `xclock`. If the forwarding works, exit the SSH connection, and reconnect to the server with

```
$ vglconnect -s username@REMOTE_IP -e none -p $REMOTE_SSH_PORT
```

This will automatically open a network stream called *VGL Transport* which is essentially a dedicated TCP socket for sending encoded or compressed 3D images between client and server. OpenGL applications can then be launched from the terminal by typing their name prepended with `vglrun`.

A.3 Installing the RAFSINE Dependencies

Install the latest NVIDIA GPU drivers as described in appendix A.1, then the latest CUDA toolkit by following the instructions on the NVIDIA downloads website[5].

The rest of the build dependencies can be installed by the package manager,

```
# apt-get install libboost-all-dev lua5.1 pkg-config \
        libglew-dev freeglut3-dev libjpeg9-dev \
        libpthread-stubs0-dev libc6-dev gcc-6-base luarocks \
        libfltk1.3-dev libglfw3-dev \
        cmake
```

and by `luarocks`

```
# luarocks install penlight
# luarocks install multikey
```

[5] https://developer.nvidia.com/cuda-downloads

Appendix B

Plots and Charts

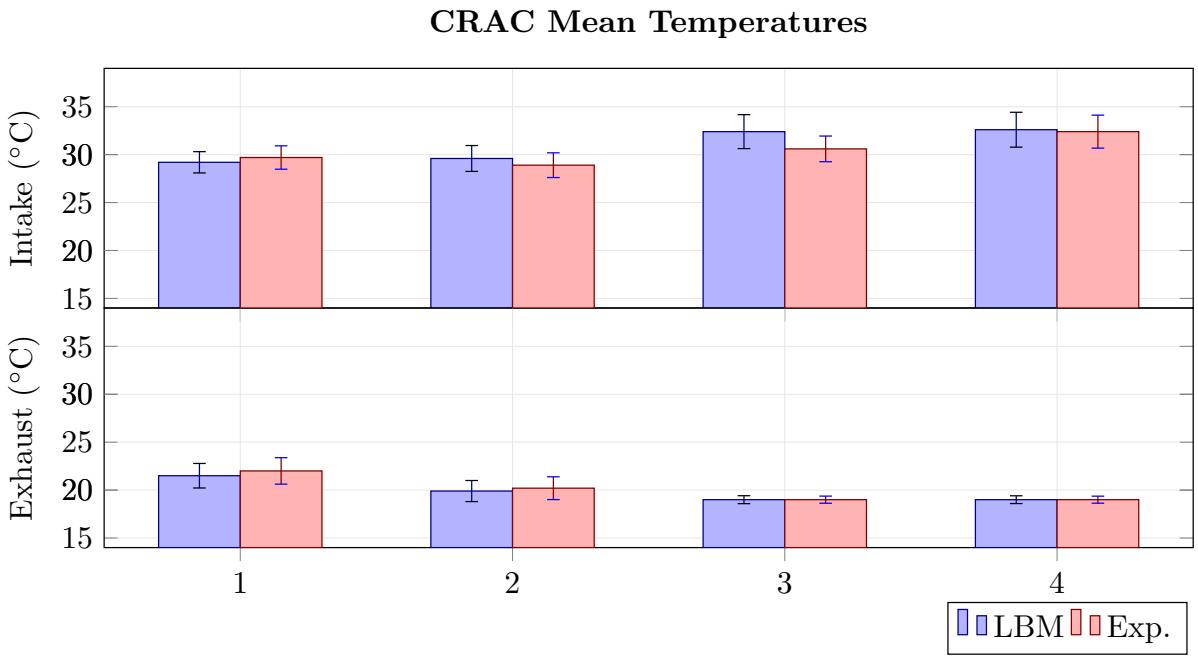


Figure B.1: Comparison of simulated and experimental mean temperatures over the different measurement points on the CRAC units. The error bars represent the standard deviation of each data set.

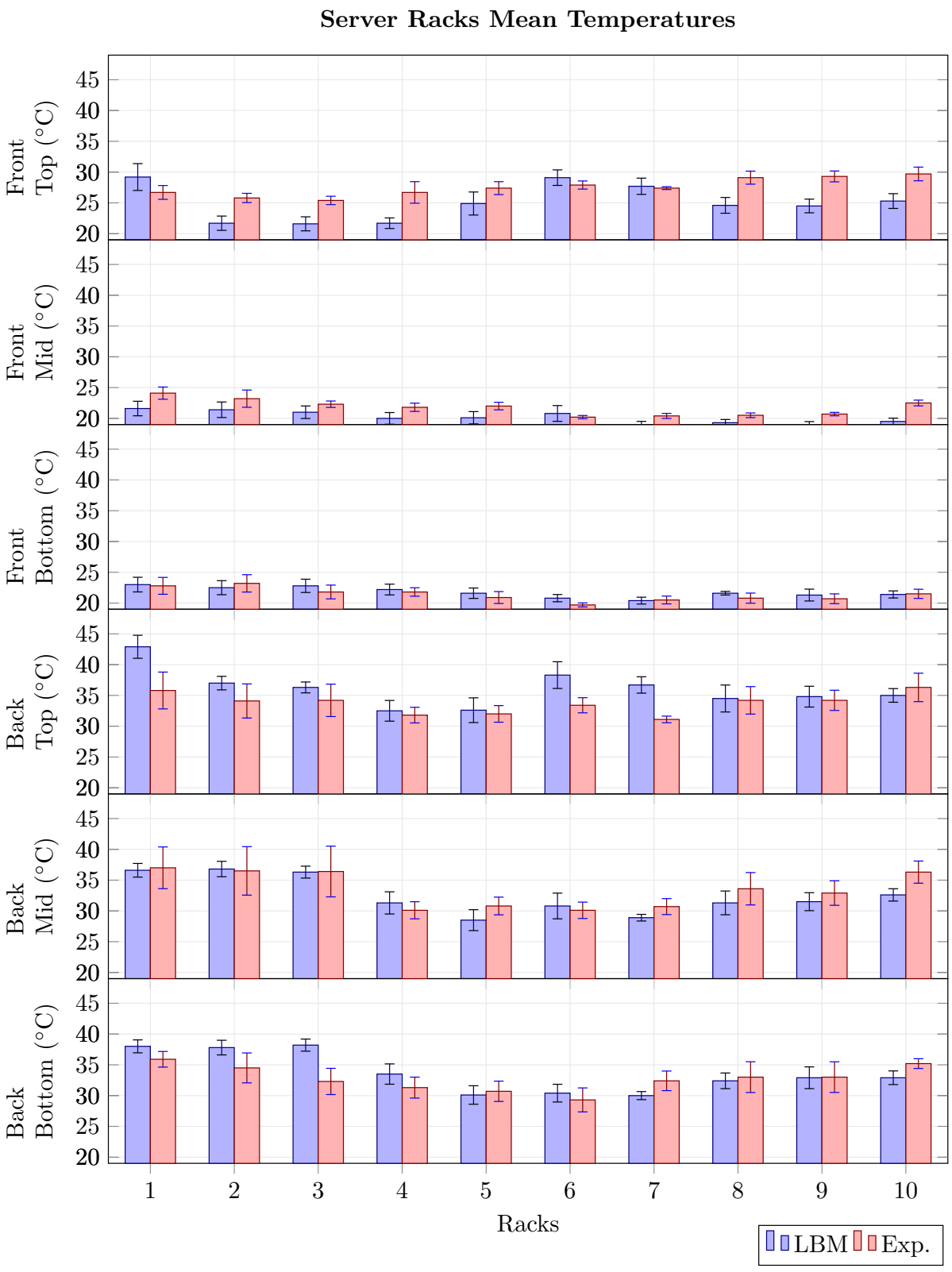


Figure B.2: Comparison of simulated and experimental mean temperatures over the different measurement points on the racks. The error bars represent the standard deviation of each data set.

