National Chiao Tung University

EECS International Graduate Program

Master's Thesis

混熱擾流力與分子動能模擬之 GPU 加速

GPU-Acceleration of the Hybrid Fluctuating Hydrodynamics and Molecular Dynamics Simulation

Student: Thomas Pak
Advisor: Chung-Ping Chung
Co-Advisor: Jhih-Wei Chu

July, 2016

混熱擾流力與分子動能模擬之 GPU 加速

GPU-Acceleration of the Hybrid Fluctuating Hydrodynamics and Molecular Dynamics Simulation

研 究 生：白托馬	Student: Thomas Pak
指導教授：鍾崇斌	Advisor: Chung-Ping Chung
指導教授：朱智瑋	Co-Advisor: Jhih-Wei Chu

國 立 交 通 大 學
電機資訊國際學位學程
碩 士 論 文

A Thesis
EECS International Graduate Program
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Electrical Engineering and Computer Science

July 2016

Hsinchu, Taiwan, Republic of China

中華民國一百零五年七月

摘要

分子層級的流體性質經常以全原子模擬(all-atom simulations)的方法進行研究，該方法以古典力學描述每一個原子的動態行爲，因此爲目標研究系統提供最清楚的動態資訊。另一方面，流體在巨觀的行爲則視流體爲一連續的變量，並藉由流體動力學方程式(hydrodynamic equations)預測隨時間下流體的表現。奈米層級下的探討則需要結合介於上述兩種尺度下的模型來進行分析。此一新的耦合系統對每個粒子配置了與其相對應的網格。粒子對場變數、或反之場變數對粒子的配對是透過粒子與場網格之內插來計算。然而此一耦合演算法尚未建立於高效能運算架構之下。

近年來，圖形處理器(graphics processing units, GPUs)儼然成爲科學計算上富競爭的平台。原本GPU是爲電腦繪圖而設計，但是GPU架構現在被優化於處理計算密集型任務與高通量資料。比起傳統的高效能運算叢集系統，上述特性使得GPU更爲吸引人且計算更有效率。因此新的運算架構即選擇了通用圖形處理器(GPU–CPU)之架構來進行混合模型之高效能運算。

本篇碩士論文討論以兩種力學的混合模型在GPU加速模擬下的設計與實踐。目的是要將原先之CPU演算架構改建置於可大量同步控制、以達到最高加速運算可能性之GPU演算架構。此一新的GPU演算架構使用共享記憶體爲暫存區域以對耦合系統進行快速地局部內插運算。兩階段執行緒對應將額外內存空間之使用降到最低以達到最大限度地提高運算處理量。藉由徹底地增加模擬的計算效率，利用混合模型所探索的時空間尺度將可以被大幅的增加。

Abstract

Fluid properties at the molecular scale are often investigated using all-atom simulations, which provide the highest level of detail attainable using classical mechanics. On the other hand, the behavior of fluids at the macroscopic scale is modeled by approximating the fluid as a continuous quantity and tracking its evolution by hydrodynamic equations. At the nanoscale both of these modeling paradigms are necessary. A hybrid model implementing molecular dynamics and hydrodynamics has previously been designed for simulations of nanoscale fluids. It implements a novel coupling scheme that associates a collocating grid with each particle. The mapping of particle to field variables and vice versa is then achieved through interpolation of particle and field grids. However, the coupling algorithm has not yet been adapted for high-performance computing (HPC).

In recent years, graphics processing units (GPUs) have emerged as a competitive platform for scientific computations. Originally designed for computer graphics, the GPU architecture is optimized for computationally intensive tasks and high data throughput. These features make them attractive and cost-effective alternatives compared to traditional HPC clusters. Therefore, a GPU–CPU framework was chosen as the HPC platform for the hybrid model.

This thesis thus presents the design and implementation of a GPU-accelerated simulation of the hybrid model. The objective was to reformulate the original CPU algorithms to expose massive concurrency, implement them on the GPU and achieve the highest computational speedup possible. A novel GPU algorithm was designed for the coupling scheme that uses shared memory as a staging area to perform fast local interpolations. To maximize computational throughput, a two-stage thread mapping was employed with a minimal amount of additional memory overhead. By drastically increasing the computational efficiency of simulations, the spatial and temporal scales that can be explored using the hybrid model were greatly expanded.

Acknowledgements

First and foremost, I would like to extend my deepest gratitude to professor Jhih-Wei Chu for agreeing to advise me on my thesis. I am honored that he let me join his research group and that he allowed me to be a part of the cutting-edge research that is taking place there. Not only was he a great mentor that taught me many things about biosystems and molecular simulation, he was also willing to share his advice on the pragmatic aspects of research.

I would also like to thank my advisor, professor Chung-Ping Chung, for introducing me to the world of computer architecture and giving me the opportunity to learn about the inner workings of a GPU. Without his valuable lessons it would have been impossible to achieve my research goals. I am also very thankful for his assistance in many practical matters related to the university and its administration.

I am deeply grateful to professor Hiroshi Watanabe for taking the time and effort to join my thesis committee as the committee chairman. I also thank him for teaching me how to be a critical researcher and for showing me how important it is to have a complete and detailed understanding of the underlying physics when modeling any sort of system.

Despite the immense distance, my KU Leuven co-advisors, professor Bart Nicolai and professor Dirk Roose, were always prepared to answer any question I had and I am very thankful to them for that.

I also want to express my gratitude towards Dr. Rasmus Persson, who, despite never having met me in person, was extremely patient and helpful while answering the many technical and theoretical questions that came up as I was familiarizing myself with the hybrid model.

Special thanks to professor Guido Groeseneken and the administration of the Faculty of Engineering at KU Leuven, Ms. Isabelle Benoit and Ms. Marie-Paule Buyse in particular, for giving me the incredible opportunity to participate in the dual-degree program with NCTU that enabled this thesis. At the other side, I would like to thank Ms. Angel Yu, Ms. Aileen Yu, Ms. Avis Liu and Ms. Janet Chen for assisting me throughout the various stages of the dual-degree program and my thesis defense. It was not always easy to coordinate the requirements and regulations of two universities at the same time, but fortunately both sides were helpful and understanding.

Many thanks as well to my labmates at the Biosystems Simulation Lab who made me feel welcome from day one and were always ready to help whenever I got stuck. I am especially thankful to Charlie Yu and Haoting Chang for assisting me with the Chinese abstract, and Yizao Chen for helping me organize my oral defense.

Finally, none of this would have been possible without the support of my parents, who have always believed in me and have never stopped encouraging me to explore my full potential. I am also grateful to my friends, in Belgium, Taiwan and elsewhere, who never cease to amaze me by being wonderful people.

Last but not least, I want to thank my significant other, Pamika Wantanaphan, for her unwavering support and endless love. I simply could not imagine going through this project without her being by my side.

This work was supported by the Ministry of Science and Technology of Taiwan, Republic of China with grant number 103-2628-M-009-003-MY3.

Table of Contents

List of Figures

List of Tables

List of Listings

I Introduction

The function of biological systems at the subcellular scale relies on ensembles of macromolecules interacting in an aqueous medium. The mathematical modeling of such systems is vital to understanding the mechanisms that underlie phenomena in this mesoscopic scale. However, the spatial scale in which these macromolecules reside and interact is neither atomic nor macroscopic. In such regimes, the events of interest occur at the junction where atomic and macroscopic behaviors meet and overlap. In order to simulate mesoscopic fluid dynamics efficiently, a hybrid model was proposed [21, 22, 18] to treat macromolecules as particles and the solvent medium as continuous density fields. Thermal effects were accounted for by incorporating stochastic fluctuations into the motion of particles and the evolution of the fields.

The particle-based part of the hybrid model is simulated using molecular dynamics (MD). In this approach, molecules are decomposed into particles that are subject to a certain force field [4, 8, 1]. This force field defines the energy potential of all intramolecular and intermolecular interactions. The particle trajectories are then calculated by deriving the forces acting on each particle and solving their equations of motion. Because the number of pairwise interactions in a N-particle system grows as $O(N^2)$, naively computing all of them is computationally intractable for most systems. Therefore, short-range interactions are only calculated for particles that are within a certain cutoff distance of each other. Efficient algorithms to find pairs of nearby particles are widespread in MD simulations and enable drastic reductions in time spent computing interactions [4, 8]. Electrostatic interactions on the other hand, are inherently long-range and require more sophisticated methods to compute efficiently. The Ewald summation [8] is such a method that splits up the electrostatic contribution of a charged particle into a fast-decaying part and a long-range part. The former is calculated using a real space sum, while the latter is calculated using a Fourier sum. In the particle-mesh method [8], the summation is accelerated further by assigning the charges to a discrete grid and using the fast Fourier transform (FFT) algorithm to compute the electrostatic potential. Lastly, thermal fluctuations are represented in the particle's equation of motion as a stochastic force [22].

A different approach was taken to simulate the dynamics of the solvent because using an explicit, all-atom MD model for the solvent requires a tremendous amount of computational resources. Moreover, most of these resources would be wasted calculating interactions between solvent molecules, which is significantly less important for nanoscopic phenomena than the interaction between macromolecules and the solvent. Therefore, a field-based model was developed to capture the essential physics of the solvent medium, while saving costs in computing its dynamics. In this approach, the solvent is not regarded as a large collection of solvent molecules, but rather as a continuous substance that obeys certain conservation laws and is described by a density field. More specifically, by applying the principles of mass conservation and of momentum conservation, a set of partial differential equations (PDEs) is obtained that dictates the evolution of the field. They are respectively known as the continuity equation and the Navier–Stokes equation. Together they are the fundamental equations of hydrodynamics (HD) [6]. General analytical solutions of the Navier–Stokes equation are not known and therefore they are solved numerically. In the current framework, the PDEs are discretized using a staggered grid and propagated in time using a second-order Euler predictor–corrector scheme [18]. The factors contributing to the evolution of the solvent density include convection, pressure and viscosity. Stochastic stresses were also added to model the effects of thermal fluctuations, which is the reason for the name *fluctuating* hydrodynamics (FHD) [22].

In the hybrid fluctuating hydrodynamics/molecular dynamics (FHD/MD) model, both the molecular trajectories and the solvent density fields are propagated in time within a common framework. The presence of the particles is manifested in the field equations as an occupied volume density that gives rise to a mass flux pushing the solvent out of the space occupied by particles. Conversely, the particles experience an equal, but opposite force that satisfies Newton's third law. Additionally, any difference in velocity between particles and the solvent induces a viscous drag force that opposes the movement of particles relative to the solvent. Put together, the trajectories of macromolecules can be determined at the atomic scale, while the solvent interactions are modeled in an accurate, yet cost-effective way [22, 18].

Both MD and HD are known to be computationally intensive problems, therefore many efforts have gone into devising efficient algorithms to tackle them [4, 8, 1]. Especially parallel algorithms are of interest because they allow simulations to be decomposed into smaller portions, which can then be distributed across a network of computing nodes [10]. Moreover, due to the aggressive pursuit of Moore's law, transistors have scaled down to such an extent that leakage currents are rampant in modern processors [13]. As a result, processor designers have shifted their focus from a few, very fast processor cores to many, more power-efficient computational units that perform calculations in parallel. The epitome of this parallel design paradigm is the graphics processing unit (GPU) architecture, which typically employs hundreds or even thousands of functional units and is capable of virtually all forms of parallelism. It was originally developed in order to handle rendering in 3D video games, but has since evolved to a general-purpose computational platform that provides high-performance computing at a low cost and consumes significantly less power than conventional CPUs. Due to these technology trends, it has become even more important to develop and implement parallel algorithms for intensive scientific computation [10, 13].

So far, MD and HD simulations have been successfully accelerated by GPUs [2, 12], but parallelization of the *hybrid* FHD/MD model has not yet been researched. In this thesis I have ported the hybrid FHD/MD model, which was implemented in FORTRAN 90 as a single-threaded CPU application in previous work, to CUDA C/C++, which is a programming language that allows the programmer to write general-purpose code for both the CPU and the GPU within a common framework. The most important part of this thesis is the parallel algorithm that I have designed to efficiently execute the particle–solvent coupling scheme.

The following two chapters introduce the hybrid model and its numerical solution. Chapter II defines the simulated variables, as well as the equations that govern their temporal evolution. It treats the particle and solvent models, as well their coupling, through an occupied volume field and by electrostatic interactions. Once the theoretical framework has been established, its numerical solution is discussed in Chapter III. In order to provide the proper background to the design rationale of the GPU implementation, an introduction to the GPU architecture is given in Chapter IV. The chapter focuses on the features and characteristics that are relevant to scientific computing. The CUDA C/C++ programming language used for general-purpose GPU programming is also briefly introduced.

Chapter V starts by laying out the design objectives for the GPU-accelerated simulation of the hybrid model. Then, the design and implementation of the GPU-implemented algorithms for the hybrid model are treated. This includes the novel GPU algorithm that interpolates the particle's occupied volume efficiently to the solvent field variables. The performance improvements are then displayed and discussed in Chapter VI. Finally, Chapter VII ends with an overall conclusion.

II Mathematical Modeling of Complex Molecular Systems at the Nanoscale

This chapter deals with the theory of the hybrid model in four major parts. First, the fluctuating hydrodynamics model is introduced. This component of the hybrid model simulates the dynamic state of the solvent. Then, the basic equations of molecular dynamics are formulated. They determine the motion of particles, as well as the forces that they exert on each other.

Section 2.3 deals with the theory of particle–solvent coupling that was developed by Chu et al. [21, 22, 18] It defines the forces that the solvent experiences due to particles and vice versa. Section 2.4 on the other hand is concerned with the electrostatic forces that originate from electrically charged particles and a polarized solvent.

Finally, the state variables that define the system are summarized in the last section and their governing equations are recapitulated, along with references to the relevant equations and subsections. A table listing all the physical constants appearing in the equations is also included.

2.1 Fluctuating Hydrodynamics

Hydrodynamics describe the transport of mass and momentum in a fluid using the continuum approximation. Treating the fluid as a continuum means that the fluid is considered to be a continuous substance that completely fills the space it inhabits. All of its properties, such as velocity and density, are continuously defined at every point of its spatial domain. They are therefore mathematically represented by fields and their behavior is defined by partial differential equations.

In particular, there are two types of equations needed to determine the evolution of density and velocity fields: governing equations and constitutive equations [6]. Sections 2.1.1 and 2.1.2 are dedicated to explaining their significance. In the subsequent subsections, they are applied to fluids derive the transport equations for mass and momentum.

In the hybrid modeling framework, the equations of hydrodynamics are used to model a solvent at the nanoscopic scale. At this scale, thermal fluctuations play an important role in the behavior of fluids. Therefore, stochastic forces were included in the equations for momentum conservation, which characterizes the modeling approach as a *fluctuating* hydrodynamics (FHD) model. The derivation of the fluctuating stress tensor using the fluctuation–dissipation theorem is found at the end of this section, followed by a summary of the FHD model.

2.1.1 Governing Equations of Conservation Laws

Transport equations rely heavily on two types of equations: **governing equations** and **constitutive equations**. Conservation equations express conservation laws relating the accumulation of a quantity—like mass or species concentration—to the rate at which it enters, or is formed in a region of space. Their general form is as follows:

$$\underbrace{\frac{\partial a}{\partial t}}_{\text{accumulation}} = \underbrace{-\nabla\cdot\mathbf{q_a}}_{\text{net flux}} + \underbrace{g_a}_{\text{net generation}} \tag{II.1}$$

In (II.1), a is some quantity of interest, $\mathbf{q_a}$ is the flux at which it moves around in space and g_a is the generation of that quantity. If a is a scalar quantity, $\mathbf{q_a}$ is a vector that specifies the rate and orientation of the flow of a. For an intuitive understanding of this equation, suppose a represents the local concentration of a chemical species. If the flux is completely homogeneous in space, for example moving in one direction at the same rate everywhere, there is as much of a entering into any pocket of space as there is leaving it. A homogeneous flux means that the divergence $\nabla\cdot\mathbf{q_a}$ equals 0, and as result the governing equation tells us that there is no accumulation of a.