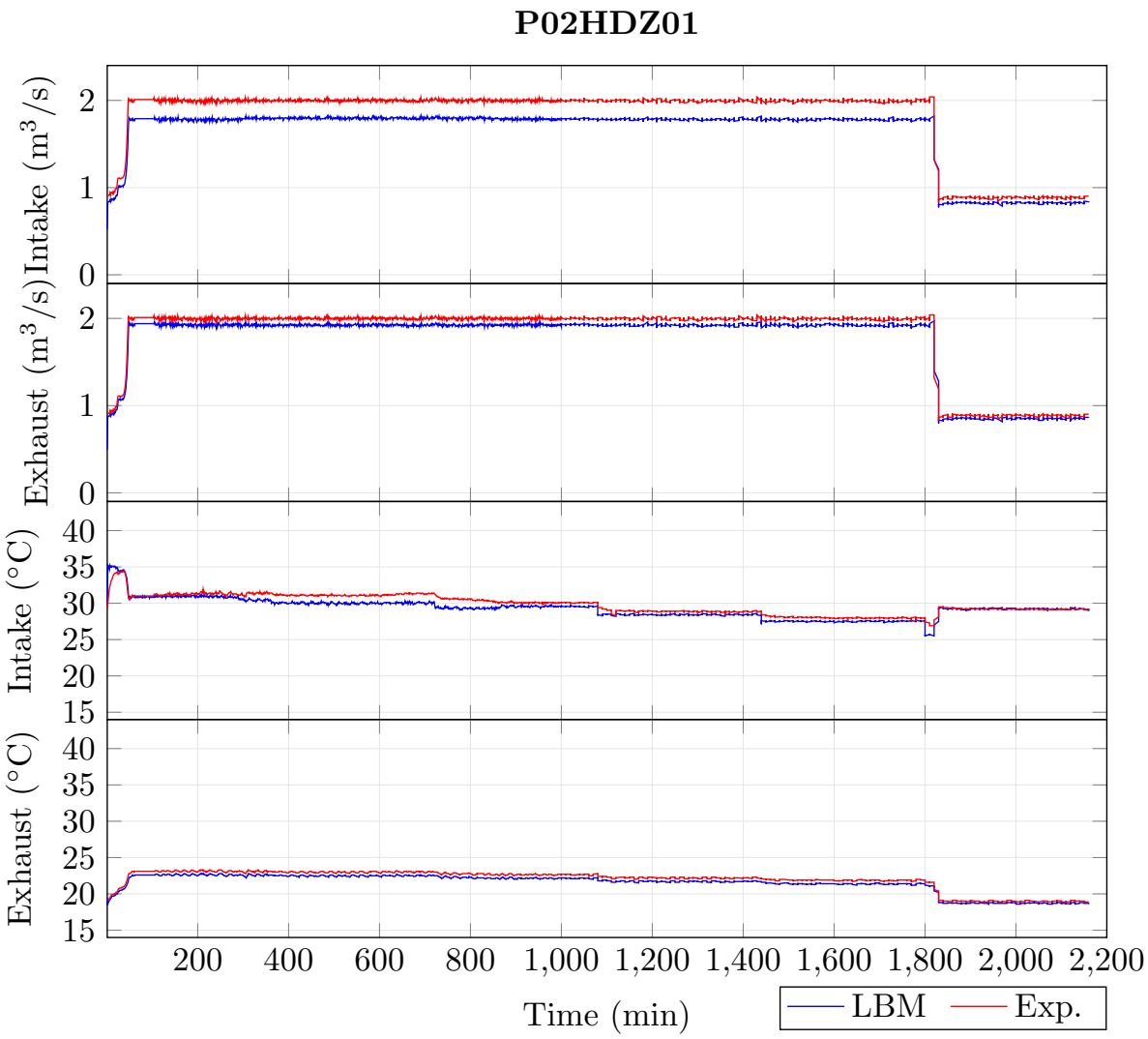


Figure B.3: Comparison of simulated and experimental temperatures for CRAC unit P02HDZ01.

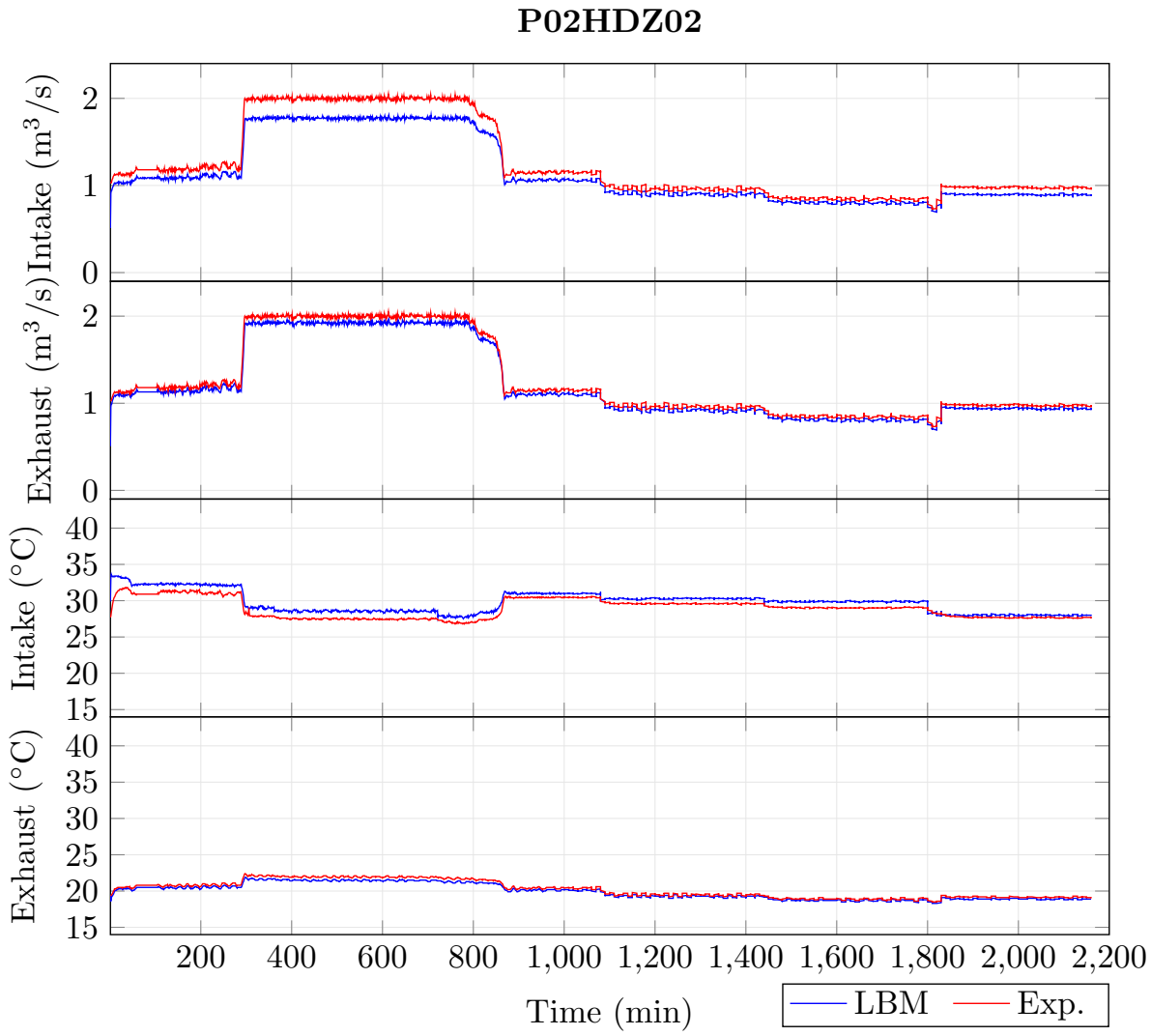


Figure B.4: Comparison of simulated and experimental temperatures for CRAC unit P02HDZ02.

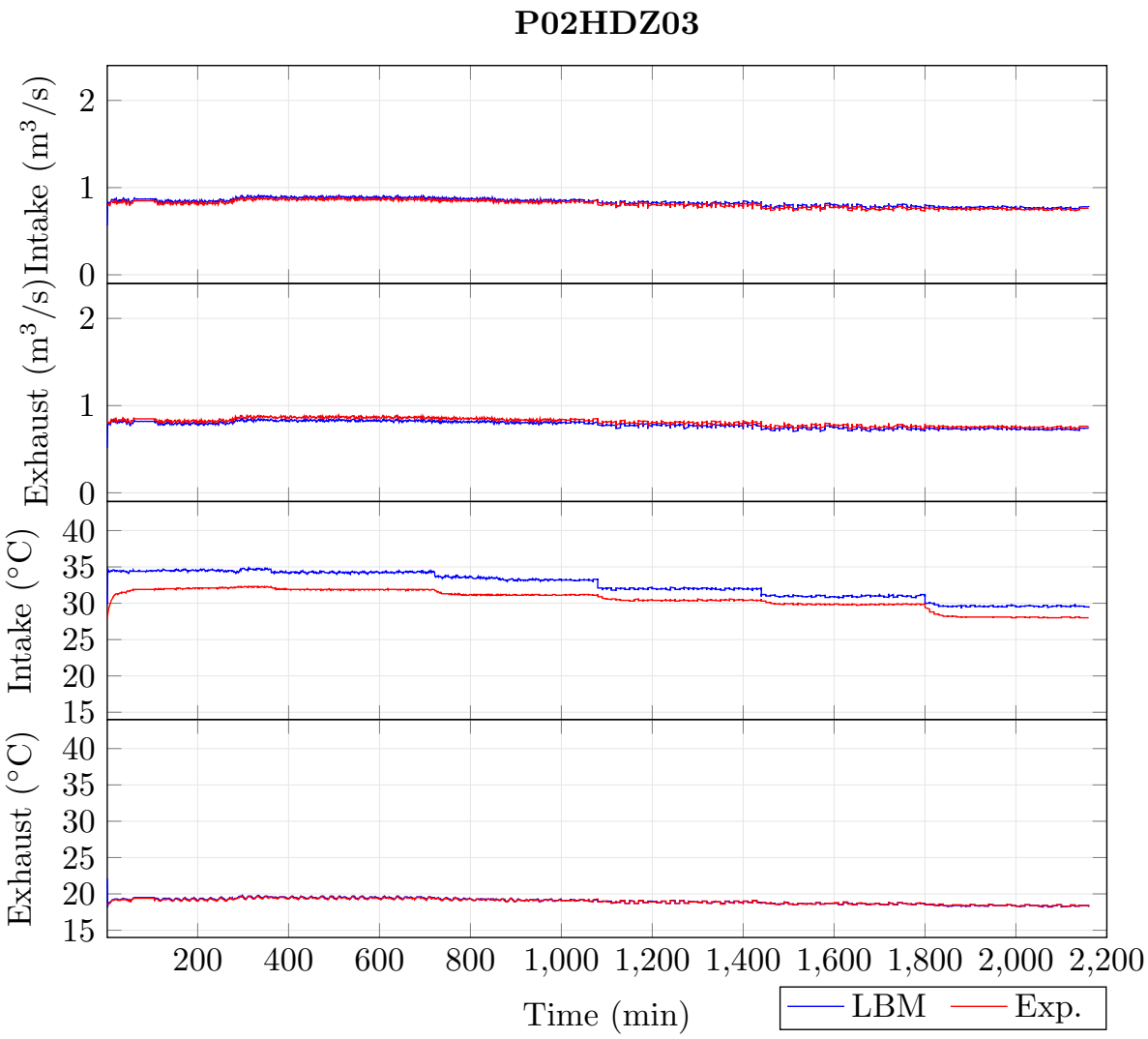


Figure B.5: Comparison of simulated and experimental temperatures for CRAC unit P02HDZ03.

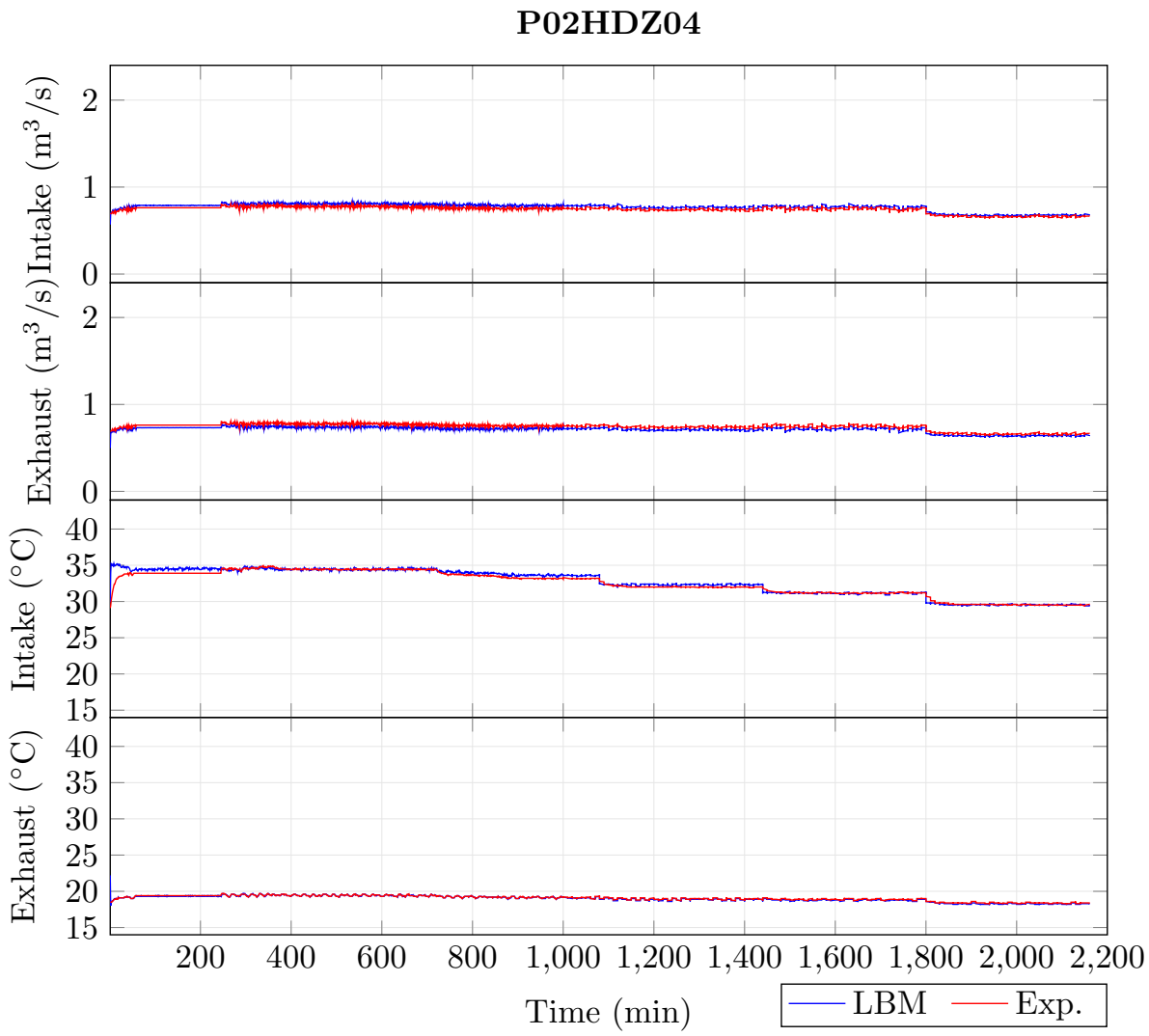


Figure B.6: Comparison of simulated and experimental temperatures for CRAC unit P02HDZ04.

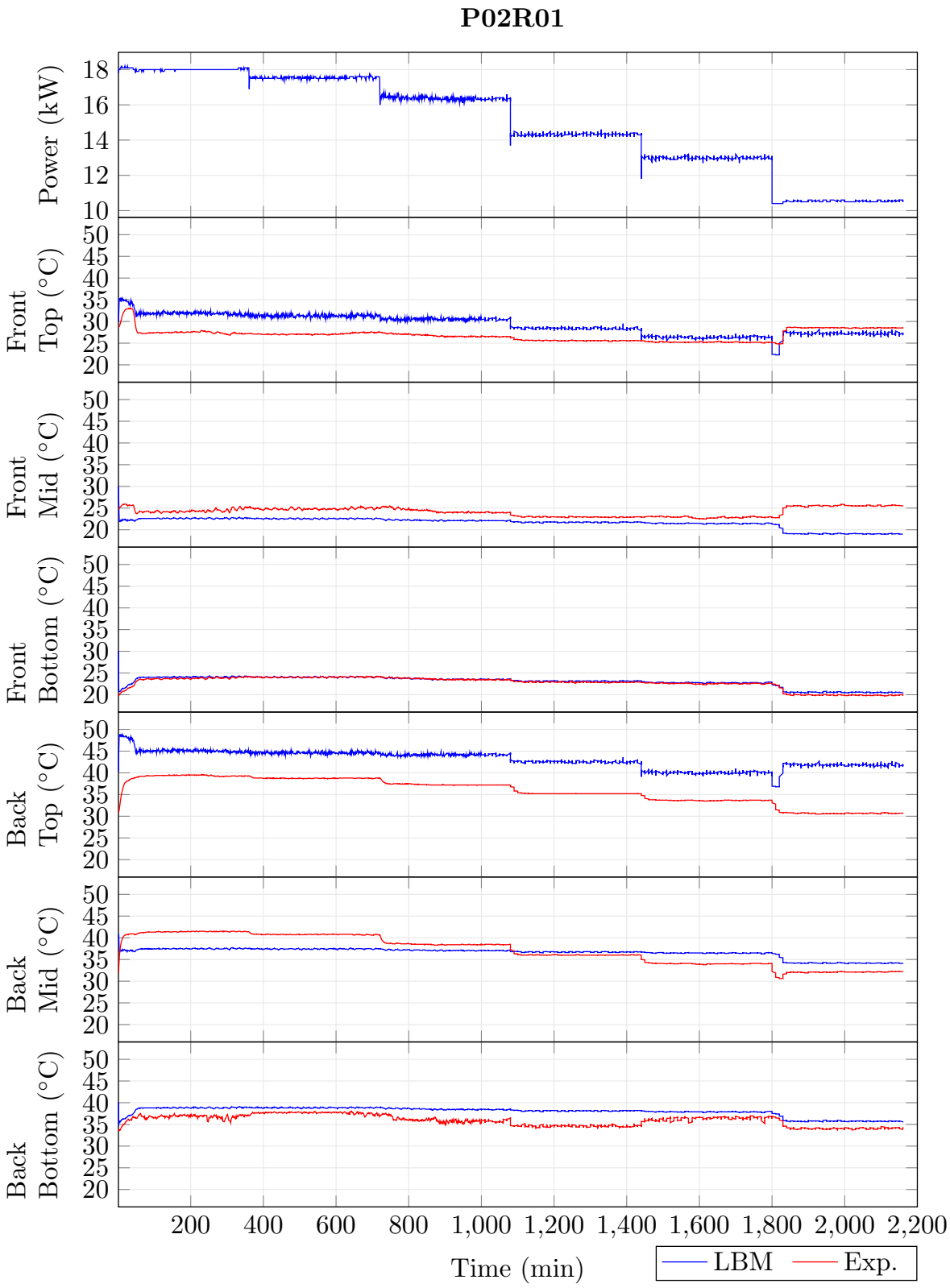


Figure B.7: Comparison of simulated and experimental temperatures for server P02R01. Note that the accuracy of this simulation is expected to be quite poor since fan speed data was missing for this rack.