Now suppose that the species is not moving uniformly, but that it is moving faster into a region of space then it is moving out. The divergence is then negative ($\nabla\cdot\mathbf{q_a} < 0$) leading to a net accumulation of a due to the minus sign in (II.1). When the species are on the other hand moving out at a higher rate than they are coming in, the signs are reversed and the concentration of a decreases over time.

If a is generated by some chemical reaction, the generation term g_a is positive, leading to an increase in concentration. Conversely, the consumption of a as a reagent in a chemical reaction is represented by a negative g_a.

2.1.2 Constitutive Equations

Governing equations are generally applicable equations that describe how a quantity will evolve in terms of its fluxes, but they do not say anything about transport phenomena themselves. To construct a hydrodynamic model therefore, additional equations are necessary that define these fluxes and how they relate to the properties of the material. These are called constitutive equations. Together with the governing equations, they fully determine the mechanics of a fluid.

In fluids, there generally two types of transport that constitutive equations need to characterize: "convective" transport and "molecular" or "diffusive" transport. Convective transport is the flux of some quantity due to the bulk movement of the fluid. As fluid flows from one region to another, it carries with itself all the quantities that it contains. For example, the convective flux of a chemical species in a fluid is given by the local flow velocity multiplied by the local species concentration.

Whereas convective transport is fairly easily described in a general and straightforward manner, molecular transport processes depend on specific material properties. Therefore they are usually characterized using empirical relationships. The transport coefficients that appear in them are in turn determined by experiments. Nonetheless, molecular transport processes share a universal, which is that they always dissipate spatial gradients of energy, matter or momentum over time. For example, species in a fluid will always migrate from higher concentrations to lower concentrations until the concentration profile is uniform throughout the entire fluid.

2.1.3 Conservation of Mass

The first component of the FHD model is the governing equation for the total mass in a fluid. The density of mass is denoted by ρ and the average fluid velocity by $\mathbf{v}$. By definition, there are no mass fluxes relative to the fluid velocity and therefore no diffusive fluxes. Additionally, there is neither creation nor destruction of matter. This leads to the following equation:

$$\underbrace{\frac{\partial \rho}{\partial t}}_{\text{accumulation}} = \underbrace{-\nabla \cdot (\rho \mathbf{v})}_{\text{convective flux}} + \underbrace{0}_{\text{generation}} \tag{II.2}$$

The product of $\mathbf{v}$ and ρ is the convective flux of mass entering or exiting any region in space. Since momentum is the product of velocity and mass, $\rho\mathbf{v}$ is equivalent to the concentration of momentum. Due to its special physical significance, it is treated as a distinct variable and given the symbol $\mathbf{g}$. (II.2) is therefore often written as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{g} = 0 \tag{II.3}$$

(II.3) is known as the continuity equation.

2.1.4 Conservation of Momentum

The continuity equation relies on the momentum density $\mathbf{g}$, to describe the temporal evolution of ρ. The second task of the hydrodynamic model is therefore to describe the evolution of $\mathbf{g}$. First, consider Newton's second law of motion for a body that has a constant mass m and is experiencing a force $\mathbf{F}$[1]:

[1] $\mathbf{v}$ and $\mathbf{p}$ are the body velocity and momentum respectively.

$$\mathbf{F} = m\frac{d\mathbf{v}}{dt} = \frac{d\mathbf{p}}{dt} \tag{II.4}$$

From (II.4), it can be seen that the rate of change in momentum of a body is equal to the net force acting on it. Therefore, deriving the governing equation for momentum density is the same as characterizing the forces acting on any part of the fluid.

There are two types of forces that affect fluids: body forces, which act on the entire volume of the fluid body, and stresses, which act on its surfaces. In view of (II.1), body forces correspond to the generation term as they "generate" momentum by applying an external force. Gravitational and electrostatic forces are examples of such body forces. On the other hand, stresses are diffusive fluxes, which transfer momentum from fluid regions with high momentum to regions with low momentum.

Since the hybrid model is concerned with modeling nanoscopic fluids, gravitational forces are disregarded. Leaving out thermal forces (Section 2.1.6), coupling forces (Section 2.3) and electrostatic forces (Section 2.4) for now, the basic governing equation for momentum density is given by:

$$\underbrace{\frac{\partial \mathbf{g}}{\partial t}}_{\text{accumulation}} = \underbrace{-\nabla\cdot(\mathbf{vg})}_{\text{convective flux}} \underbrace{+\nabla\cdot\boldsymbol{\sigma}}_{\text{diffusive flux}} \tag{II.5}$$

Because the conserved quantity $\mathbf{g}$ is a vector quantity, the fluxes $\mathbf{vg}$ and $\boldsymbol{\sigma}$ are given by second-order tensors rather than vectors. Whereas the flux for a scalar quantity is a vector quantity to specify the oriented flow of the scalar quantity, the flux for a vector quantity needs to be specified for every component in every direction and is therefore expressed as a tensor.

2.1.5 The Stress Tensor

The diffusive flux of momentum $\boldsymbol{\sigma}$ in (II.5) is called the **stress tensor** and relating this tensor to the dynamic conditions and material properties of a fluid is the final part in establishing a basic hydrodynamic model.

An important property of $\boldsymbol{\sigma}$ is that it contains two distinct types of stresses: **pressure** and **viscous stresses** [6]. Pressure is the only stress that remains when a fluid is at rest. Furthermore, pressure is a normal and isotropic stress, which means that for any point, it exerts the same stress perpendicular to any surface of any orientation. Viscous stresses on the other hand are only present when the fluid is moving relative to itself, i.e. when the fluid is being deformed, and also contain shear forces. As a result, $\boldsymbol{\sigma}$ is decomposed into:

$$\boldsymbol{\sigma} = \underbrace{-P\boldsymbol{\delta}}_{\text{pressure}} + \underbrace{\boldsymbol{\tau}}_{\text{viscosity}} \tag{II.6}$$

In (II.6), the tensor $\boldsymbol{\delta}$ is the identity tensor, ensuring that pressure exerts the same normal stress P in all directions, and $\boldsymbol{\tau}$ is the viscous stress tensor. The divergence of the $\boldsymbol{\sigma}$ in terms of $\boldsymbol{\tau}$ and P is:

$$\nabla \cdot \boldsymbol{\sigma} = \nabla \cdot (-P\boldsymbol{\delta}) + \nabla \cdot \boldsymbol{\tau} = -\nabla P + \nabla \cdot \boldsymbol{\tau} \tag{II.7}$$

Since P has the same meaning as thermodynamic pressure, the thermodynamic equation of state for a fluid serves as a constitutive equation. For the hybrid model, the following equation of state was used[2]:

$$P = c_l^2 \rho \tag{II.8}$$

To resolve $\boldsymbol{\tau}$, it is assumed that the fluid is Newtonian. This means that the magnitude of viscous stresses are linearly proportional to the local strain rate of the fluid. It can be shown [6] that this leads to the following constitutive equation:

$$\boldsymbol{\tau} = \eta^S(\nabla\mathbf{v} + (\nabla\mathbf{v})^T) + \left(\eta^B - \frac{2}{3}\eta^S\right)(\nabla \cdot \mathbf{v})\boldsymbol{\delta} \tag{II.9}$$

The coefficients η^S and η^B are the shear and bulk viscosity respectively.

2.1.6 The Fluctuating Stress Tensor

Any system with nonzero temperature experiences fluctuations. This is because of the kinetic energy contained in the vibrational and translational motion of matter at the molecular scale. At the nanoscopic scale, thermal fluctuations cause random variations of fluid momentum, which in turn thrusts its mass into random directions and generates fluctuations in mass density.

The properties of the thermal fluctuations are described by the fluctuation–dissipation theorem. It states that the fluctuation of a physical variable is related to the linear response of that variable to an external perturbation. For the FHD model, this means that the momentum fluctuation is expressed in terms of the stress tensor $\boldsymbol{\sigma}$. Accordingly, its mathematical representation is also a tensor, named the fluctuating stress tensor and denoted by $\boldsymbol{\sigma}^S$. The fluctuation–dissipation theorem dictates that its components σ_{ij}^S are normally distributed variables whose covariance is given by:

$$\begin{aligned}\langle \sigma_{ij}^S(\mathbf{x};t)\sigma_{kl}^S(\mathbf{x}';t)\rangle = {} & 2k_BT\delta(t-t')\delta(\mathbf{x}-\mathbf{x}') \\ & \times\left[\eta^S(\delta_{ij}\delta_{jl} + \delta_{il}\delta_{jk}) + \left(\eta^B - \frac{2}{3}\eta^S\right)\delta_{ij}\delta_{kl}\right]\end{aligned} \tag{II.10}$$

The subscripted delta functions δ_{ij} are equal to unity if and only if $i = j$. Likewise,

[2] c_l is the speed of sound in a fluid.

$\delta(\mathbf{x}-\mathbf{x}') = 1$ and $\delta(t-t') = 1$ when $\mathbf{x} = \mathbf{x}'$ and $t = t'$ respectively.

The constant k_B is called Boltzmann's constant. It relates the average thermal energy of a microscopic particle to the temperature T of its thermodynamical system. If the temperature is high, the molecules that make up the fluid possess more energy and therefore exhibit stronger fluctuations, which is reflected in a higher variance in (II.10).

2.1.7 Summary

The constitutive equations (II.8) and (II.9) together fully specify the stress tensor $\boldsymbol{\sigma}$. In turn this means that (II.5) is fully determined. Additionally, the fluctuating stress tensor $\boldsymbol{\sigma}^S$ is described by (II.10). In combination with (II.3), the equations of the FHD model are then given by:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \mathbf{g} \tag{II.11a}$$

$$\begin{aligned}\frac{\partial \mathbf{g}}{\partial t} &= -\nabla \cdot (\mathbf{v}\mathbf{g}) + \nabla \cdot (\boldsymbol{\sigma}^S + \boldsymbol{\sigma}) \\ &= -\nabla \cdot (\mathbf{v}\mathbf{g}) + \nabla \cdot \boldsymbol{\sigma}^S - \nabla P + \nabla \cdot \boldsymbol{\tau} \\ &= -\nabla \cdot (\mathbf{v}\mathbf{g}) + \nabla \cdot \boldsymbol{\sigma}^S - c_l^2 \nabla \rho \\ &\quad + \nabla \cdot \left[\eta^S(\nabla \mathbf{v} + (\nabla \mathbf{v})^T) + \left(\eta^B - \frac{2}{3}\eta^S\right)(\nabla \cdot \mathbf{v})\boldsymbol{\delta}\right]\end{aligned} \tag{II.11b}$$

It is important to note that (II.11a) and (II.11b) belong to a class of equations called the **Navier–Stokes equations**. These equations are significant because they have wide-ranging applications in engineering and physics, yet at the same time are poorly understood from a theoretical point of view. Analytical solutions only exist for a narrow set of problems and it is for this reason that computational techniques are an indispensable tool for studying fluids.

2.2 Molecular Dynamics

Molecular dynamics (MD) is a computational method for studying the dynamic properties of molecular ensembles by treating them as a classical many-body system. "Classical" here means that their motion is determined by Newtonian mechanics and that quantum effects are not explicitly accounted for. Fortunately, this is an excellent approximation for a wide range of materials. It is an especially useful tool when dealing with biomolecules because their molecular structure and function are tightly related. Accurate MD simulations thus provide valuable insight into the molecular mechanisms of biological processes. Moreover, MD methods can serve as exploratory tools to examine interactions between proteins and potential drugs *in silico* as an alternative to expensive drug-screening experiments.

However, treating every molecule as an individual particle has its drawbacks as well. It provides the large amount of detail needed to elucidate phenomena at the molecular scale, but at the same time requires immense amounts of computation. Not only can MD simulations easily involve millions of particles, the number of particle interactions to be calculated is even larger. As a result, molecular simulations tend to very computationally intensive.

2.2.1 Equations of Motion

In an MD simulation, every ith particle of the N_p-body system with mass m_i is associated with a velocity vector $\mathbf{v}_i$ and position vector $\mathbf{r}_i$. The forces experienced by particles cause them to move by Newton's second law[3]:

$$\mathbf{F}_i = m_i \frac{d\mathbf{v}_i}{dt} = m_i \frac{d^2\mathbf{r}_i}{dt^2} \qquad \text{for} \quad i = 1, \ldots, N_p \tag{II.12}$$

2.2.2 Force Fields

Defining the equations of motion for an MD simulation is relatively straightforward. However, the assignment of forces to particles is a much more difficult task. They are most commonly defined by a set of interatomic energy potentials that model the attraction and repulsion between pairs of atoms, but more sophisticated models may include other types of interactions. The functional form of these potentials and their parameters uniquely determine the mechanical characteristics of an MD simulation and is called a "force field".

For the hybrid model, it is assumed that all particles are identical and interact through the Lennard-Jones potential. This interatomic potential is a relatively simple model for the interaction between two neutral atoms or molecules. It is defined as:

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{II.13}$$

A plot of the Lennard-Jones potential is shown in Figure 1. As two particles approach each other from afar, they will initially feel an attractive force due to the potential energy well. The depth of this well is equal to ϵ. The equilibrium distance between two particles is at r_m, which can be calculated from σ as $r_m = 2^{1/6}\sigma \approx 1.122\sigma$. At a distance σ the potential energy has climbed back to zero and any closer approach is strongly disfavored by a steep, repulsive potential gradient.

The Lennard-Jones interatomic potential is a relatively simple model for the interaction between two neutral atoms or molecules. There is a medium-range attraction force,

[3]Particle mass m_i is actually not an explicit parameter in the simulation. Instead, the particle density ρ_i is specified and multiplied by the particle's coupling volume $V = \frac{4}{3}\pi R_{\text{coupling}}$. The particle "radius" R_{coupling} is defined later in Section 2.3.

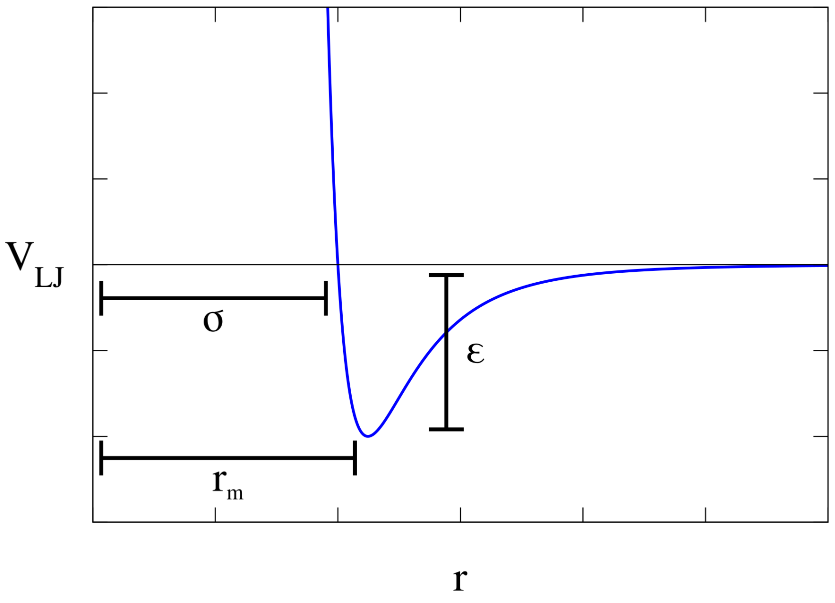


Figure 1: The Lennard-Jones interatomic potential

representing the van der Waals force, and a strong short-range repulsion force, which simulates the Pauli repulsion caused by overlapping electron orbitals.