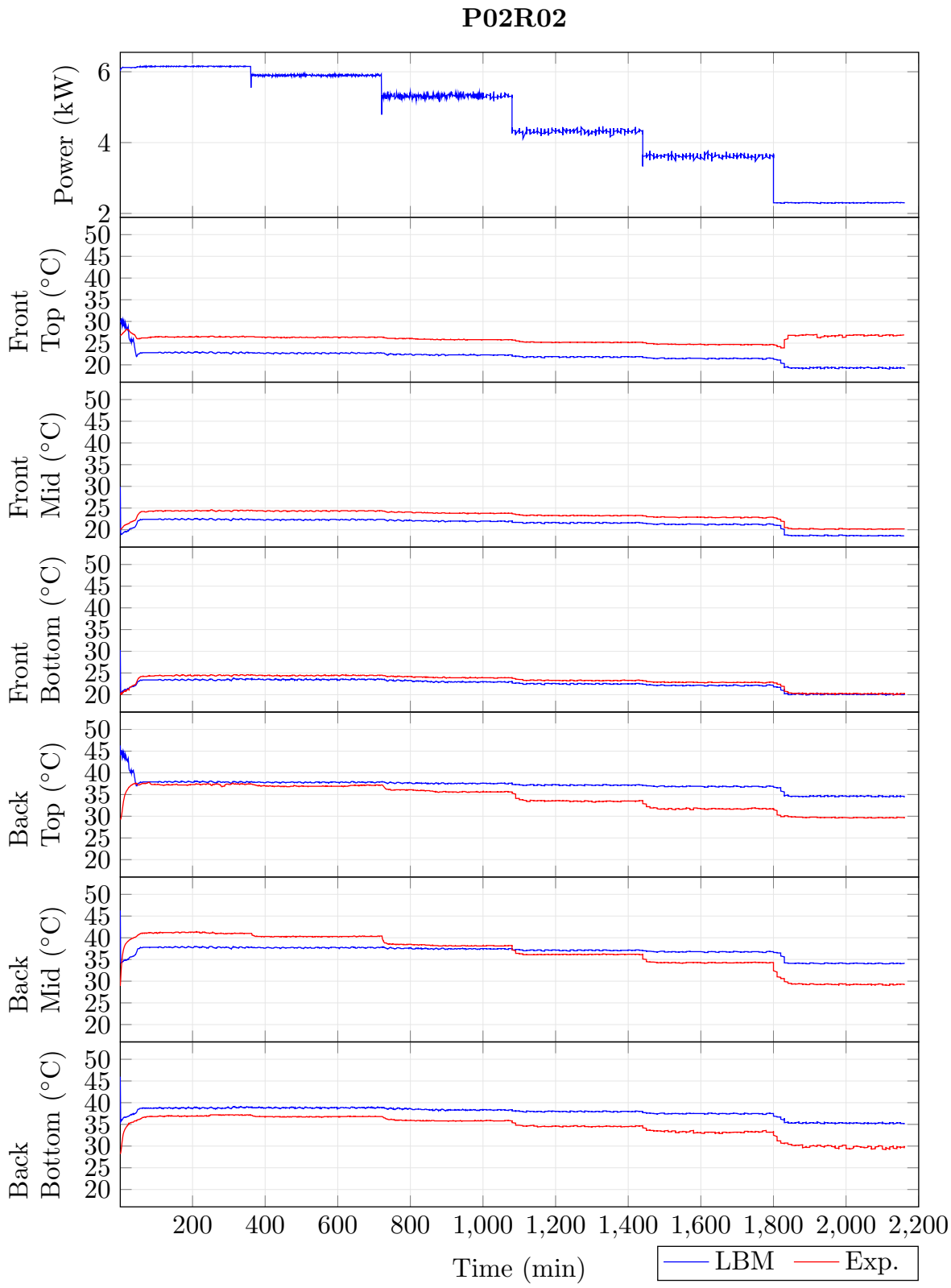


Figure B.8: Comparison of simulated and experimental temperatures for server P02R02. Note that the accuracy of this simulation is expected to be quite poor since fan speed data was missing for this rack.

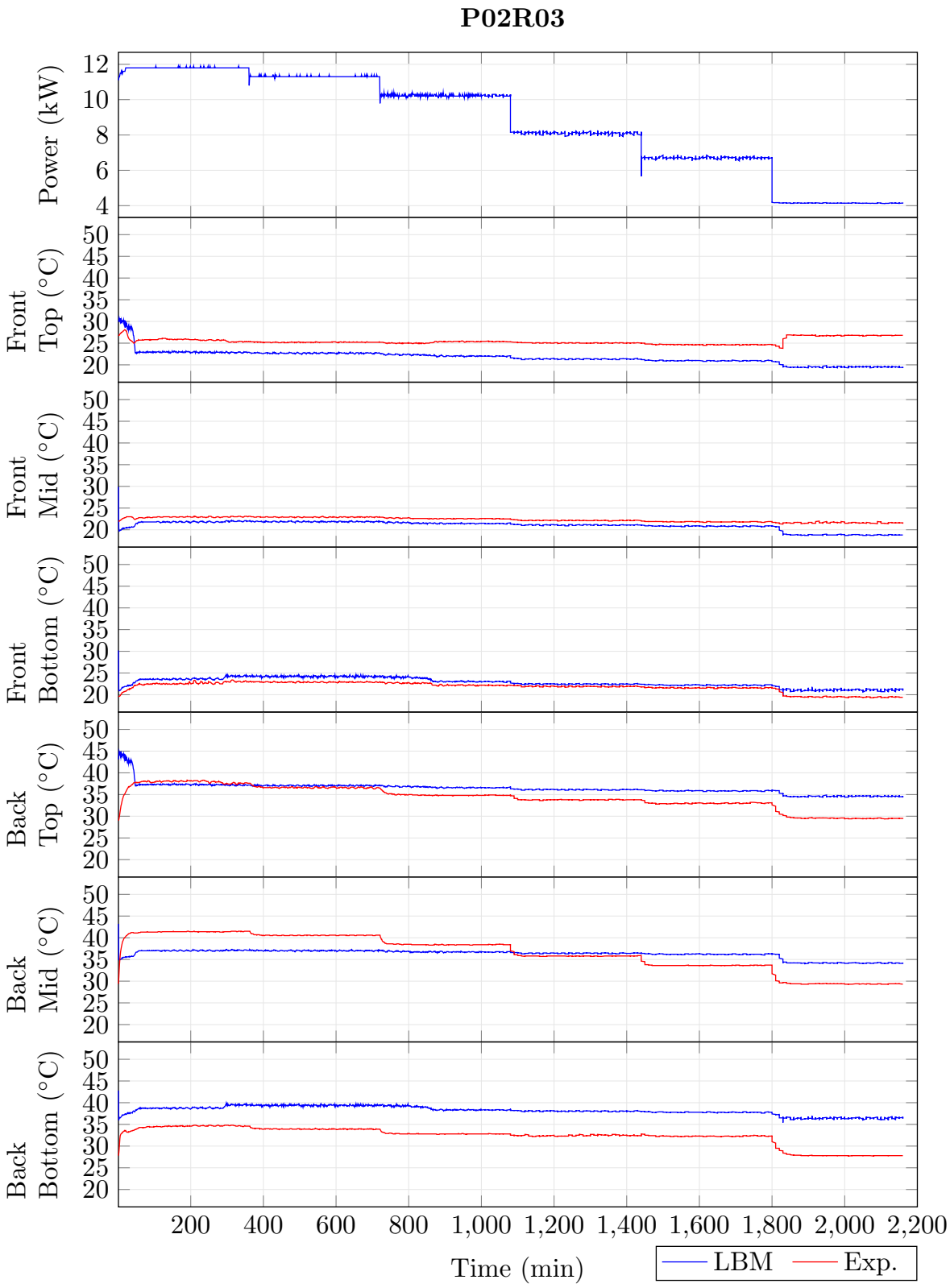


Figure B.9: Comparison of simulated and experimental temperatures for server P02R03. Note that the accuracy of this simulation is expected to be quite poor since fan speed data was missing for this rack.

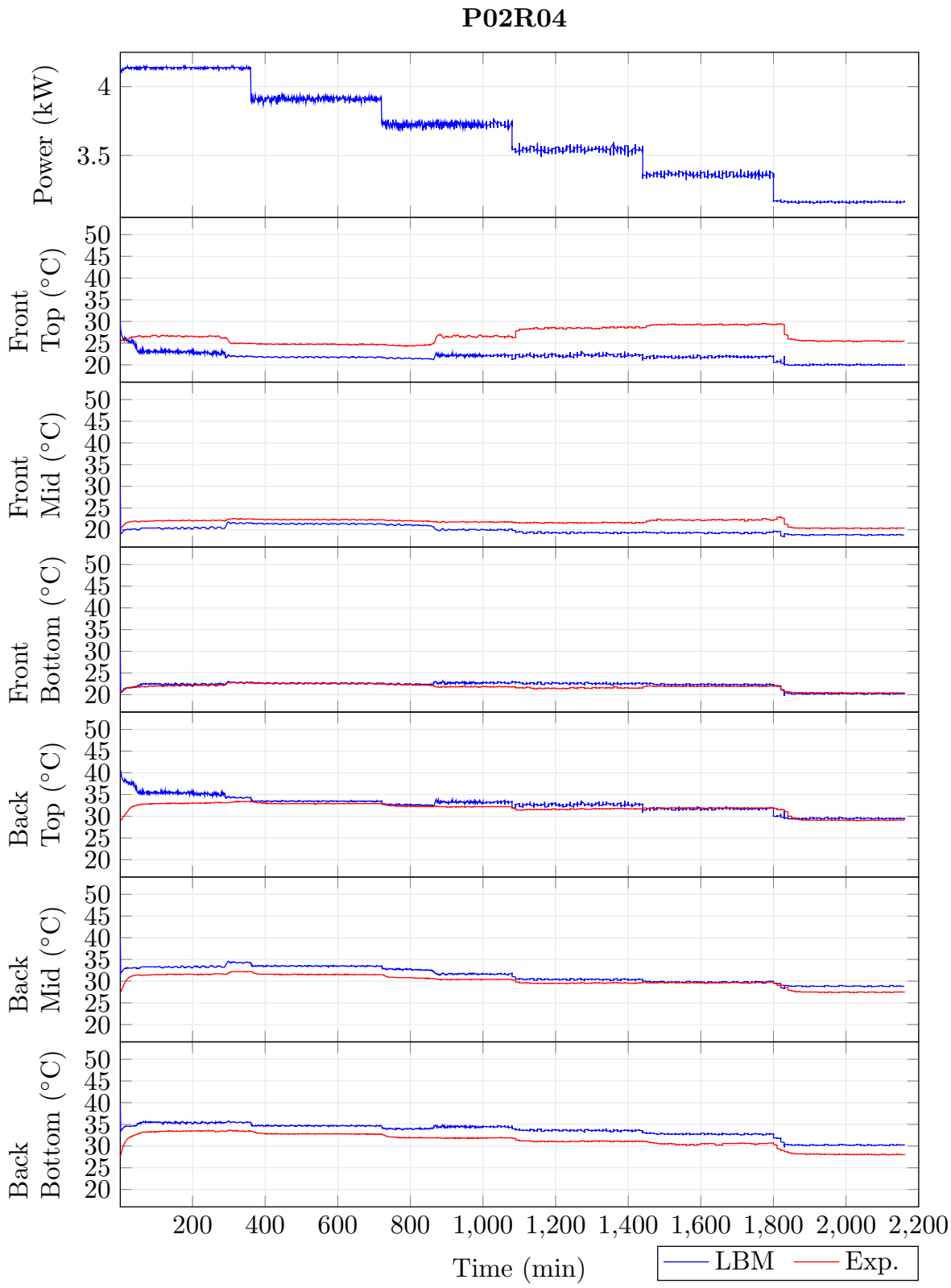


Figure B.10: Comparison of simulated and experimental temperatures for server P02R04.

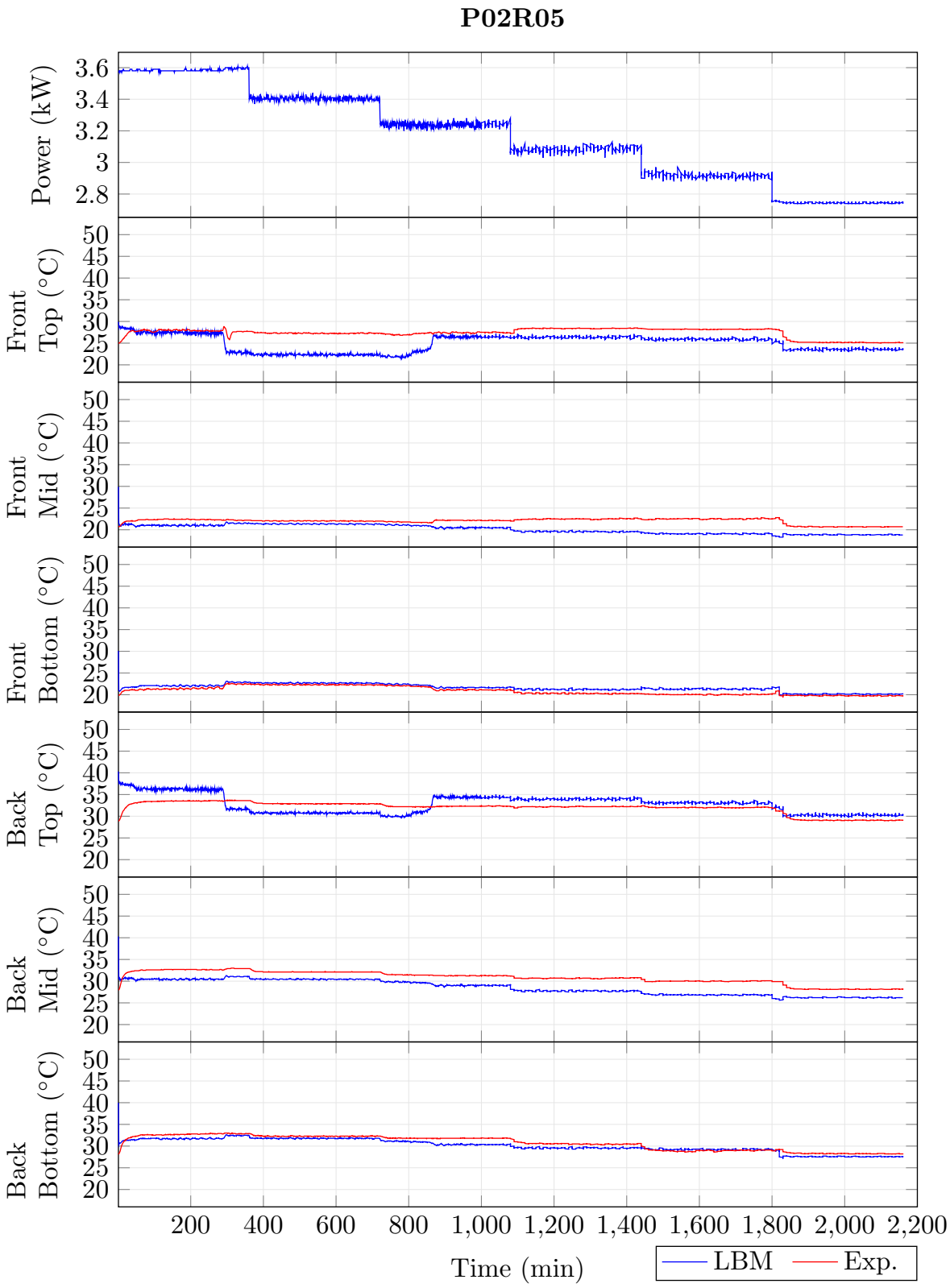


Figure B.11: Comparison of simulated and experimental temperatures for server P02R05.

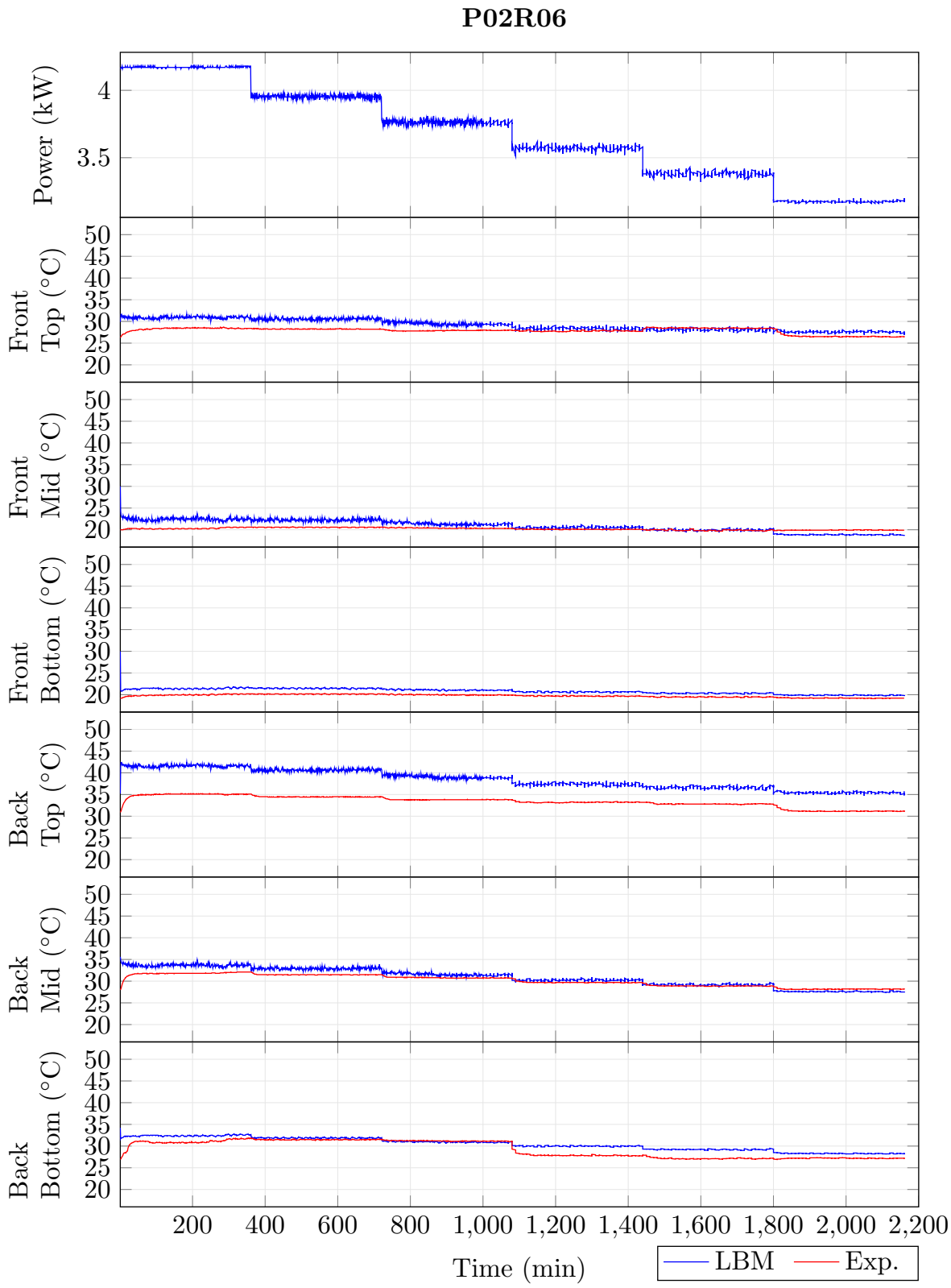


Figure B.12: Comparison of simulated and experimental temperatures for server P02R06.