The total force experienced by one particle is then the sum of the interactions with all other particles in the system[4]:

$$\mathbf{F}_i = \sum_{j \neq i}^{N_p} \frac{\partial V_{LJ}(r_{ij})}{\partial r_{ij}} \hat{\mathbf{r}}_{i \to j} \tag{II.14}$$

2.2.3 Periodic Boundary Conditions

When modeling phenomena at the nanoscopic scale, it is important to remember that the simulated domain can only be a tiny portion of the entire fluid. The simulated space is surrounded by fluid on all sides and the particles and the solvent should behave accordingly. This has important consequences for the boundary conditions of the model, because they define how the simulated system experiences its environment.

A common technique in MD is to use periodic boundary conditions[8, 1]. This means that the simulated box is actually a primitive cell in an infinitely repeating lattice. If a particle exits the domain at one side, its periodic image enters at the other side. Likewise, the values of the solvent field variables at the edge of the domain are identical to those at the opposite edge. This is schematically represented in Figure 2.

[4]Variable r_{ij} is the distance between particle i and j, and $\hat{\mathbf{r}}_{i \to j}$ is defined as unit vector with the same direction as $\mathbf{r}_j - \mathbf{r}_i$, i.e. pointing from particle i to particle j.

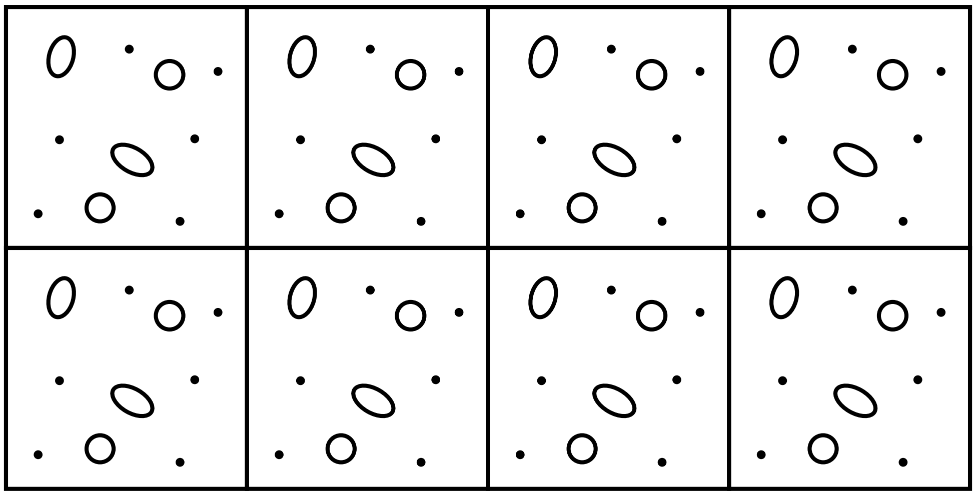


Figure 2: Schematic representation of periodic boundary conditions

In principle, this would mean that a particle interacts with all its mirror images in other cells, turning (II.14) into an infinite sum. Fortunately, the Lennard-Jones potential is a short-range interaction and thus the infinite sum can be avoided by confining its effects to a small volume centered on each particle. This is further explained in Section 3.5.

2.3 The Particle–Solvent Coupling Scheme

The FHD and MD model were established in sections 2.1 and 2.2 respectively. In the hybrid model, the strengths of these two models are combined to model the detailed molecular mechanics of macromolecular biomolecules using MD, while simultaneously simulating the encapsulating solvent environment with a FHD model to avoid the large cost of an explicit solvent model.

The central problem in the hybrid modeling methodology then becomes accurately mapping the entities from one model in the other model. Two major parts were implemented in the hybrid model to achieve this. First, a free energy density is introduced in the subsection below, that causes a diffusive flux directed outwards from the space inhabited by particles. This free energy also produces an opposite force felt by particles. Secondly, the frictional force that arises from velocity differences between particles and the surrounding fluid is discussed in Section 2.3.2.

2.3.1 Free-Energy Densities

In order to enable a solvent in the vicinity of a particle to separate into a solvent-free and solvent-rich region, the intrinsic free energy density of the fluid was defined to depend on a double-well potential:

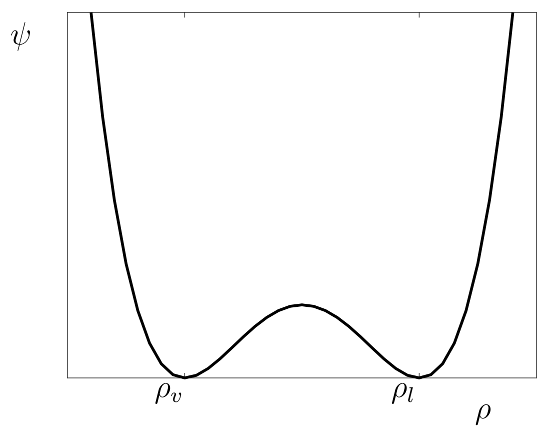


Figure 3: The double-well potential

$$\psi(\rho(\mathbf{x})) = \frac{C}{2}(\rho(\mathbf{x}) - \rho_l)^2(\rho(\mathbf{x}) - \rho_v)^2 \tag{II.15}$$

The effect of this potential is that the fluid can exist in two different stable states: a) $\rho = \rho_l$, where it is in a liquid state, and b) $\rho = \rho_v$, which corresponds to a low-density vapor state. The constant C modulates the energy barrier height between the two states. A plot of the potential is given in Figure 3.

Furthermore, the squared gradient of density was added to the intrinsic free energy density in order to stabilize the interface. The strength of this stabilization effect is controlled by the surface tension coefficient m. Altogether, the intrinsic free energy density is then written as:

$$F_0[\rho(\mathbf{x})] = \int \left[\psi(\rho(\mathbf{x})) + \frac{m}{2}(\nabla\rho(\mathbf{x}))^2\right] dV \tag{II.16}$$

The intrinsic free energy density $F_0[\rho(\mathbf{x})]$ is a functional that evaluates the integral in (II.16) to return the intrinsic free energy density of the system for a specific distribution of $\rho(\mathbf{x})$.

The presence of particles is manifested by a field variable that indicates the amount of space that is locally occupied by particles, denoted $v_{\text{occ}}(\mathbf{x})$. It is defined as[5]:

$$v_{\text{occ}}(\mathbf{x}) = \sum_i^{N_p} v_{\text{occ},i}(\mathbf{x}) \tag{II.17}$$

$$v_{\text{occ},i}(\mathbf{x}) = \begin{cases} 1 & \text{for } |\mathbf{r}_i - \mathbf{x}| \leq R_{\text{coupling}} \\ 0 & \text{for } |\mathbf{r}_i - \mathbf{x}| > R_{\text{coupling}} \end{cases} \tag{II.18}$$

[5]Variable $v_{occ}(\mathbf{x})$ is a "volume density". Its dimensions is Å^3 Å^{-3}, or equivalently, dimensionless. It can be thought of as indicating the presence of the particle in space. The integral $\int v_{\text{occ},i}(\mathbf{x})dV$ evaluates to the total volume of particle i.

The occupied volume density $v_{\text{occ}}(\mathbf{x})$ is a sum over all particles. The individual contribution of a particle $v_{\text{occ},i}(\mathbf{x})$ is represented by a sphere centered on the particle and bounded by the radius R_{coupling}.

A quadratic form was chosen to represent the effect of occupied volume in the particle–solvent free energy density[22]:

$$F_{ps}[\rho(\mathbf{x})] = \int \frac{k}{2} v_{\text{occ}}^2(\mathbf{x})\rho^2(\mathbf{x})dV \tag{II.19}$$

The constant k modulates the strength of the free energy penalty due to occupied volume.

The free energy contribution coming from the particles makes it unfavorable for the solvent to be present inside of the volume occupied by particles. Coupled with (II.16), the total free energy of the solvent causes it to segregate into a stable vapor state inside the particle and a liquid state surrounding it:

$$F[\rho(\mathbf{x})] = F_0[\rho(\mathbf{x})] + F_{ps}[\rho(\mathbf{x})] \tag{II.20}$$

The functional derivative $\delta F/\delta\rho$ returns a scalar field that indicates where an increase in ρ will lead to the largest increase in the free energy density F. Because the system seeks to minimize its free energy, the resulting diffusive flux is opposite to the gradient of $\delta F/\delta\rho$. This flux is added to (II.11a) to get[6]:

$$\frac{\partial\rho}{\partial t} = -\nabla\cdot\mathbf{g} - \nabla\cdot(\mathbf{J}+\mathbf{J}^S) \tag{II.21a}$$

$$\mathbf{J} = -\lambda\nabla\left(\frac{\delta F}{\delta\rho}\right) \tag{II.21b}$$

The constant λ is the diffusion coefficient of the fluid. It modulates the fluid's response to the gradient of $\delta F/\delta\rho$. A large diffusion coefficient will more quickly push the solvent out than a small diffusion coefficient. Because this process is a dissipating process that attempts to diminish gradients of $\delta F/\delta\rho$, the fluctuation–dissipation theorem requires the presence of a corresponding fluctuating flux, as denoted by $\mathbf{J}^S$. Its covariance is given by:

$$\langle \mathbf{J}_i^S(\mathbf{x};t)\mathbf{J}_j^S(\mathbf{x}';t')\rangle = 2\lambda k_B T\delta_{ij}\delta(\mathbf{x}-\mathbf{x}')\delta(t-t') \tag{II.22}$$

Equation (II.19) is a crucial step in defining the interactions between solvent and particle that also highlights the main difference between the FHD and MD model. In the MD model, particles do not have any physical volume. They "occupy" space by repelling

[6]Equation (II.21a) as such has a few issues when used in practice. The locally small fluid densities may cause spurious currents. Additionally, combined with fluctuating fluxes, negative fluid densities may appear inside particles. Both of these phenomena are unphysical and are an artifact caused by numerical discretization. These issues are addressed in Section 3.1.

nearby particles, but this volume is merely the result of a physical phenomenon rather than a physical property in itself. The volume V_{occ} is therefore a sort of pseudo-volume that acts as carrier for the force that the particle exerts on its environment.

The particle–solvent free energy density F_{ps} also determines the force felt by particles due to the solvent. Since the particles are defined to occupy a finite volume in space, the solvent force acting on them is likewise expressed by an integral over a region of space. Particles should only interact with the solvent through their surface, leading to the following expression:

$$\mathbf{F}_{ps,i} = \int_{S_i} \Big[-\nabla\Big(\frac{\delta F_{ps}}{\delta \rho}(\mathbf{x})\Big)\Big] dV \tag{II.23}$$

The integration volume S_i is defined as the "vicinity" of particle i's surface. The exact meaning of S_i and the evaluation of this integral will be discussed more deeply in Section 3.3.

2.3.2 Friction Forces

In addition to volume exclusion forces, particles also experience friction forces as they move through the solvent. For small particles, the friction force is given by Stokes' law, which states that the friction force is proportional to the particle's velocity relative to the fluid. The friction coefficient $\gamma = \frac{\gamma_0}{\rho_l}\rho$ is defined to have a linear dependence on the solvent density to model the effects of lower drag in regions where there is less solvent. Adding this to (II.23) gives:

$$\mathbf{F}_{ps,i} = \int_{S_i} \Big[-\nabla\Big(\frac{\delta F_{ps}}{\delta \rho}(\mathbf{x})\Big) - \frac{\gamma_0}{\rho_l}\rho(\mathbf{x})(\mathbf{v}_i - \mathbf{v}(\mathbf{x}))\Big] dV \tag{II.24}$$

Explicit position-dependent arguments $(\mathbf{x})$ were added to field variables to emphasize that $\mathbf{v}_i$ is the velocity vector of particle i, while $\mathbf{v}(\mathbf{x})$ is the velocity field of the solvent.

The origin of this friction force is similar to that of the stresses felt internally by the fluid due to velocity differences. The friction force dissipates the velocity gradients between particles and the surrounding fluid by transferring momentum. Likewise, the fluctuation–dissipation theorem predicts that there is a corresponding fluctuating force $\mathbf{s}(\mathbf{x};t)$ with covariance:

$$\langle s_i(\mathbf{x};t)s_j(\mathbf{x}';t')\rangle = 2\gamma k_B T \delta_{ij}\delta(\mathbf{x}-\mathbf{x}')\delta(t-t') \tag{II.25}$$

The full expression for the forces that the particle experiences due to the solvent is:

$$\mathbf{F}_{ps,i} = \int_{S_i} \Big[-\nabla\Big(\frac{\delta F_{ps}}{\delta \rho}(\mathbf{x})\Big) - \frac{\gamma_0}{\rho_l}\rho(\mathbf{x})(\mathbf{v}_i - \mathbf{v}(\mathbf{x})) + \mathbf{s}(\mathbf{x};t)\Big] dV \tag{II.26}$$

The integrand in (II.26) is a force density that acts on a surface-bound volume of the particle. By Newton's third law, an opposite force is exerted on the solvent. Therefore, a body force $\mathbf{f}_f$ needs to be added to (II.11b) that represents the reaction forces to (II.26). For every particle, $\mathbf{f}_{p,i}(\mathbf{x})$ is defined as the corresponding integrand in (II.26) inside S_i, and zero elsewhere. In other words:

$$\mathbf{f}_{p,i}(\mathbf{x}) = \begin{cases} -\nabla\left(\frac{\delta F_{ps}}{\delta \rho}(\mathbf{x})\right) - \frac{\gamma_0}{\rho_l}\rho(\mathbf{x})(\mathbf{v}_i - \mathbf{v}(\mathbf{x})) + \mathbf{s}(\mathbf{x};t) & \text{for } \mathbf{x} \in S_i \\ 0 & \text{for } \mathbf{x} \notin S_i \end{cases} \tag{II.27}$$

The total force density $\mathbf{f}_f$ that is felt by the fluid is then:

$$\mathbf{f}_f = -\sum_{i}^{N_p} \mathbf{f}_{p,i} \tag{II.28}$$

The definition of $\mathbf{f}_{p,i}$ in (II.27) ensures that $\mathbf{f}_f$ only receives the contribution of the integrand in (II.26) over the same volume S_i that is felt by particle i. As a result, momentum is conserved between the particles and the solvent.

2.4 Electrostatic Interactions

Biomolecular processes at the nanoscale rely heavily on electrostatic interactions. Charged residues are commonly found in biomacromolecules and play important roles in stabilizing their molecular structure, as well as in their interactions with other molecules. Implementing electrostatics into the hybrid FHD/MD model has two facets: calculating the electrostatic interactions between charged particles in the MD model, and simulating the dielectric response of the solvent to electric fields in the FHD model.

2.4.1 Charged Particles

When considering only the interaction between charged particles, calculating their dynamics is relatively straightforward; the force field is simply extended to include electric potentials. The potential due to particle i with charge q_i is derived from Coulomb's law and has the form[7]:

$$\phi_i(r) = \frac{q_i}{4\pi\epsilon_0 r} \tag{II.29}$$

The particle force equation (II.14) then contains additional terms to include electric forces:

[7] ϵ_0 is the vacuum permittivity

$$\mathbf{F}_i = \sum_{j\neq i}^{N_p} \frac{\partial V_{LJ}(r_{ij})}{\partial r_{ij}} \hat{\mathbf{r}}_{i\to j} + q_i \sum_{j\neq i}^{N_p} \frac{\partial \phi_j(r_{ij})}{\partial r_{ij}} \hat{\mathbf{r}}_{i\to j} \tag{II.30}$$

However, such an approach has two major shortcomings. First of all, equation (II.29) is only valid for charged particles in a vacuum. The system of interest however involves particles embedded in a dielectric solvent and therefore the vacuum permittivity is not appropriate. As a matter of fact, using any constant value for the permittivity is questionable because the polarization density of the solvent is heterogeneous at the nanoscale both in time and space.