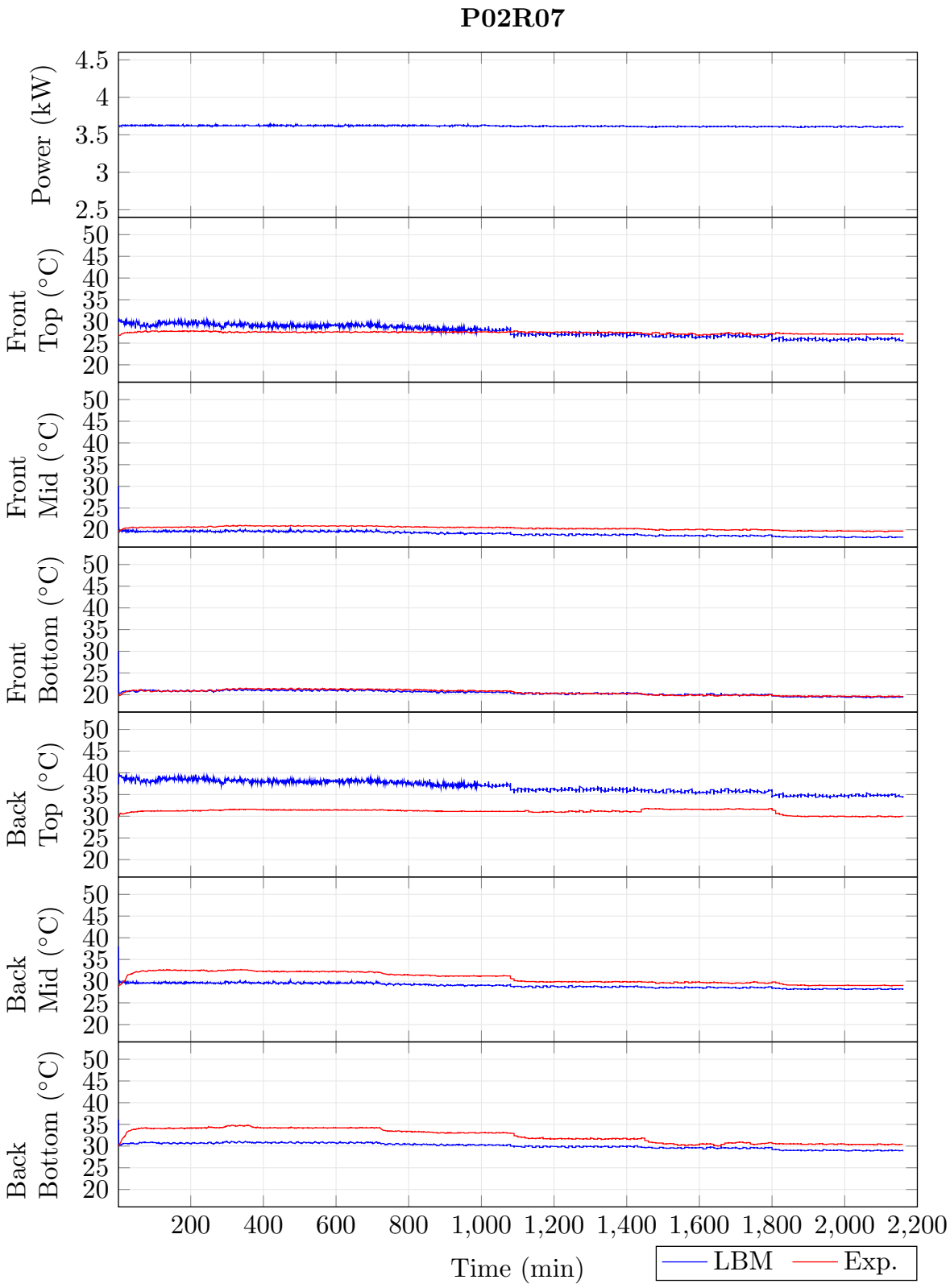


Figure B.13: Comparison of simulated and experimental temperatures for server P02R07.

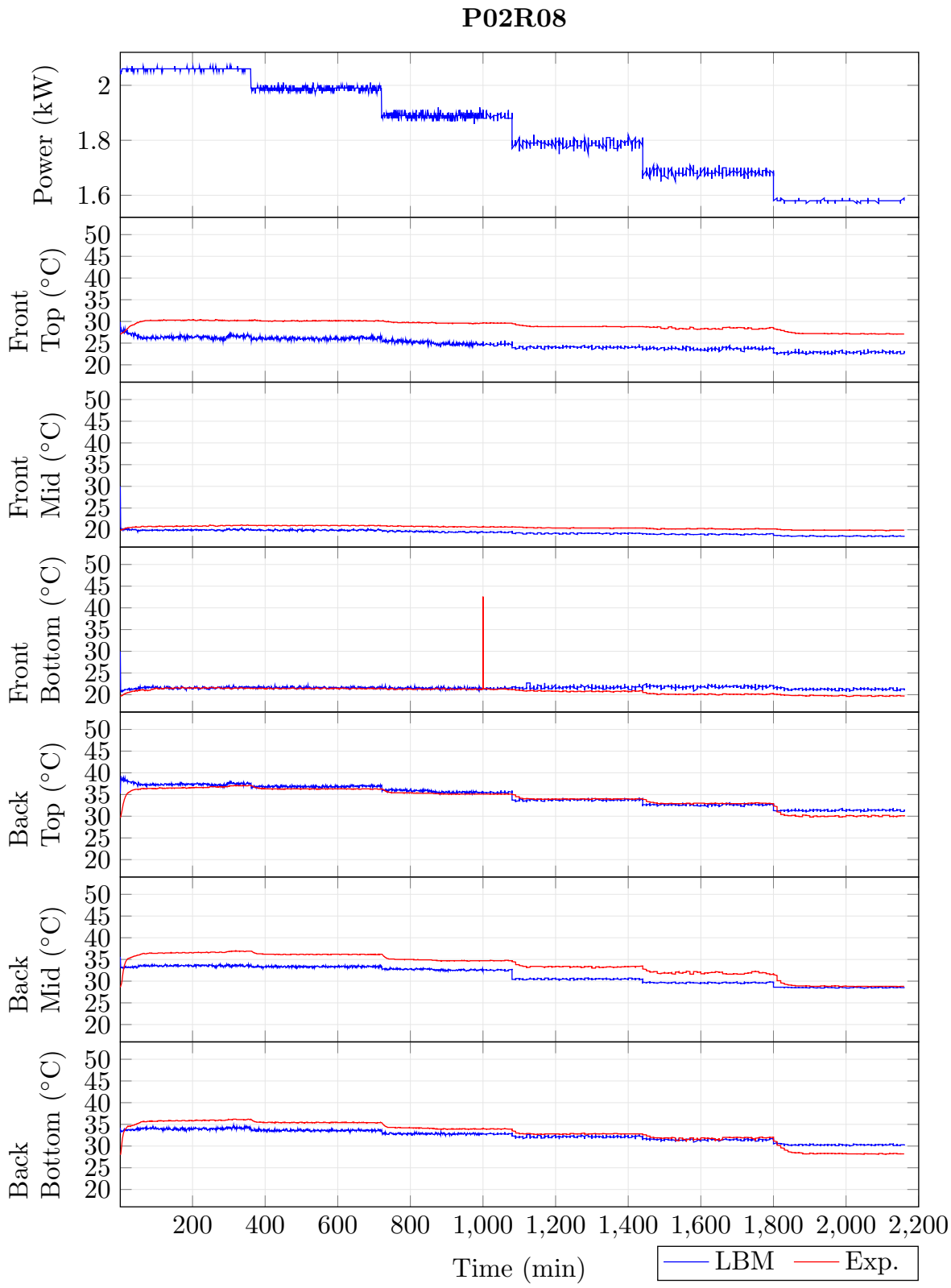


Figure B.14: Comparison of simulated and experimental temperatures for server P02R08.

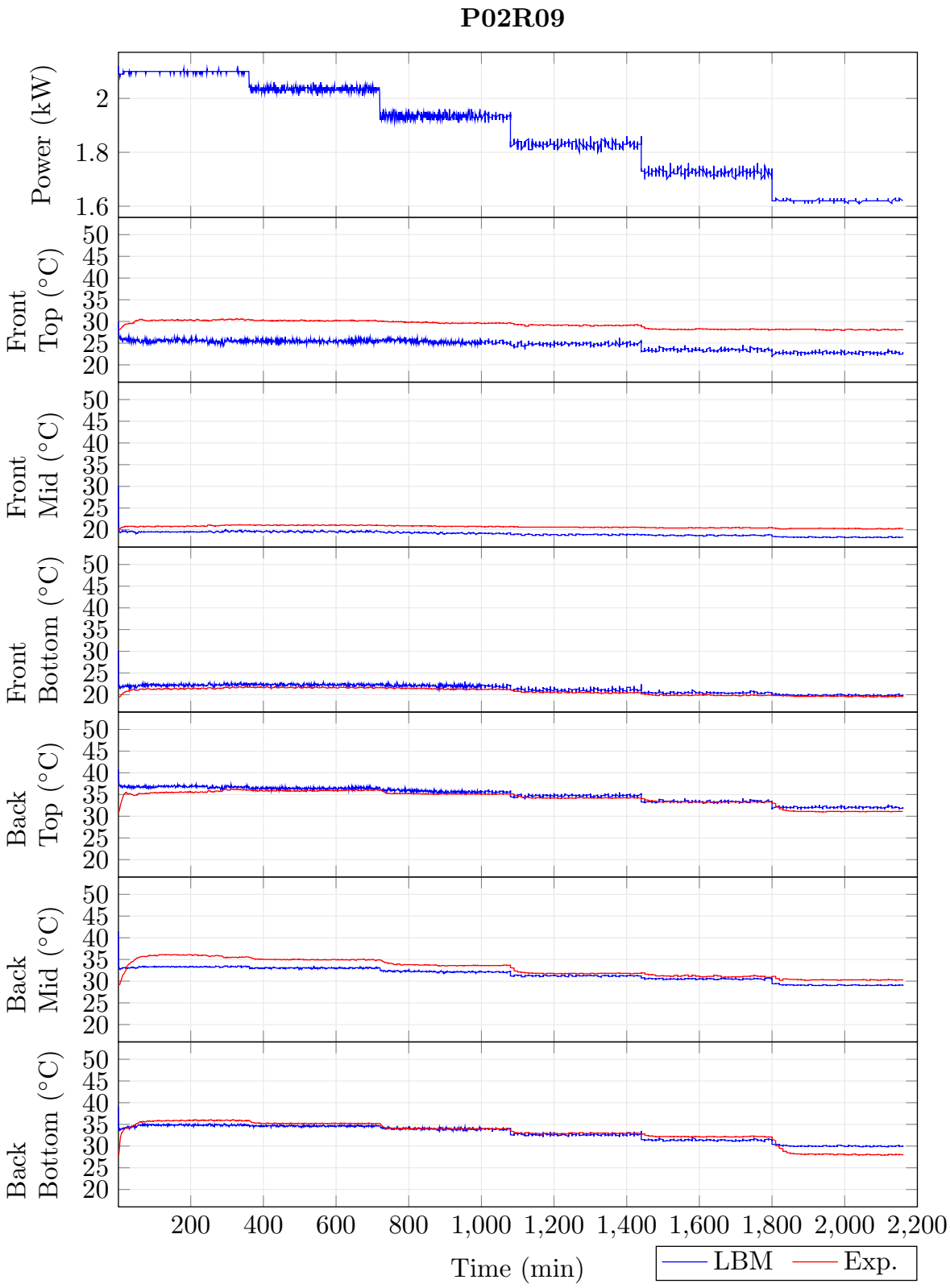


Figure B.15: Comparison of simulated and experimental temperatures for server P02R09.

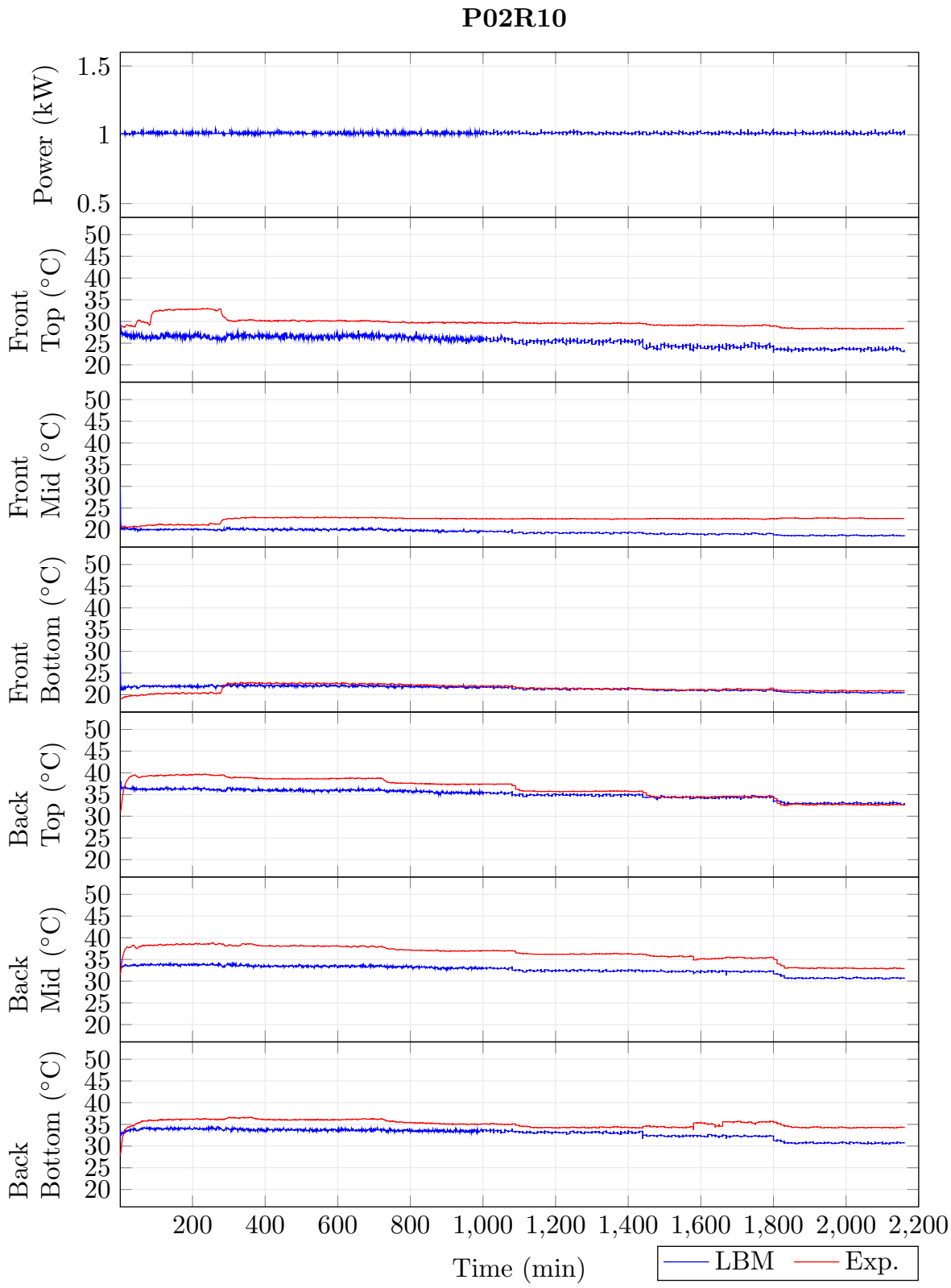


Figure B.16: Comparison of simulated and experimental temperatures for server P02R10.

Appendix C

Acronyms

1. **API** Application Programming Interface. 20, 37, 51, 53, 85, *Glossary:* API
2. **BGK** Bhatnagar–Gross–Krook. 12, 14, 16, 17, 53, *Glossary:* BGK
3. **BTE** Boltzmann transport equation. 1, *Glossary:* BTE
4. **C++** Programming language. 21, 22, 25, 33, 35, 84, 86, *Glossary:* C++
5. **CFD** Computational Fluid Dynamics. 1, 8–10, 15, 20, 23, 24, 26, 32, 47, 51–53, 85, *Glossary:* CFD
6. **CPU** Central Processing Unit. 21, 31–35, 60, *Glossary:* CPU
7. **CRAC** Computer Room Air Conditioner. 37, 39–42, 44, 45, 47–49, 51, 52, 63, 65–68, *Glossary:* CRAC
8. **CSV** Comma-separated values. 45, *Glossary:* CSV
9. **CUDA** NVIDIA CUDA. 20–22, 24–26, 32–37, 51, 53, 56, 61, *Glossary:* CUDA
10. **DRI** Direct Rendering Interface. 27, *Glossary:* DRI
11. **FDM** Finite Difference Method. 8, *Glossary:* FDM
12. **FEM** Finite Element Method. 1, 8, *Glossary:* FEM
13. **FLTK** Fast Light Toolkit. 32, *Glossary:* FLTK
14. **FVM** Finite Volume Method. 1, 8, 20, 84, *Glossary:* FVM
15. **GDB** The GNU Debugger. 58, *Glossary:* GDB

16. **GLEW** OpenGL Extension Wrangler Library. 25, *Glossary:* GLEW

17. **GLX** OpenGL extension to the X Window System. 26–29, 31, 51, 80, 85, *Glossary:* GLX

18. **GPU** Graphics Processing Unit. 1, 2, 20–22, 25–29, 31, 32, 34, 36, 51, 53, 55–57, 61, 84, 85, *Glossary:* GPU

19. **GUI** Graphical User Interface. 84, *Glossary:* GUI

20. **IBVP** Initial Boundary Value Problem. 8, 9, *Glossary:* IBVP

21. **JPEG** Joint Photographic Experts Group. 25, *Glossary:* JPEG

22. **LBM** Lattice Boltzmann method. 1, 2, 9–11, 15, 16, 20, 21, 24, 39, 41, 48, 51–53, 83, *Glossary:* LBM