The second shortcoming is a computational issue. In contrast to the Lennard-Jones potential (II.13), which decays as $1/r^6$, the electric potential levels off as $1/r^2$. This means that electric forces are not negligible even over long distances. Therefore the truncation method, discussed in Section 3.5, would severely undermine the accuracy of modeling electrostatic interactions. The alternative is to explicitly calculate all $O(N_p^2)$ pairwise electric forces, which is computationally intractable for large systems.

For these reasons, and also because it is more compatible with the FHD model, a field-based approach was chosen. Instead of calculating pairwise interactions, the particle charges are mapped to a charge density field $n_p(\mathbf{x})$[8]:

$$\tilde{n}_p(\mathbf{x}) = \sum_{i}^{N_p} q_i \delta_d(\mathbf{r}_i - \mathbf{x}) \tag{II.31}$$

The charge contribution of particles is added to the charge density field obtained from the FHD model (described below) to get the total charge density. Next, Poisson's equation is solved to obtain the electrostatic potential $\phi(\mathbf{x})$ from $n(\mathbf{x})$. Finally, the electric field is derived as $-\nabla\phi(\mathbf{x})$, which is evaluated once per particle to get the net electric force acting on it:

$$\mathbf{F}_i = \sum_{j\neq i}^{N_p} \frac{\partial V_{LJ}(r_{ij})}{\partial r_{ij}} \hat{\mathbf{r}}_{i\to j} - q_i \nabla\phi(\mathbf{r}_i) \tag{II.32}$$

This technique is called the **particle mesh Ewald method** and is widespread in MD methods. The reason for its success is that the numerical solution of Poisson's equation can be done efficiently using the fast Fourier transform (FFT) algorithm. The application of FFT in the hybrid model is discussed in Section 3.4.

2.4.2 The Polarization Density

A common drawback of MD simulations that do not include an explicit solvent is that electrostatic interactions are not accurately modeled due to the lack of a dielectric re-

[8]The function δ_d is the Dirac delta function.

sponse. In fact, their dielectric properties are one of the main reasons that solvents are modeled alongside macromolecules in the first place.

In order to simulate the dynamic dielectric properties of the solvent, the polarization density field $\mathbf{p}(\mathbf{x})$ is integrated in parallel with the other fluid state variables. The contribution to the charge density field is given by:

$$\tilde{n}_f(\mathbf{x}) = -\nabla \cdot \mathbf{p}(\mathbf{x}) \tag{II.33}$$

It is assumed here that only induced dipole moments contribute to the polarization density n_f, higher-order electrostatic moments are ignored. As a consequence, $\mathbf{p}$ only responds to the local electric field $\nabla\phi$ and the electrostatic energy density can be written as:

$$u(\mathbf{p}) = \mathbf{p} \cdot \nabla\phi + \frac{\mathbf{p} \cdot \mathbf{p}}{2n\alpha} \tag{II.34}$$

Here, α is the dipole polarizability, and n is the solvent number density. The latter can be derived from ρ as $n = \rho/m_{\text{solvent}}$, where m_{solvent} is the mass of a solvent molecule.

At equilibrium, the energy density $u(\mathbf{p})$ is minimized with respect to $\mathbf{p}$, which leads to the following equilibrium polarization density distribution:

$$\mathbf{p}(\mathbf{x}) = -\alpha n(\mathbf{x}) \nabla\phi(\mathbf{x}) \tag{II.35}$$

Based on these observations, the following overdamped Langevin equation was proposed as an *Ansatz* for the evolution of $\mathbf{p}$ [18]:

$$\underbrace{\frac{\partial \mathbf{p}}{\partial t}}_{\text{accumulation}} = \underbrace{-\nabla \cdot (\mathbf{v}\mathbf{p})}_{\text{convective flux}} \underbrace{-\eta^P\left(\nabla\phi + \frac{\mathbf{p}}{n\alpha}\right) + \mathbf{b}(\mathbf{x};t)}_{\text{generation}} \tag{II.36}$$

The second term on the right-hand side ensures that $\mathbf{p}$ evolves to (II.35) over time, and vanishes when (II.35) is achieved. The constant η^P is the mobility coefficient that determines the timescale at which $\mathbf{p}$ reaches equilibrium, and is related to solvent viscosity.

Since any deviation from the equilibrium polarization density is dissipated, the fluctuation–dissipation theorem necessitates a corresponding fluctuating term $\mathbf{b}(\mathbf{x};t)$, whose components have the following covariance:

$$\langle b_i(\mathbf{x};t) b_j(\mathbf{x}';t') \rangle = 2\eta^P k_B T \delta_{ij} \delta(\mathbf{x} - \mathbf{x}') \delta(t - t') \tag{II.37}$$

2.4.3 Poisson's Equation

This section has introduced the particle charge density $\tilde{n}_p(\mathbf{x})$ and the polarization charge density $\tilde{n}_f(\mathbf{x})$. The former is mapped from the spatial coordinates of charged particles using (II.31), and the latter is calculated from the polarization density $\mathbf{p}(\mathbf{x})$ according to (II.33). The total charge density is:

$$\tilde{n}(\mathbf{x}) = \tilde{n}_p(\mathbf{x}) + \tilde{n}_f(\mathbf{x}) \tag{II.38}$$

The electrostatic field $\phi(\mathbf{x})$ is related to the charge density $\tilde{n}(\mathbf{x})$ by Poisson's equation[9]:

$$\nabla^2\phi(\mathbf{x}) = -\frac{\tilde{n}(\mathbf{x})}{\epsilon_0} = -\frac{\tilde{n}_p(\mathbf{x}) + \tilde{n}_f(\mathbf{x})}{\epsilon_0} \tag{II.39}$$

The electric force $-\nabla\phi(\mathbf{x})$ is coupled back to the charged particles in the force equation (II.32), and to the dynamics of the polarization density by (II.36). Additionally, the electric force affects the dynamic state of the fluid momentum density as well[18]:

$$\frac{\partial \mathbf{g}}{\partial t} = -\nabla \cdot (\mathbf{v}\mathbf{g}) + \nabla \cdot (\boldsymbol{\sigma}^S + \boldsymbol{\sigma}) \underbrace{- \tilde{n}_f \nabla\phi}_{\text{electric force}} \tag{II.40}$$

It is important to note that charged particles, solvent and solvent polarization interact with each other through the common electrostatic potential field $\phi(\mathbf{x})$. Thus, the electric forces serve as a second coupling mechanism between particles and solvent.

2.5 Summary

In Section 2.1, the governing equations for solvent density $\rho(\mathbf{x})$ and momentum density $\mathbf{g}(\mathbf{x})$ were derived. The stress tensor $\boldsymbol{\sigma}$ and fluctuating stress tensor $\boldsymbol{\sigma}^S$ were obtained from the constitutive equations of Newtonian fluids.

Section 2.2 introduced the concept of force fields in MD and defined the Lennard-Jones potential used in the hybrid model. The equations of motion for particles were also given.

Both the FHD and MD models were coupled with the scheme described in Section 2.3. A free energy density F_{ps} was defined that causes solvent and particles to repel each other. Additionally, friction forces that arise from a velocity mismatch between particles and solvent were defined, as well as the corresponding fluctuating forces predicted by the fluctuation–dissipation theorem.

Finally, the electrostatic interactions between charged particles and a polarized solvent were outlined in Section 2.4. The polarization density $\mathbf{p}(\mathbf{x})$ was defined for this purpose, as was the electrostatic charge density $\tilde{n}(\mathbf{x})$ and electrostatic potential $\phi(\mathbf{x})$.

A total of five different state variables can be distilled from the hybrid FHD/MD model, as summarized in Table 2 (see Table 1 for a definition of the base units). Together, they completely define the state of the system at any point in time, as well as the deterministic forces that govern their temporal evolution. Their governing equations are summarized below.

[9]The numerical solution of Poisson's equation is discussed in Section 3.4.

Table 1: Base units

Physical quantity	Unit	Symbol	Value
Length	Ångström	Å	10^{-10} m
Time	Picosecond	ps	10^{-12} s
Mass	Atomic mass unit	amu	1.6605×10^{-27} kg
Charge	Elementary charge	e	1.6022×10^{-19} C

Table 2: State variables

Symbol	Type	Description	Units	Governing equation
$\rho(\mathbf{x})$	Scalar field	Solvent mass density	amu Å^{-3}	(II.21a)
$\mathbf{g}(\mathbf{x})$	Vector field	Solvent momentum density	amu Å ps^{-1} Å^{-3}	(II.41)
$\mathbf{p}(\mathbf{x})$	Vector field	Solvent polarization density	e Å Å^{-3}	(II.36)
$\mathbf{r}_i$	Vector	Particle position	Å	(II.12)
$\mathbf{v}_i$	Vector	Particle velocity	Å ps^{-1}	(II.12)

The evolution of ρ is determined by:

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \mathbf{g} - \nabla \cdot (\mathbf{J} + \mathbf{J}^S) \tag{II.21a}$$

$$\mathbf{J} = -\lambda \nabla \left(\frac{\delta F}{\delta \rho} \right) \tag{II.21b}$$

By definition, $\mathbf{g}$ is equal to $\rho \mathbf{v}$, and therefore the first term on right-hand side of (II.21a) is equal to the convective flux of ρ. The free energy density F in (II.21b) is defined by equations (II.20), (II.16), (II.19) and (II.15). $\mathbf{J}^S$ on the other hand, is characterized by (II.22).

The final governing equation for $\mathbf{g}$ is found by adding $\mathbf{f}_f$, defined in (II.28), to (II.40):

$$\frac{\partial \mathbf{g}}{\partial t} = -\nabla \cdot (\mathbf{v}\mathbf{g}) + \nabla \cdot (\boldsymbol{\sigma}^S + \boldsymbol{\sigma}) - \tilde{n}_f \nabla \phi + \mathbf{f}_f \tag{II.41}$$

Analogously to the previous equation, $\mathbf{vg}$ is the convective flux of $\mathbf{g}$. See (II.7), (II.8), (II.9) and (II.10) for the definition of $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}^S$. Sections 2.4 and 2.3 cover $-\tilde{n}_f \nabla \phi$ and $\mathbf{f}_f$ respectively.

The last field-based state variable is the polarization density $\mathbf{p}$. The proposed *Ansatz* for its governing equation is:

$$\frac{\partial \mathbf{p}}{\partial t} = -\nabla \cdot (\mathbf{v}\mathbf{p}) - \eta^P \left(\nabla \phi + \frac{\mathbf{p}}{n\alpha} \right) + \mathbf{b}(\mathbf{x};t) \tag{II.36}$$

Again, the first term **vp** is the convective flux of **p**. The case for the second term is made in Section 2.4.2 and the third term is specified by (II.37).

Finally, the particle state variables evolve according to Newton's second law:

$$\mathbf{F}_i = m_i \frac{d\mathbf{v}_i}{dt} = m_i \frac{d^2\mathbf{r}_i}{dt^2} \qquad \text{for} \quad i = 1, \ldots, N_p \tag{II.12}$$

To retrieve the total force $\mathbf{F}_i$ acting on particle i, the particle–solvent coupling force $\mathbf{F}_{ps,i}$, defined in (II.26), is added to (II.32) to obtain:

$$\mathbf{F}_i = \sum_{j \neq i}^{N_p} \frac{\partial V_{LJ}(r_{ij})}{\partial r_{ij}} \hat{\mathbf{r}}_{i \to j} - q_i \nabla \phi(\mathbf{r}_i) + \mathbf{F}_{ps,i} \tag{II.42}$$

The first force is the result of direct interactions with other particles through the Lennard-Jones potential, as defined by (II.13). The second force is the electric force that particle i experiences due to its charge q_i. Lastly, $\mathbf{F}_{ps,i}$ is the particle–solvent coupling force that models repulsion between particle and solvent, as well as friction and the corresponding fluctuating force. The in-depth treatment of electric forces and coupling forces was delivered in sections 2.4 and 2.3 respectively.

The set of all physical parameters encountered in the hybrid model, along with the values used in this work, is given by Table 3.

[10]Because of the periodic boundary conditions (Section 2.2.3, the total net charge of the system has to be zero. Otherwise, the total system would have infinite charge because the cell is part of an infinite lattice. As a result, there must be as many positively charged particles as negatively charged particles.

Table 3: Physical parameters

Symbol	Description	Value	First appearance
c_l	Speed of sound	$13.937\,\frac{\text{Å}}{\text{ps}}$	(II.8)
η^S	Shear viscosity	$19.204\,\frac{\text{amu}}{\text{Å}\,\text{ps}}$	(II.9)
η^B	Bulk viscosity	$53.858\,\frac{\text{amu}}{\text{Å}\,\text{ps}}$	(II.9)
k_B	Boltzmann's constant	$0.831\,447\,\frac{\text{amu}\,\text{Å}^2}{\text{ps}^2\,\text{K}}$	(II.10)
T	Temperature	$300\,\text{K}$	(II.10)
ρ_i	Mass density of particle i	$0.6023\,\frac{\text{amu}}{\text{Å}^3}$	(II.12)
ϵ	Depth of well in Lennard-Jones potential	$1.5 \times k_b T$	(II.13)
σ	Distance at which Lennard-Jones potential is zero	$40.0\,\text{Å}$	(II.13)
C	Determines height of energy barrier in double-well potential	$900\,\frac{\text{Å}^8}{\text{amu}^3\,\text{ps}^2}$	(II.15)
ρ_l	Equilibrium solvent density in liquid state	$0.6027\,\frac{\text{amu}}{\text{Å}^3}$	(II.15)
ρ_v	Equilibrium solvent density in vapor state	$0\,\frac{\text{amu}}{\text{Å}^3}$	(II.15)
m	Surface tension coefficient	$1100\,\frac{\text{Å}^4}{\text{amu}\,\text{ps}^2}$	(II.16)
R_{coupling}	Radius of coupling, determines size of occupied volume	$5\,\text{Å}$	(II.18)
k	Determines the strength of the free energy penalty for volume exclusion	$585\,\frac{\text{Å}^2}{\text{amu}\,\text{ps}^2}$	(II.19)
λ	Diffusion coefficient of solvent	$0.25\,\text{amu}\,\text{ps}$	(II.21b)
γ_0	Friction coefficient of solvent in liquid state	$3911.8\,\frac{\text{amu}}{\text{Å}\,\text{ps}}$	(II.24)
ϵ_0	Vacuum permittivity	$5.727\,657 \times 10^{-7}\,\frac{\text{ps}^2\,e^2}{\text{Å}^3\,\text{amu}}$	(II.29)
q_i	Charge of particle i [10]	$\pm 1\,e$	(II.29)
α	Dipole polarizability	$1.325 \times 10^{-3}\,\frac{e^2\,\text{ps}^2}{\text{amu}}$	(II.34)
m_{solvent}	Mass of solvent molecule	$18\,\text{amu}$	(II.34)
η^P	Mobility coefficient of polarization	$1.50 \times 10^{-6}\,\frac{e^2\,\text{ps}}{\text{amu}\,\text{Å}^3}$	(II.36)

III Numerical Solution of the Hybrid Model

Whereas the previous chapter introduced the governing equations of the hybrid FHD/MD model, this chapter explains how they are solved numerically. The first section addresses the numerical issues of small or negative fluid densities, a problem first alluded to on page 14. Section 3.2 then deals with the discretization of field variables. The formulation of gradient and divergence operators using finite differences is given as well.

The concept of Lagrangian grids is introduced in Section 3.3. This numerical technique is vital to the efficient implementation of the coupling scheme described in Section 2.3. Next, the fast Fourier transform method for solving Poisson's equation (Section 2.4.3) is discussed in Section 3.4.