23. **LES** Large Eddy Simulation. 15, 16, *Glossary:* LES

24. **LibGL** Library which implements the GLX interface. 27, *Glossary:* LibGL

25. **Lua** Scripting language. 22, 25, 26, 35, 37, 39, 51–53, *Glossary:* Lua

26. **mutex** Mutual Exclusion. 32, 33, *Glossary:* mutex

27. **OpenGL** Open Graphics Library. 2, 22, 23, 25–28, 31, 32, 34, 35, 51, 55, 58–60, 84–86, *Glossary:* OpenGL

28. **PDE** Parital Differential Equation. 1, 8, 84, *Glossary:* PDE

29. **pthreads** POSIX Threads. 32, *Glossary:* pthreads

30. **RANS** Reynolds Averaged Navier-Stokes. 7, 15, *Glossary:* RANS

31. **RISE SICS North** Research Institutes of Sweden, Swedish Institute of Computer Science, North. 2, 24, 25, 29, 37, 38, 51, *Glossary:* RISE SICS North

32. **RMS** Root Mean Square. 48, 49, *Glossary:* RMS

33. **SM** Streaming Multiprocessor. 21, *Glossary:* SM

34. **SSH** Secure Shell. 2, 26–28, 58–60, *Glossary:* SSH

35. **UNIX** A family of operating systems. 2, 21, 26, 28, 86, *Glossary:* UNIX

Appendix C

Glossary

40. API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. 20

41. BGK Bhatnagar–Gross–Krook (BGK) is a simple model for calculating the collision step in LBM. 12

42. BTE The Boltzmann transport equation (BTE) was developed by Ludwig Boltzmann to describe the statistical behavior of a thermodynamic system not in a state of equilibrium. 1

43. C++ The (C++) programming language is a general purpose multi-paradigm programming language with imperative and object oriented features. 21

44. CFD Computational Fluid Dynamics (CFD) is a branch of fluid mechanics which uses numerical analysis and data structures to solve the dynamics of fluid behaviours. 1, 51

45. CPU A Central Processing Unit (CPU) is an electronic hardware circuit inside a computer which can perform arithmetic, logic, input/output and flow control operations from computer program instructions. 21

46. CRAC A Computer Room Air Conditioner (CRAC) is an air cooling device containing heat exchange cooling coils and fans. 37

47. CSV Comma-separated values (CSV) is text table file where the values are separated by commas. 45

48. CUDA NVIDIA CUDA (CUDA) is a parallel computing platform developed by NVIDIA. Its main usage is to perform general purpose data processing on GPUs. 20

49. DRI Direct Rendering Interface (DRI) is a framework which allows an OpenGL based application to directly access the graphics hardware (the GPU). 27

50. FDM The Finite Difference Method (FDM) is a numerical method for solving PDEs by approximating them with difference equations. 8

51. FEM The Finite Element Method (FEM) is a method for solving PDEs similar to FVM. 1, 8

52. FLTK Fast Light Toolkit (FLTK) is a graphical widget library for Graphical User Interface (GUI)s supporting 3D OpenGL rendering. 32

53. FVM The Finite Volume Method (FVM) is a numerical method for solving PDEs using systems of algebraic equations. The problem domain is discretized into smaller volumes yielding a system where the value for each point can be approximated. 1, 8

54. GDB The GNU Debugger (GDB) is an open-source debugger for many programming languages such as C++, Fortran, Go, Java etc. 58

55. GLEW The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. 25

56. GLX The OpenGL extension to X11 (GLX) is a software interface which enables applications to access an OpenGL context inside a graphical window provided by X11. The GLX libary is usually provided by graphics card manufacturers as part of their drivers. 26

57. GPU A Graphics Processing Unit (GPU) is an electronic hardware circuit specialized for manipulating and drawing computer graphics. 1, 20

58. GUI A Graphical User Interface (GUI) allows the user of a computer program to interact with it using visual indicators and graphical icons instead of text-based user interfaces. 84

59. IBVP An Initial Boundary Value Problem (IBVP) is a system of differential equations constrained by initial conditions and boundary conditions unique for a specific problem. 8

60. JPEG Joint Photographic Experts Group (JPEG) is a popular file format for storing compressed bitmap images. 25

61. LBM The Lattice Boltzmann method (LBM) is a CFD method for fluid (e.g. liquid, gas and plasma) simulation. 1, 9, 51

62. LES Large Eddy Simulation (LES) is a mathematical model for fluid turbulence commonly used in CFD. 15

63. LibGL LibGL implements the GLX interface. Software based on OpenGL must link to this library to access its entry points. Responsible for dispatching OpenGL library calls to the graphics card driver. 27

64. Lua The (Lua) programming language is a lightweight general purpose multi-paradigm programming language with scripting, imperative, functional and object oriented features. 22

65. mutex Mutual Exclusion (mutex) is a concept in computer science which acts as locking mechanism allowing only one execution thread at time to manipulate shared memory. 32

66. OpenGL Open Graphics Library (OpenGL) is a cross-platform API for rendering 2D and 3D graphics. 2

67. PDE A Parital Differential Equation (PDE) is a differential equation with unknown multivariable functions and their partial derivatives. 1

68. pthreads POSIX Threads (pthreads), is a multi-threading execution model which is implemented in Linux, Mac OS X and Android among others. 32

69. RANS The Reynolds Averaged Navier-Stokes (RANS) equations model the motion of fluid flow by time-averaging and are commonly used to model turbulent flows. 7

70. RISE SICS North Research Institutes of Sweden, Swedish Institute of Computer Science, North, (RISE SICS North) is a research institute dedicated to the development of data centers and related technology, located in northern Sweden. 2

71. RMS The Root Mean Square (RMS) is a statistical measure defined as the square root of the mean of the squares of a set of numbers. 48

72. SM Streaming Multiprocessor (SM) is a GPU integrated circuit introduced with the NVIDIA Fermi architecture. 21

73. SSH Secure Shell (SSH) is a cryptographic network protocol for secure access and control of remote computer systems. 2

74. UNIX UNIX is a family operating systems that derive from the original AT&T Unix developed during the 1970s. The most popular member is called Linux, followed by BSD. 2

75. VirtualGL VirtualGL is an open source toolkit that gives any Unix or Linux remote display software the ability to run OpenGL applications with full 3D hardware acceleration. 2

76. VNC Virtual Network Computing (VNC) is a graphical desktop sharing system which allows keyboard and mouse events to be transmitted to a remote computer system, while receiving graphical screen updates. 2

77. X11 The X Window System (X11) is a graphical windowing system used primarily in UNIX-like operating systems like Linux. 25

78. Xlib Xlib is a C/C++ library which allows programs to interact with X11 events and commands, such as input devices and graphical window data. 27

Bibliography

[1] Eduardo W.V. Chaves. *Notes on Continuum Mechanics. Lecture Notes on Numerical Methods in Engineering and Sciences*. Springer Netherlands, 2013. ISBN: 9789400759862.

[2] CMake.org. *About CMake*. URL https://cmake.org/overview/ [Online; accessed 02-February-2018].

[3] NVIDIA Corporation. *CUDA C Programming Guide*. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [Online; accessed 01-June-2018]. 2018.

[4] Nicolas Delbosc. "Real-Time Simulation of Indoor Air Flow using the Lattice Boltzmann Method on Graphics Processing Unit". In: (2015). University of Leeds.

[5] Freedesktop.org. *libGL*. URL https://dri.freedesktop.org/wiki/libGL/ [Online; accessed 01-February-2018]. 2013.

[6] The Open Group. *Xlib - C Language X Interface*. URL ftp://www.x.org/pub/current/doc/libX11/libX11/libX11.html [Online; accessed 02-February-2018]. 2002.

[7] COMSOL INC. *The Boussinesq Approximation*. URL https://www.comsol.com/multiphysics/boussinesq-approximation [Online; accessed 28-May-2018]. 2015.

[8] Tamás István Józsa et al. "*Validation and Verification of a 2D Lattice Boltzmann Solver for Incompressible Fluid Flow*". In: (2016). National Technical University of Athens.

[9] Tsutomu Kambe. *Elementary fluid mechanics*. World Scientific, 2007. ISBN: 978-9812565976.

[10] NVIDIA Corporation Mark Harris. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*. URL https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/ [Online; accessed 16-April-2018]. 2015.

[11] Wolfram MathWorld. *Finite Volume Method*. URL http://mathworld.wolfram.com/FiniteVolumeMethod.html [Online; accessed 24-May-2018]. 1999-2018.

[12] A. A Mohamad. *Lattice Boltzmann Method. Fundamentals and Engineering Applications with Computer Codes*. Springer London, 2011. ISBN: 978-0-85729-455-5.

[13] The VirtualGL Project. *VirtualGL Background*. URL https://virtualgl.org/About/Background [Online; accessed 29-January-2018]. 2017.

[14] Roberto Benzi Sauro Succi Giorgio Amati. *Challenges in lattice Boltzmann computing*. 1994.

[15] Michael C. Sukop and Daniel T. Thorne Jr. *Lattice Boltzmann Modeling. An Introduction for Geoscientists and Engineers*. Springer Berlin Heidelberg, 2006. ISBN: 978-3-540-27981-5.

[16] Emelie Wibron. *A Numerical and Experimental Study of Airflow in Data Centers*. Luleå University of Technology, 2018. ISBN: 978-91-7790-063-4.