Section 3.5 modifies the formulation of interparticle Lennard-Jones forces, given in Section 2.2, to enable drastic reductions in computational time using neighbor lists. The implementation of stochastic variables is briefly discussed as well in Section 3.6.

This chapter concludes with the numerical method used to propagate the hybrid FHD/MD model in time (Section 3.7). It is the final component of the numerical implementation that defines how the discretized state variables evolve given its time derivatives, as specified by their governing equations (Section 2.5). This finally leads to the establishment of a general procedure that integrates all of the calculations needed to numerically solve the hybrid FHD/MD model.

3.1 Auxiliary Fluids

As mentioned in the footnotes of page 14, equation (II.21a) has some issues when it is applied in practice. The low densities of the vapor phase occurring inside the volume of particles leads to numerical imbalances [19]. The stress tensor $\boldsymbol{\sigma}$ (II.6) in particular depends on ρ for the evaluation of the pressure field, as well as for the fluid velocity, derived from the momentum density as $\mathbf{v} = \mathbf{g}/\rho$. Steep gradients of ρ in a finite discretization scheme result in so-called spurious or parasitic currents. These are constant flows going in and out of phase boundaries where there should be none.

Additionally, in the governing equations of ρ there are no explicit constraints to enforce that ρ stays positive. Due to the presence of fluctuating fluxes in (II.21a), this means that negative density values are unavoidable inside particles.

These issues result in both erroneous results and numerical instabilities. Therefore, an approach that avoids dealing with negative or low densities altogether was implemented in the hybrid FHD/MD model [22]. Instead of treating the inside of particles as empty space in the FHD model, an auxiliary fluid is introduced that does not mix with the solvent and occupies the particle volume. The behavior of the auxiliary fluid is exactly like that of the solvent fluid, and all the same governing equations apply, except that the free energy density functional is altered so that it favors being *inside* the particle. Concretely, this means that there are two distinct free energy densities:

$$F^A[\rho(\mathbf{x})] = F_0^A[\rho(\mathbf{x})] + F_{ps}^A[\rho(\mathbf{x})] \tag{III.1a}$$

$$F^B[\rho(\mathbf{x})] = F_0^B[\rho(\mathbf{x})] + F_{ps}^B[\rho(\mathbf{x})] \tag{III.1b}$$

The intrinsic free energy density functionals F_0^A and F_0^B are exactly the same as defined in (II.16). However, the functional F_{ps}^B differs from (II.19):

$$F_{ps}^B[\rho^B(\mathbf{x})] = \int \frac{k}{2} v_{\text{occ}}^2(\mathbf{x})(\rho^B(\mathbf{x}) - \rho_l)^2 dV \tag{III.2}$$

The contribution of fluid-particle interaction free energy thus favors a liquid state within the particle, as this minimizes F_{ps}^B.

The total mass density of the fluid is then the sum of both fluids:

$$\rho(\mathbf{x}) = \rho^A(\mathbf{x}) + \rho^B(\mathbf{x}) \tag{III.3}$$

It is this aggregate fluid density which is used to evaluate the stress tensor and the pressure field. On average, it is equal to ρ_l because the auxiliary fluid fills in the gaps of the main fluid. The coupling forces in equation (II.26) however, are calculated using ρ^A. As a result, the particle–solvent interactions are conserved, while the technical issues associated with small numerical values are eliminated.

3.2 Staggered Grids

The partial differential equations (PDEs) that govern the evolution of the field variables in Table 2 are generally not solvable using analytical methods. Instead, the continuous fields are discretized into a finite number of finite volumes, called "cells". Each cell is associated with a number—multiple numbers if it is a vector field—that represents the value of the field in that cell, and has a predefined location relative to other cells. The arrangement of cells in space is called a "grid".

Finite difference formulas are used to express spatial derivatives as a linear combinations of cell values. Finally, these finite differences are inserted into PDEs in order to obtain approximate solutions of the PDE.

In this section, the staggered grid discretization scheme used in the hybrid model is introduced. Also, the basic field-based operations underpinning PDE calculations are discussed.

3.2.1 Definitions

Consider a rectangular box of width L_x, height L_y and depth L_z. The origin is placed at one of the corners of the box. The x-axis is defined to go along its width, the y-axis along its height and the z-axis along its depth. The domain is subdivided into cells by making N_x equally sized partitions along its x-axis, and similarly N_y and N_z partitions along the y- and z-axis respectively. This results in $N_x \times N_y \times N_z$, or N_{total} cells in total. The dimensions of each cell is then given by:

$$d_x = \frac{L_x}{N_x}, \quad d_y = \frac{L_y}{N_y}, \quad d_z = \frac{L_z}{N_z} \tag{III.4}$$

The vectors $\mathbf{d}_x$, $\mathbf{d}_y$, $\mathbf{d}_z$ and $\mathbf{d}$ are defined as follows:

$$\mathbf{d}_x = \begin{pmatrix} d_x \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{d}_y = \begin{pmatrix} 0 \\ d_y \\ 0 \end{pmatrix}, \quad \mathbf{d}_z = \begin{pmatrix} 0 \\ 0 \\ d_z \end{pmatrix}, \quad \mathbf{d} = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} \tag{III.5}$$

The volume of each cell is given by:

$$V_c = d_x d_y d_z \tag{III.6}$$

The indices i, j and k are used to index along the x-, y- and z-axis respectively. Following the convention of the C programming language, zero-based indexing is used. This means that:

$$i \in 0, 1, \ldots, N_x - 1; \quad j \in 0, 1, \ldots, N_y - 1; \quad k \in 0, 1, \ldots, N_z - 1 \tag{III.7}$$

Any set of coordinates (i, j, k) within this range uniquely identifies a cell. For instance, the cell that contains the origin in its corner has coordinates $(0, 0, 0)$. The cell diagonally opposite to that cell has coordinates $(N_x - 1, N_y - 1, N_z - 1)$.

Periodic boundary conditions are enforced as follows:

$$i + lN_x \mapsto i, \quad j + mN_y \mapsto j, \quad k + nN_z \mapsto k \qquad \text{for} \quad l, m, n \in \mathbb{Z} \tag{III.8}$$

For example, this means that the coordinates $(N_x + 2, -1, -4N_z + 3)$ are mapped to the cell at $(2, N_y - 1, 3)$.

Furthermore, each cell is assigned a unique global index $n_{i,j,k}$ as:

$$n_{i,j,k} = kN_yN_x + jN_x + i \tag{III.9}$$

The global index indicates the order in which the cells are arranged in memory. As a corollary, the x-dimension is the innermost dimension and the z-dimension is the outermost dimension.

When integrating over the cell (i, j, k), the integration volume $V^c_{i,j,k}$ is the volume spanned by:

$$\begin{aligned} \mathbf{x} \quad &\in \quad V^c_{i,j,k} \\ &\Leftrightarrow \\ id_x \leq x &\leq (i+1)d_x \\ jd_y \leq y &\leq (j+1)d_y \\ kd_z \leq z &\leq (k+1)d_z \end{aligned} \tag{III.10}$$

Lastly, within a cell the corner closest to the origin is indicated as the $(0, 0, 0)$-corner. This corner serves as the reference point for identifying the other corners. For example, the corner diagonally opposite to the $(0, 0, 0)$-corner is the (d_x, d_y, d_z)-corner. The corner along the positive x-direction from the reference is the $(d_x, 0, 0)$-corner.

3.2.2 Discretization over Staggered Grids

As mentioned in the introduction of this section, discretization schemes associate each cell with a single scalar value for a scalar field, and with a single vector for a vector field. Thus a continuous scalar field $s(\mathbf{x})$ is discretized into N_{total} values. The discrete value of $s(\mathbf{x})$ in cell (i, j, k) is indicated by $s_{i,j,k}$.

For a scalar field the location of each value is placed at the cell center. Thus:

$$s_{i,j,k} \mapsto i\mathbf{d}_x + j\mathbf{d}_y + k\mathbf{d}_z + \frac{1}{2}\mathbf{d} \tag{III.11}$$

Instead of mapping each vector to the center of the cell, the discretization scheme spatially separates their components and shifts them half a cell-width along its positive direction. Accordingly, the different components of a vector field $\mathbf{v}(\mathbf{x})$ are indicated by separate symbols: $v^x_{i,j,k}$, $v^y_{i,j,k}$ and $v^z_{i,j,k}$ for the x-, y- and z-components respectively. The

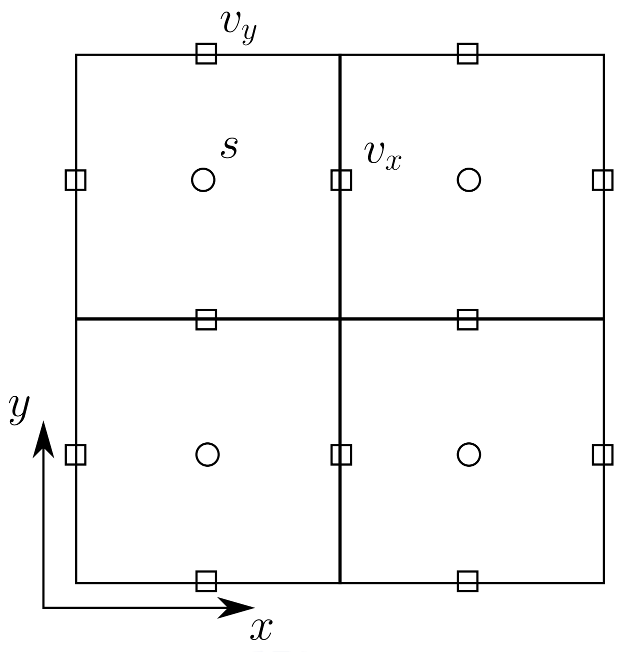


Figure 4: Staggered grid discretization. Scalar values are indicated by circles and vector components are indicated by squares.

shifted mappings are as follows:

$$v^x_{i,j,k} \mapsto i\mathbf{d}_x + j\mathbf{d}_y + k\mathbf{d}_z + \frac{1}{2}\mathbf{d} + \frac{1}{2}\mathbf{d}_x \tag{III.12a}$$

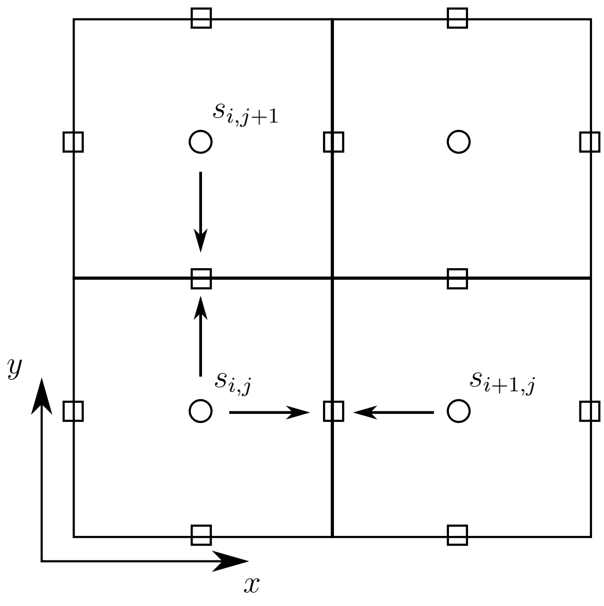
$$v^y_{i,j,k} \mapsto i\mathbf{d}_x + j\mathbf{d}_y + k\mathbf{d}_z + \frac{1}{2}\mathbf{d} + \frac{1}{2}\mathbf{d}_y \tag{III.12b}$$

$$v^z_{i,j,k} \mapsto i\mathbf{d}_x + j\mathbf{d}_y + k\mathbf{d}_z + \frac{1}{2}\mathbf{d} + \frac{1}{2}\mathbf{d}_z \tag{III.12c}$$

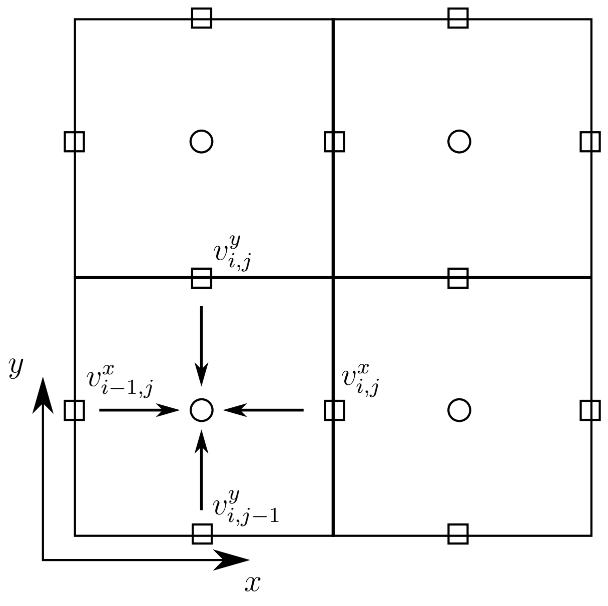
A two-dimensional visualization of the staggered grid is given in Figure 4. The circles indicate cell centers and squares represent cell faces. The locations of scalar and vector values are explicitly shown for cell $(0, 1)$. The center contains s_{01}, while the vector components v^x_{01} and v^y_{01} are separated and shifted to cell faces.

An equivalent way to describe the staggered grid is that vector fields are decomposed into a three scalar grids, each of which is shifted by $1/2 \cdot d_\alpha$, where α is the corresponding dimension of the vector component. The cell centers of the shifted grids then coincide with cell faces of the unshifted scalar grid.

The reason for using a staggered discretization scheme in the hybrid model is that the scalar quantity mass density ρ is carried from one cell to another by the vector quantity momentum density $\mathbf{g}$. Therefore, the causal relationship between flux and accumulation is best described by placing the flux at cell boundaries. The direction of flux is also respected by this scheme. For instance, the x-components of the flux are located between



(a) Gradient



(b) Divergence

Figure 5: Stencils for gradient and divergence

cells along the x-direction. It was shown by Voulgarakis et al. [21] that this scheme leads to improved accuracy in modeling transport phenomena at the nanoscopic scale.

3.2.3 Calculation of Gradients and Divergences

The staggered spatial arrangement of scalar and vector values means that the gradient and divergence operators do not map derivatives from the grid in the same manner.

The gradient operator is defined as follows:

$$\nabla s \equiv \left(\frac{\partial s}{\partial x} \quad \frac{\partial s}{\partial y} \quad \frac{\partial s}{\partial z}\right)^T \tag{III.13}$$

It maps a scalar field to a vector field whose components correspond to spatial derivatives of the respective dimension. Since the x-component of a vector is shifted by $\mathbf{d}_x$, it is located exactly in between the cell it is bound to and the neighboring cell along the positive x-direction. The staggered grid thus provides a natural way to construct the finite central difference from neighboring cells, as shown in Figure 5a. For scalar field s, the gradient $\mathbf{v} = \nabla s$ is defined as:

$$v^x_{i,j,k} = \frac{s_{i+1,j,k} - s_{i,j,k}}{d_x} \tag{III.14a}$$

$$v^y_{i,j,k} = \frac{s_{i,j+1,k} - s_{i,j,k}}{d_y} \tag{III.14b}$$

$$v^z_{i,j,k} = \frac{s_{i,j,k+1} - s_{i,j,k}}{d_z} \tag{III.14c}$$

Each component of the gradient is expressed as a forward sum along its respective dimension.

The divergence operator is defined as:

$$\nabla \cdot \mathbf{v} \equiv \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \tag{III.15}$$

As opposed to the gradient, the divergence operator maps a vector field to a scalar field by deriving each component to the corresponding derivative. It is used in conservation equations to get the accumulation from a flux field (see Section 2.1). Figure 5b shows that a central difference scheme arises trivially from the structure of the staggered grid. There is, however, an important distinction with regards to the gradient operator. The vector quantities are defined on the "forward" face of the cell, therefore to get the difference centered on the cell center, it needs to vector value of the cell "backwards" along its respective direction. The divergence is $s = \nabla \cdot \mathbf{v}$ is then defined as:

$$s_{i,j,k} = \frac{v^x_{i,j,k} - v^x_{i-1,j,k}}{d_x} + \frac{v^y_{i,j,k} - v^y_{i,j-1,k}}{d_y} + \frac{v^z_{i,j,k} - v^z_{i,j,k-1}}{d_z} \tag{III.16}$$

$$s_{i,j,k} = \frac{v^x_{i,j,k} - v^x_{i-1,j,k}}{d_x} \tag{III.17}$$

$$+ \frac{v^y_{i,j,k} - v^y_{i,j-1,k}}{d_y} \tag{III.18}$$

$$+ \frac{v^z_{i,j,k} - v^z_{i,j,k-1}}{d_z} \tag{III.19}$$

Both for the gradient and divergence operators, central differences need only the evaluation of the current cell's value and its direct neighbor. In a conventional grid, the central difference requires evaluations of both its forward and backward neighbor. Moreover, the spacing between function evaluations is consistently equal to the grid spacing for the staggered grid, whereas it is double the grid spacing for conventional grids.

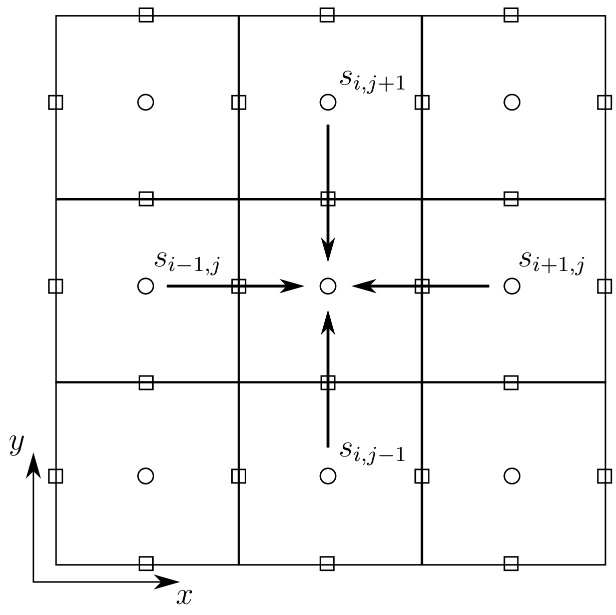


Figure 6: The Laplacian stencil

From a computational perspective, the staggered grid thus simplifies algorithms by requiring only the evaluation of the current cell and a single neighbor. From a numerical point of view, the scheme is more accurate than a conventional scheme. The staggered grid is thus clearly superior in computations involving gradient and divergence operations.

3.2.4 Calculation of Laplacians

The gradient operator followed by the divergence operator is defined as the Laplacian:

$$\nabla \cdot \nabla s \equiv \nabla^2 s \equiv \frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2} + \frac{\partial^2 s}{\partial z^2} \tag{III.20}$$

In accordance with the definition of the gradient and divergence operators, the Laplacian maps a scalar field to a scalar field. The result is a field equal to the sum of the second derivatives to each spatial coordinate. It is an important operator that appears in many physics equations, including Poisson's equation. The finite difference formula is derived by sequentially applying the schemes of the gradient and divergence. The Laplacian $a = \nabla^2 s$ is then:

$$a_{i,j,k} = \frac{s_{i+1,j,k} + s_{i-1,j,k} - 2s_{i,j,k}}{d_x^2} + \frac{s_{i,j+1,k} + s_{i,j-1,k} - 2s_{i,j,k}}{d_y^2} + \frac{s_{i,j,k+1} + s_{i,j,k-1} - 2s_{i,j,k}}{d_z^2} \tag{III.21}$$

The stencil is shown in Figure 6, and is identical to the stencil for a nonstaggered grid.

3.2.5 Multiplication and Division of Scalar and Vector Fields

Multiplication of scalar fields with scalar fields and vector fields with vector fields pose no numerical problems because their values are located at the same place. They can therefore be defined as simple element-wise operations.

However, the staggered grid makes the multiplication or division of a scalar field with a vector field less straightforward. The first step is an interpolation from the cell centers to the cell faces. For a discretized scalar field $s_{i,j,k}$, the resulting "vector field", given by $s^x_{i,j,k}$, $s^y_{i,j,k}$ and $s^z_{i,j,k}$ is defined as[1]:

$$s^x_{i,j,k} = \frac{s_{i+1,j,k} + s_{i,j,k}}{2} \tag{III.22a}$$

$$s^y_{i,j,k} = \frac{s_{i,j+1,k} + s_{i,j,k}}{2} \tag{III.22b}$$

$$s^z_{i,j,k} = \frac{s_{i,j,k+1} + s_{i,j,k}}{2} \tag{III.22c}$$

The multiplication or division of a vector field with a scalar field is then defined as the corresponding operation of two discretized vector fields, in which the scalar field is represented by the equation above. It should not however, be interpreted as an actual vector field, but rather as a technical representation of an interpolation to cell faces in three dimensions.

3.3 Lagrangian Grids

The grid used to describe solvent variables is called an Eulerian grid because they describe the flow of the solvent using a set of fixed points in space. The particles on the other hand are assigned mobile grids that move together with the particle. These grids are called Lagrangian grids. They serve to represent the particle's occupied volume in the underlying Eulerian grid of the solvent variables[2], and conversely provide a method to obtain field variables in the particle's frame of reference to calculate particle–solvent coupling forces[3].

The size of Lagrangian grid cells determines the resolution of the particle's spatial representation. A finer grid more accurately resolves the surface of the particle, but requires more parameters to store. A looser grid on the other hand provides a blurrier picture of the particle, but is less expensive to store and handle. Since the purpose of the Lagrangian grid is to interpolate variables to and from the Eulerian grid, there is no benefit for either grid to be finer than the other. Therefore the spatial resolution was

[1]The stencil is identical to that of the gradient operator and therefore Figure 5a applies here as well.
[2]Equation (II.18).
[3]Equation (II.26).

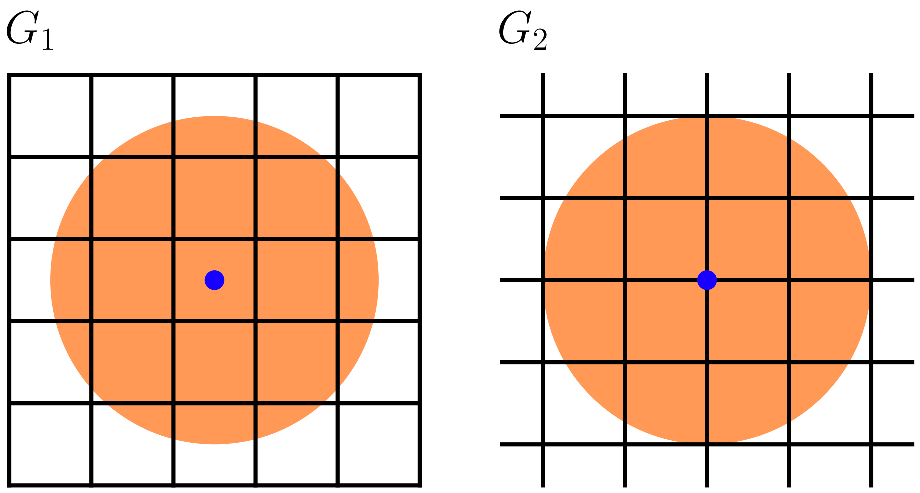


Figure 7: Lagrangian grids. Grid G_1 locates the particle at a cell center, while grid G_2 places the particle at a cell corner. The blue dot indicates the fixed position of the particle.

chosen to be consistent for Eulerian and Lagrangian grids.

3.3.1 Occupied Volumes

To efficiently compute the occupied volume on the Eulerian grid, it is first calculated from the perspective of the Lagrangian grid. The particle is fixed at some location in the grid, which can be in the center of a cell or at a cell corner. The volume density $v_{\text{occ},l}(\mathbf{X})$[4], as defined in equation (II.18), is then integrated for each grid cell separately. For a cell that is inside the particle, the occupied volume is equal to the cell volume V_c. The cells at the surface of the particle have an occupied volume between zero and V_c, and the cells outside of the particle have zero occupied volume.

Formally, the **Lagrangian** discretized occupied volume field is obtained for every particle l as[5]:

$$V^l_{i,j,k} = \int_{V^c_{i,j,k}} v_{\text{occ},l}(\mathbf{X})dV \qquad \text{(III.23)}$$

The result is illustrated in Figure 7.

Because the grid is fixed relative to the particle, $V^l_{i,j,k}$ only needs calculated once at the start of a simulation.

The procedure for calculating the **Eulerian** discretized occupied volume field $V^{occ}_{i,j,k}$ is as follows:

1. The Eulerian field $V^{occ}_{i,j,k}$ is initialized to zero.

[4]Capital $\mathbf{X}$ indicates that the field is evaluated in a Lagrangian grid. Furthermore, “fixing” the particle location means that $\mathbf{r}^{\text{lag}}_l$ is constant in the Lagrangian grid.

[5]For the definition of integration volume $V^c_{i,j,k}$, see Section 3.2.

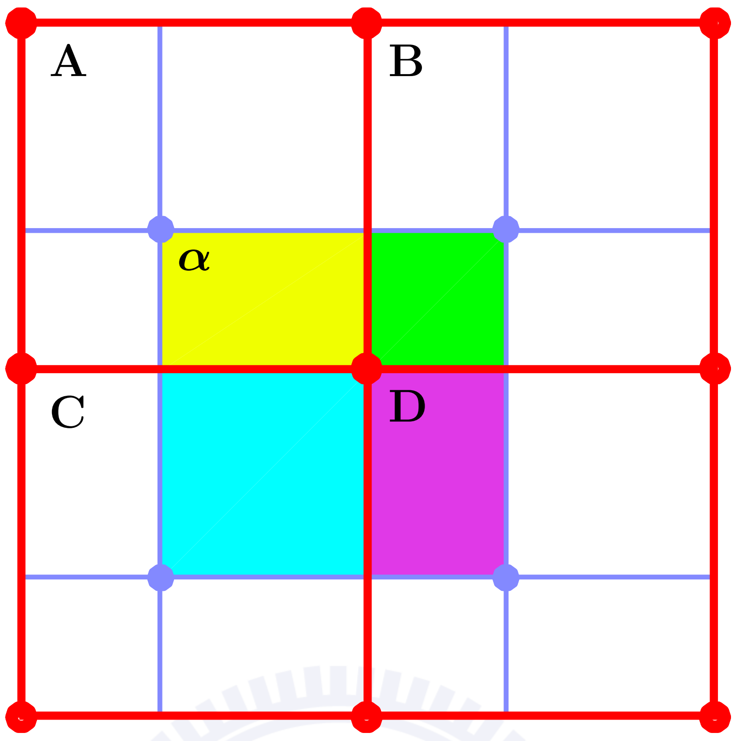


Figure 8: Interpolation between Lagrangian grid (blue) and Eulerian grid (red).

2. The Lagrangian grid of particle l is superimposed onto the Eulerian grid so that the fixed position vector in the Lagrangian grid coincides with the position vector $\mathbf{r}_l$ in the Eulerian grid.

3. The first Lagrangian cell of $V^l_{i,j,k}$ with nonzero occupied volume calculates its overlapping volume with the Eulerian cells that it lies in.

4. The occupied volume associated with the Lagrangian cell is added to the Eulerian cells, weighted by the corresponding fraction of overlapping volume.

5. Repeat steps 3 and 4 for all Lagrangian cells with nonzero occupied volume.

6. Repeat steps 2–5 for all particles.

In Figure 8, the interpolation step in two dimensions is visualized for the Lagrangian cell α that lies within the Eulerian cells indicated by A, B, C and D. The Eulerian grid is indicated using a red color and the Lagrangian grid is colored blue. The overlapping volumes computed in step 3 correspond to the colored areas.

The final field $V^{occ}_{i,j,k}$ thus indicates how much volume is occupied by particles in each cell. The particle–solvent free energy expression (II.19) deals with the occupied volume *density*, therefore the calculates occupied volume is divided by V_c to obtain the fractional extent to which an Eulerian grid cell is occupied by particles. If multiple particles overlap each other in the same cell, this quantity may be larger than unity.

3.3.2 Particle–Solvent Forces

The particle–solvent coupling scheme introduced in Section 2.3 integrates repulsion and friction forces over a volume S_l to get the total coupling force for particle l. The integration volume S_l was described as the "vicinity of the particle's surface", but was not specified further.

Because equation (II.26) concerns a particle force, it follows that the integration should be computed over a volume that consistently represents the particle's surface. The numerical integration is thus performed in the Lagrangian grid. The grid cells that constitute S_l are defined to be those cells that intersect only partially with the particle volume. Formally:

$$(i, j, k) \in S_l \iff 0 < V^l_{i,j,k} < V_c \tag{III.24}$$

The field variables that appear in (II.26) are interpolated to the Lagrangian grid by a similar procedure to the one described above but with the flow of data reversed. Once the partial force in a particle is computed, the opposite force is added to the force density $\mathbf{f}_f$.

1. The force $F_{ps,l}$ is initialized to zero.
2. The Lagrangian grid of particle l is superimposed onto the Eulerian grid so that the fixed position vector in the Lagrangian grid coincides with the position vector $\mathbf{r}_l$ in the Eulerian grid.
3. The first Lagrangian surface cell calculates its overlapping volume with the Eulerian cells that it lies in.
4. The field variables associated with the overlapping Eulerian cells are interpolated to the Lagrangian cell, weighted by the corresponding fraction of overlapping volume.
5. The integrand of (II.26) is evaluated using the interpolated field variables, and the resulting partial force is added to $F_{ps,l}$.
6. Repeat steps 3 and 5 for all Lagrangian surface cells.
7. Repeat steps 2–6 for all particles.

The pattern from Figure 8 also applies here to step 3. But instead of transferring the Lagrangian variables to the underlying Eulerian grid, the values of cells A, B, C and D are used to compute an interpolated value in cell α. In the configuration shown in the figure, the Eulerian cell C contributes the most, while cell B contributes the least.

3.4 Numerical Solution of Poisson's Equation

Section 2.4 introduced the charge density $\tilde{n}(\mathbf{x})$, as well as Poisson's equation:

$$\nabla^2\phi(\mathbf{x}) = -\frac{\tilde{n}(\mathbf{x})}{\epsilon_0} \tag{II.39}$$

The rest of that section assumed that Poisson's equation could be solved and paid no further attention to it. The electric field could then easily be derived from ϕ by applying the gradient operator. Therefore, the electrostatic forces in (II.32) and (II.40) rely on the solution of Poisson's equation.

Finding a field ϕ which satisfies (II.39) for an arbitrary field $\tilde{n}$ is not a trivial task however. Fortunately, Poisson's equation with periodic boundary conditions can be solved efficiently using Fourier transforms.

3.4.1 The Fourier Transform

Consider Poisson's equation in one dimension:

$$\frac{d^2\phi}{dx^2}(x) = \tilde{n}(x) \tag{III.25}$$

The Fourier transforms of $\phi(x)$ and $\tilde{n}(x)$ are defined as:

$$\hat{\phi}(k) = \frac{1}{\sqrt{2\pi}}\int \phi(x)e^{-ikx}dx, \qquad \hat{\tilde{n}}(k) = \frac{1}{\sqrt{2\pi}}\int \tilde{n}(x)e^{-ikx}dx \tag{III.26}$$

Applying the *inverse* Fourier transform then expresses the original functions in terms of their Fourier transforms:

$$\phi(x) = \frac{1}{\sqrt{2\pi}}\int \hat{\phi}(k)e^{ikx}dk, \qquad \tilde{n}(x) = \frac{1}{\sqrt{2\pi}}\int \hat{\tilde{n}}(k)e^{ikx}dk \tag{III.27}$$

Substituting these into (III.25) gives:

$$\begin{aligned} & (-k^2)\frac{1}{\sqrt{2\pi}}\int \hat{\phi}(k)e^{ikx}dk = \frac{1}{\sqrt{2\pi}}\int \hat{\tilde{n}}(k)e^{ikx}dk \\ \Rightarrow \quad & -k^2\hat{\phi}(k) = \hat{\tilde{n}}(k) \\ \Rightarrow \quad & \hat{\phi}(k) = -\frac{\hat{\tilde{n}}(k)}{k^2} \end{aligned} \tag{III.28}$$

As a result, $\phi(x)$ is given by:

$$\phi(x) = -\frac{1}{\sqrt{2\pi}}\int \frac{\hat{\tilde{n}}}{k^2}e^{ikx}dk \tag{III.29}$$

The general procedure for obtaining $\phi(x)$ from $\tilde{n}$ therefore starts by applying the

Fourier transform to $\tilde{n}$. Substituting the transformed variable into Poisson's equation then yields an expression for the Fourier transform of $\phi(x)$ as in (III.28). In the frequency domain, solving the Poisson's equation is thus reduced to solving an algebraic equation. Finally, $\phi(x)$ is obtained by applying the inverse Fourier transform.

3.4.2 The Discrete Fourier Transform

The discrete form of Poisson's equation is:

$$\begin{aligned}\nabla^2\phi_{i,j,k} &= \frac{\phi_{i+1,j,k}+\phi_{i-1,j,k}-2\phi_{i,j,k}}{d_x^2}\\ &+\frac{\phi_{i,j+1,k}+\phi_{i,j-1,k}-2\phi_{i,j,k}}{d_y^2}\\ &+\frac{\phi_{i,j,k+1}+\phi_{i,j,k-1}-2\phi_{i,j,k}}{d_z^2}\\ &=-\frac{\tilde{n}_{i,j,k}}{\epsilon_0}\end{aligned} \tag{III.30}$$

For more concise notation in the description of Fourier transforms, the following constants are used:

$$W_x = e^{2\pi i/N_x}, \quad W_y = e^{2\pi i/N_y}, \quad W_z = e^{2\pi i/N_z} \tag{III.31}$$

The discrete Fourier transforms (DFT) of $\phi_{i,j,k}$ and $\tilde{n}_{i,j,k}$ are three-dimensional grids of Fourier coefficients:

$$\hat{\phi}_{l,m,n} = \frac{1}{\sqrt{N_{\text{total}}}}\sum_{i=0}^{N_x-1}\sum_{j=0}^{N_y-1}\sum_{k=0}^{N_z-1} W_x^{-li}W_y^{-mj}W_z^{-nk}\phi_{i,j,k} \tag{III.32}$$

$$\hat{\tilde{n}}_{l,m,n} = \frac{1}{\sqrt{N_{\text{total}}}}\sum_{i=0}^{N_x-1}\sum_{j=0}^{N_y-1}\sum_{k=0}^{N_z-1} W_x^{-li}W_y^{-mj}W_z^{-nk}\tilde{n}_{i,j,k} \tag{III.33}$$

The original fields are returned by the inverse DFT:

$$\phi_{i,j,k} = \frac{1}{\sqrt{N_{\text{total}}}}\sum_{l=0}^{N_x-1}\sum_{m=0}^{N_y-1}\sum_{n=0}^{N_z-1} W_x^{li}W_y^{mj}W_z^{nk}\hat{\phi}_{l,m,n} \tag{III.34}$$

$$\tilde{n}_{i,j,k} = \frac{1}{\sqrt{N_{\text{total}}}}\sum_{l=0}^{N_x-1}\sum_{m=0}^{N_y-1}\sum_{n=0}^{N_z-1} W_x^{li}W_y^{mj}W_z^{nk}\hat{\tilde{n}}_{l,m,n} \tag{III.35}$$

Inserting these expressions into the discretized Poisson's equations provides an equation relating the Fourier coefficients of $\phi_{i,j,k}$ and $\tilde{n}_{i,j,k}$:

$$K_{l,m,n} \equiv -\epsilon_0 \Big[\frac{W_x^l + W_x^{-l}}{d_x^2} + \frac{W_y^m + W_y^{-m}}{d_y^2} + \frac{W_z^n + W_z^{-n}}{d_z^2} - 6\Big(\frac{1}{d_x^2} + \frac{1}{d_y^2} + \frac{1}{d_z^2} \Big) \Big] \tag{III.36}$$

$$K_{l,m,n} \hat{\phi}_{l,m,n} = \hat{\tilde{n}}_{l,m,n} \tag{III.37}$$

The Fourier coefficients $\hat{\phi}_{i,j,k}$ can then easily be found by solving the algebraic equation (III.37). Applying the inverse Fourier transform then gives the solution for $\phi_{i,j,k}$.

There is however a singularity in (III.37) for $l = m = n = 0$, because $K_{0,0,0} = 0$. This means that $\hat{\phi}_{0,0,0}$ is undetermined. From the definition of the DFT (III.32), it follows that this Fourier coefficient determines the average value of electric potential $\phi_{i,j,k}$. Since potential energy is only defined up to a constant, its average value over a field is arbitrary as well. Therefore the indeterminate status of $\hat{\phi}_{0,0,0}$ corresponds to the fact that the zero point of potential energy is arbitrary.

I chose $\hat{\phi}_{0,0,0}$ to be zero because subsequent calculations rely on the *gradient* of the electrostatic potential. The gradient is calculated using weighted differences of floating-point numbers (Section 3.2.2). These operations are the most accurate when the absolute values of the operands are small compared to their difference. Therefore, the floating-point accuracy of the electric field is maximal when the average value of $\hat{\phi}_{i,j,k}$ is equal to zero.

3.4.3 The Fast Fourier Transform Algorithm

The advantage of solving Poisson's equation using DFT is that there are algorithms that can perform such transformations very efficiently. A naive algorithm simply applies the definition of the DFT (III.32) and evaluates N_{total} grid points for each Fourier coefficient individually. As a result, the naive algorithm has time complexity $O(N_{\text{total}}^2)$.

Fast algorithms however, express the DFT operation as a matrix and then factorize the DFT matrix into sparse matrices. These algorithms can drastically reduce the time complexity to $O(N_{\text{total}}\log(N_{\text{total}}))$, and are thus named **fast Fourier transform** (FFT) algorithms.

A detailed description of the FFT algorithm is beyond the scope of this work, but can be found in the CUDA C programming guide [15].

3.5 Neighbor Lists

The particle force calculation (II.42) consists of three distinct terms: the interparticle Lennard-Jones forces, electrostatic forces and coupling forces. The numerical solution of the latter two have been discussed in Section 3.4 and 3.3 respectively.

The calculation of interparticle forces is straightforward in principle. However, its current formulation poses a significant problem from a computational perspective. Every particle calculates the force due to every other particle, which leads to $O(N_p^2)$ number of operations. For typical system sizes, this turns the force calculations into the dominant bottleneck. Therefore, two complementary techniques were applied to deal with interparticle forces: force truncation and cell lists.

3.5.1 Force Truncation

The shape of the Lennard-Jones potential as seen in Figure 1 indicate two important properties. First of all, the Lennard-Jones interaction is strongly repulsive at short distances. Secondly, the interaction decays rapidly beyond the equilibrium distance. As a result, particles tend to distribute themselves uniformly in space and moreover, their interactions tend to be dominated by a small number of nearby particles.

These observations suggest that computation can be scaled down by considering only the forces between particles that are sufficiently close to each other, in other words, if they are “neighbors”. For this purpose, the parameter cutoff radius R_{cutoff} is introduced. Forces are only calculated for particles i and j if $r_{ij} \leq R_{\text{cutoff}}$, and neglected otherwise. Because of the strongly repulsive short-range component of the Lennard-Jones potential, there is an upper bound on how many particles can surround a single particle in practice. Therefore, force truncation reduces the number of force calculations to $O(N_p)$ on average.

However, an algorithm implementing truncated forces needs to know which particles are neighbors. This information is provided by a data structure known as the neighbor list. For a given set of particle positions, the neighbor list records all pairs of particles that are within R_{cutoff} of each other [8, 1].

This setup moves the computational bottleneck from the force calculations to the generation of the neighbor list. A naive implementation iterates over all possible particle pairs to check if they are neighbors. However, the number of pairs to test are again $O(N_p^2)$, which means that the overall computational complexity for force calculations is unchanged unless a more sophisticated neighbor list algorithm is used. An algorithm based on “cell lists”, discussed below, is one such algorithm.

3.5.2 Cell Lists

The idea of the cell lists algorithm is to divide the simulation box into cells, similar to the field discretization in Section 3.2, and then keep track of the particles residing within each cell by a list. The cell dimensions are defined to be equal or slightly larger than R_{cutoff}. As a result, a neighbor pair must be in two cells that share a face, edge or corner. In other words, neighboring particle pairs are only found in neighboring cells The same notation as in Section 3.2 will be used to refer to grid properties for convenience.

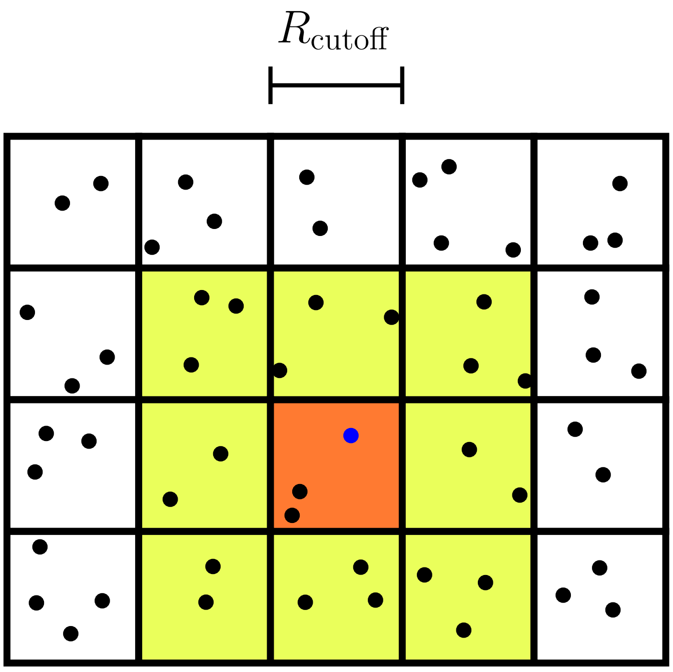


Figure 9: Neighbor search using cell lists. The blue particle only searches for neighbors in its own cell (orange) and the neighboring cells (yellow).

It is important to note however, that despite the similar structures, this grid is distinct from the discretized field grids defined in Section 3.2 because of the different cell width. There is no correspondence whatsoever between cell lists and fluid grid cells. For clarity, the grid used for the construction of the neighbor list will henceforth be referred to as the "particle grid".

Cell lists are constructed by looping over particle position vectors and finding the cells that the particles belong to. The particle grid coordinates can be found for any position within the simulation box with the following mapping:

$$\mathbf{r}_i \mapsto \left(\left\lfloor \frac{r_i^x}{d_x} \right\rfloor , \left\lfloor \frac{r_i^y}{d_y} \right\rfloor , \left\lfloor \frac{r_i^z}{d_z} \right\rfloor \right) \tag{III.38}$$

Every particle then adds its index to the list associated with that cell.

Using cell lists, the neighbor list is generated by looping over every particle and scanning the cell it belongs to and its neighboring cells to check for neighbors. This is illustrated for two-dimensional systems in Figure 9. In three dimensions, the number of cells that a given particle has to examine increases to 27, which includes its own cell.

3.6 Stochastic Fluxes

In Chapter II, a number of stochastic variables were introduced that were characterized by covariance equations. For instance, the particle coupling force equation contains a fluctuating force density $\mathbf{s}(\mathbf{x}; t)$ defined by:

$$\langle s_i(\mathbf{x};t)s_j(\mathbf{x}';t')\rangle = 2\gamma k_B T \delta_{ij}\delta(\mathbf{x}-\mathbf{x}')\delta(t-t') \tag{II.25}$$

The delta functions on the right-hand side are equal to unity only when their argument is zero, or when their subscripts are the same. Therefore, the three spatial components are statistically uncorrelated because $\delta_{ij} = 0$ for $i \neq j$. Additionally, the stochastic force is uncorrelated over both time and space. Altogether, each component of $\mathbf{s}(\mathbf{x};t)$ at any point in time or at any location is thus an independent stochastic variable.

Accordingly, the numerical solution for $\mathbf{s}(\mathbf{x};t)$ samples a univariate normal distribution $N \sim (0, \sigma^2)$ at every timestep. Because of spatial and temporal discretization (temporal discretization is discussed in the section below), the discretized stochastic variable represents the fluctuation in a finite volume V_c over a finite time duration dt. Therefore, the variance (II.25) is divided by the $V_c dt$ to get the final fluctuation:

$$\sigma^2 = \frac{2\gamma k_B T}{V_c dt} \tag{III.39}$$

The same reasoning applies to all other stochastic variables.

3.7 Temporal Propagation

An explicit predictor–corrector method is used to update the state variables with respect to time. Consider a simple differential equation:

$$y' = f(t, y(t)) \tag{III.40}$$

In this case, y is the state variables and (III.40) is the governing equation which computes its time derivative. The predictor step uses the state variables at time t to compute a “prediction” of the state variables at timestep $t + dt$.

$$\tilde{y}_{i+1} = y_i + dt \cdot f(t_i, y_i) \tag{III.41}$$

This step is identical to the explicit Euler method because it estimates the value of y_{i+1} based solely on the time derivative at t_i. The corrector step then uses the predicted state variable $\tilde{y}_{i+1}$ to estimate the time derivative at t_{i+1}. The two time derivatives for t_i and t_{i+1} are finally averaged to get the corrected state variable:

$$y_{i+1} = y_i + dt \cdot \frac{1}{2}\Big[f(t_i, y_i) + f(t_{i+1}, \tilde{y}_{i+1})\Big] \tag{III.42}$$

This also known as the trapezoidal rule. It improves the prediction made in the explicit Euler step by averaging estimates of the time derivative evaluated at different t and $t + dt$ to get a more accurate derivative for the time interval $[t, t + dt]$.

The overall procedure for numerically solving the hybrid FHD/MD equations under the predictor–corrector scheme is as follows:

1. Initialize state variables ρ^A, ρ^B, $\mathbf{g}$, $\mathbf{p}$, $\mathbf{r}_i$ and $\mathbf{v}_i$ to the user input.

2. Calculate the charge density $\tilde{n}$ by summing the contributions of charged particles and the polarization density, as described in Section 2.4.

3. Calculate the electrostatic potential ϕ from $\tilde{n}$ by solving Poisson's equation (Section 2.4 and 3.4).

4. Calculate the occupied volume field v_{occ} from particle coordinates (Section 2.3 and 3.3).

5. Calculate the particle–solvent forces $\mathbf{F}_{ps,i}$ by the algorithm described in Section 3.3, as well as the reaction force density field $\mathbf{f}_f$.

6. Construct cell lists and the neighbor list (Section 3.5).

7. Calculate the net particle forces $\mathbf{F}_i$ by evaluating (II.42). The electric potential and particle–solvent forces were calculated in steps 3 and 5 respectively. The Lennard-Jones forces are computed using the neighbor list generated in step 6.

8. Calculate the velocity field $\mathbf{v} = \mathbf{g}/(\rho^A + \rho^B)$.

9. Calculate the stress tensor $\boldsymbol{\sigma}$, fluctuating stress tensor $\boldsymbol{\sigma}^S$ and convective flux tensor $\mathbf{vg}$ (Section 2.1).

10. Calculate $\partial\mathbf{g}/\partial t$ according to governing equation (II.41). The tensors were calculated in step 9. The charge density and electric potential were obtained in steps 2 and 3 respectively. The particle–solvent force density was calculated in step 5.

11. Calculate the convective flux tensor $\mathbf{vp}$ and stochastic polarization flux $\mathbf{b}$ (Section 2.4.2).

12. Calculate $\partial\mathbf{p}/\partial t$ according to governing equation (II.36) using the quantities computed in step 11. The electric potential was obtained in step 3.

13. Calculate the mass fluxes $\mathbf{J}^A$ and $\mathbf{J}^B$ by taking the functional derivative of their respective free energy density functionals F^A and F^B, defined by (III.1a) and (III.1b), and inserting them into (II.21b). The occupied volume density field was calculated in step 4. Calculate stochastic fluxes $\mathbf{J}^{S,A}$ and $\mathbf{J}^{S,B}$ as well.

14. Calculate $\partial\rho^A/\partial t$ and $\partial\rho^B/\partial t$ from the fluxes in step 13 added to the convective flux $\mathbf{g}$.

15. The predicted state variables $\tilde{\rho}^A$, $\tilde{\rho}^B$, $\tilde{\mathbf{g}}$, $\tilde{\mathbf{p}}$, $\tilde{\mathbf{r}}_i$ and $\tilde{\mathbf{v}}_i$ are computed from the current state variables and the computed time derivatives as (III.41).

16. Repeat steps 2 to 14 with the predicted state variables.

17. The state variables are updated using the average of the time derivatives calculated for the current state variables and the predicted state variables according to formula (III.42).

IV GPU Architecture and the CUDA Execution Model

4.1 Overview of the GPU Architecture

Increases in microprocessor performance have traditionally come from improved processor cores with higher clock frequencies. The combination of Moore's law and the scaling laws of semiconductor transistors meant that processors became exponentially more powerful and faster over time.

The era of "easy scaling" has come to an end however. The feverish scaling of clock frequency has caught up with the semiconductor industry in the form of excessive heat dissipation and reliability issues. As a result, the focus of central processing unit (CPU) design has shifted from higher clock speeds to incorporating multiple CPU cores on a chip to improve performance.

The implications of this switch were profound not only for chip designers, but also for software designers. They were no longer able sit back and enjoy shorted execution time due to faster processors. Achieving meaningful speedups in application speed today requires programmers to restructure their software to run concurrent operations in parallel.

Meanwhile, a different type of architecture emerged that emphasized parallel execution even more strongly. Whereas multicore processors aim to maintain the fast execution speed of sequential programs while transitioning to multiple cores, graphics processing unit (GPU) architectures are focused on a high-throughput of many threads at the expense of the execution latency of individual threads. The result is that the potential throughput of GPUs is approximately ten times that of multicore processors. This does not necessarily reflect actual application speedup, but it does indicate the proportion of raw execution resources that can be exploited in these different types of processors.

The reason for the peak-performance gap between GPUs and multicore processors is their radically different design philosophies, as illustrated in Figure 10. Multicore processors dedicate a large portion of their transistors to control logic. This enables them to very rapidly process a sequential stream of instructions by instruction-level parallelism and branch prediction. Moreover, they have large caches to reduce the latency of instruction and memory fetches. This makes multicore processors flexible and fast. However, control logic and caches do not contribute to total execution throughput.

GPUs on the other hand focus on intensive floating-point processing workloads. The chip area for control logic and caches is smaller to make room for massive arrays of

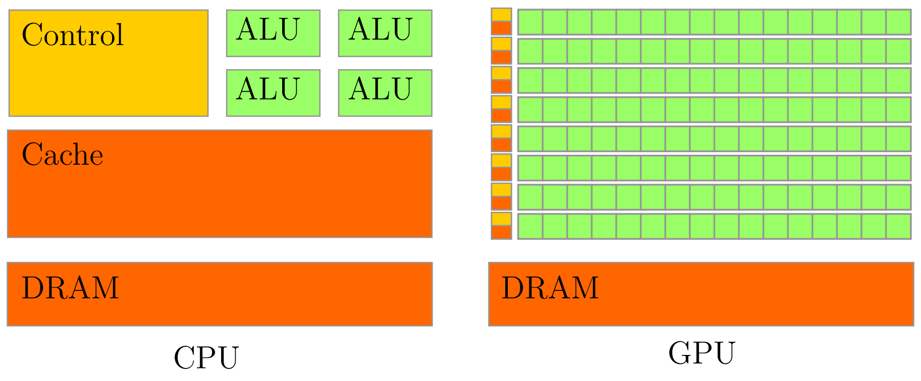


Figure 10: CPU versus GPU architecture

computational units. This accommodates the many-threads programming model, where concurrency is expressed by executing a large number of threads in parallel. The absence of large memory caches means that memory latency is higher compared to multicore processors. However, memory latency is by design less of an issue for GPUs because the large number of threads means program execution does not have to stall when fetching data. The GPU simply schedules threads which are ready to execute while other threads wait for their data. The lack of sophisticated control hardware on the other hand means that the threads running on the GPU are individually not as powerful as a thread running on a multicore processor.

Overall, these characteristics make GPUs well-suited for programs that operate on large amounts of data in a regular manner. It also explains why GPUs have become a popular platform for scientific computing. Computer simulations often operate on large amounts of data and usually involve many operations that can be executed concurrently. Indeed, scientific simulations have historically been among the most important applications of parallel computing in general.

4.2 The CUDA Execution Model

This section introduces the many-threads CUDA programming model using the CUDA platform. This platform was developed by NVIDIA to provide a general-purpose programming interface for their GPUs. The relationship between the programming interface and the underlying computing hardware is examined as well, with a focus on the features that the programmer has to pay attention to in order to maximize performance.

4.2.1 CUDA Threads

The building block of the CUDA execution model is the concept of the **CUDA thread**[13, 15, 5, 20, 14]. It is a programming abstraction that represents a single, independent path of execution. CUDA threads are programmed by writing **kernels**. The work done by a

Listing 1: Daxpy function in C

```
void daxpy(int n, double a, double *X, double *Y)
{
    for (int i = 0; i < n; i++)
        Y[i] = a * X[i] + Y[i];
}
```

Listing 2: Daxpy kernel in CUDA C/C++

```
__global__
void daxpy(int n, double a, double *X, double *Y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) Y[i] = a * X[i] + Y[i];
}
```

kernel often corresponds to the body of a loop in a conventional program. When a kernel is launched, the GPU generates thousands of threads that all execute the same kernel code. The GPU takes advantage of the fact that the threads are independent by running them concurrently.

For example, consider the calculation of $\mathbf{Y} = a\mathbf{X}+\mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ are large vectors with n elements. This operation is also known as the “daxpy” operation. A conventional C function would iterate over all elements using a loop (Listing 1). The CUDA kernel on the other hand specifies only the work for one element (Listing 2).

The reason that the daxpy operation is so easily parallelized, is because there are no dependencies between loop iterations. As a result, it does not matter in which order the iterations are executed. This makes it a straightforward task to turn each iteration into an independent thread. Tackling the computation with a many-threads approach then allows the GPU to take advantage of its large array of functional units and process the elements of $\mathbf{X}$ and $\mathbf{Y}$ in parallel.

Not all problems are as easily ported to the GPU as the daxpy operation however. More advanced problems do require coordination and data sharing between different threads. For this class of applications, the GPU offers features like shared memory, thread synchronization and atomic operations. These techniques are discussed in sections 4.3.3 and 4.3.5. In general however, thread cooperation introduces additional overhead and therefore should be kept to a minimum.

4.2.2 Thread Blocks and Grids

When launching a kernel, the programmer does not explicitly launch a certain number of threads. Rather, the programmer splits the threads into **thread blocks**, each containing

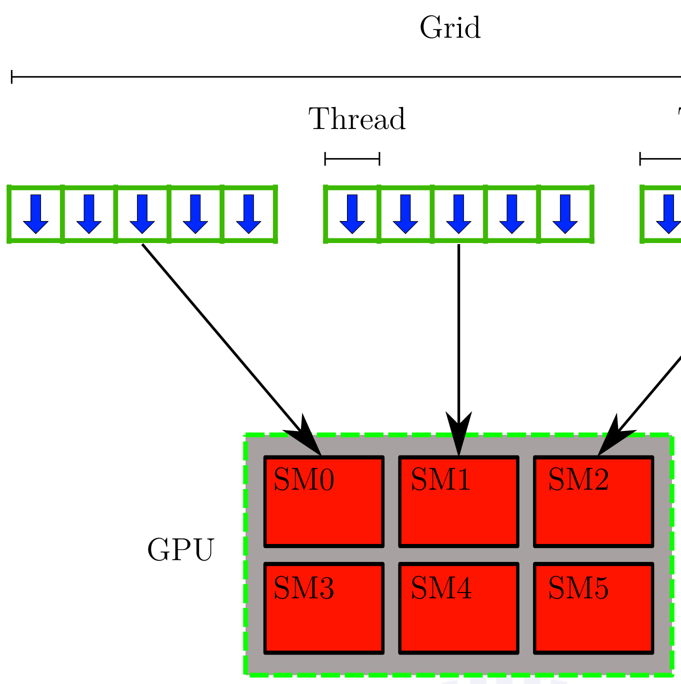


Figure 11: Organization of CUDA threads and scheduling of thread blocks

a maximum of 1024 threads. The collection of all thread blocks launched by a kernel is call the **grid**. Every thread is assigned two indices to distinguish it from other threads: a thread index, which uniquely identifies the thread within its thread block, and a block index that is unique to its thread block. Additionally, every thread has access to the block size and grid size as `blockDim` and `gridDim` respectively.

Continuing on with the example of the previous section, suppose that the vectors $\mathbf{X}$ and $\mathbf{Y}$ are 8192 elements long. If the block size is chosen to be 1024, $8192/1024 = 8$ thread blocks are needed to cover all vector elements. The appropriate kernel call is then as follows:

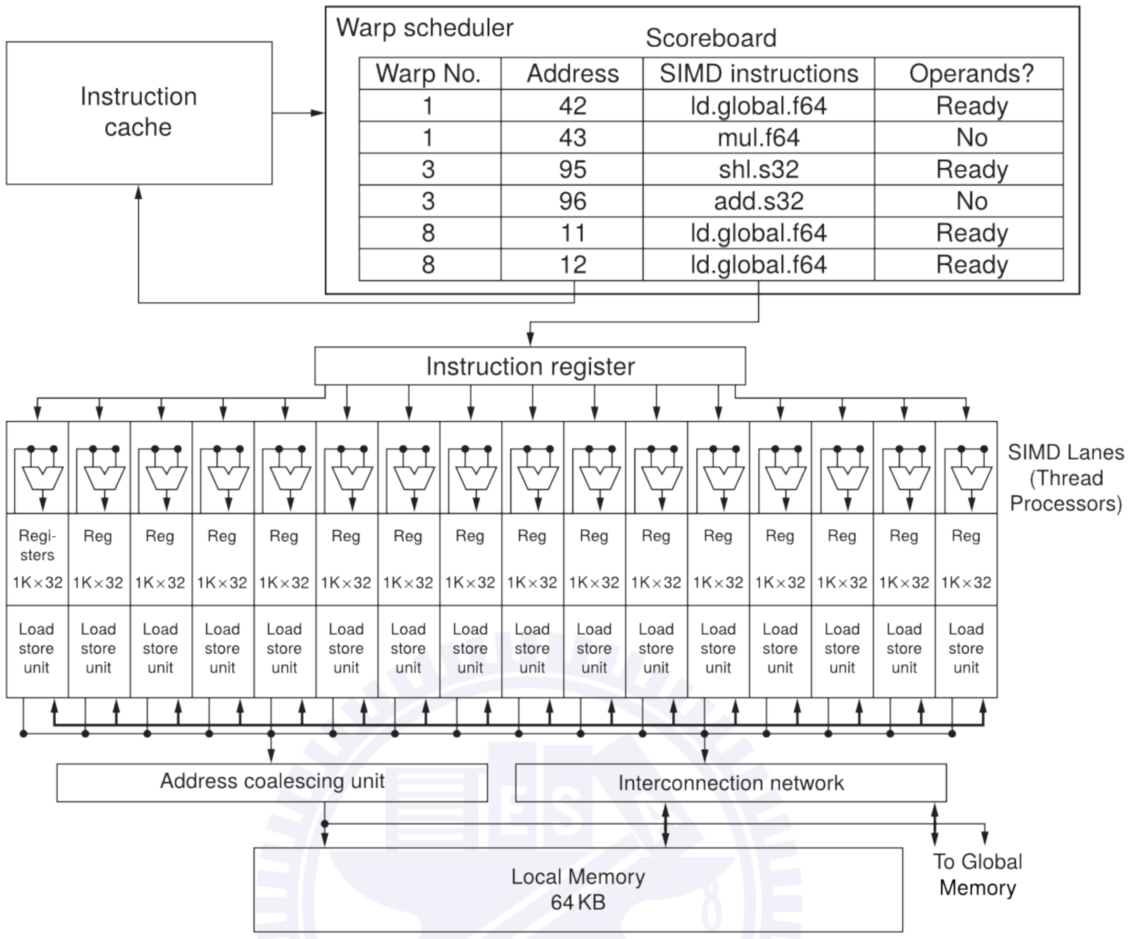
```
daxpy <<< 8, 1024 >>> (n, a, X, Y);
```

Every thread calculates its global index as specified by line 4 of the CUDA kernel in the previous section:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

For example, the thread in block 4 with thread index 215 calculates its global index as $4 \times 1024 + 215 = 4311$.

The CUDA execution model requires that these thread blocks are executed independently, which means that no communication occurs between thread blocks and that they can be processed in any order. This allows the GPU to assign thread blocks as discrete units of workload to any of its cores, called **streaming multiprocessors** (SMs), that have enough free resources. The distribution of computational workload is called scheduling and the control unit responsible for thread block scheduling is called the **GigaThread Engine**. A schematic drawing of this process is given in Figure 11.



Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

Figure 12: Simplified diagram of a streaming multiprocessor [13]

Because thread blocks are independent, CUDA kernels are very scalable across different GPUs. A small GPU might have only one or two streaming multiprocessors and can therefore only execute a few thread blocks in parallel. A GPU with more computational resources on the other hand could be capable of executing 64 or more blocks in parallel. An increase in streaming multiprocessors thus directly translates to faster kernel execution, without any changes to the code.

4.2.3 Streaming Multiprocessors

The GPU cores, or streaming multiprocessors (SMs), contain the computational resources necessary to execute thread blocks and also enable synchronization and communication between threads. An SM is best described as a *multithreaded SIMD processor*. A simplified diagram of the SM is given in Figure 12.

When a thread block is assigned to an SM, it is divided into groups of 32 threads, called **warps**. The threads in a warp are intimately bound and every instruction issued operates on them simultaneously. This is the meaning of SIMD (single instruction, multiple data) instructions. As a result, they step through their instructions together: one thread cannot advance further than the other threads in the warp. Another way to think of a warp is thus as a 32-wide SIMD thread. The functional units that execute SIMD instructions are called SIMD lanes or **CUDA cores**, as indicated in the diagram. The SIMD nature of thread execution is reflected in the figure as well, because there is only one instruction register for all CUDA cores and thus the same instruction is issued to all threads in parallel. Additionally, thread blocks usually contain many warps so streaming multiprocessors generally deal with multiple warps at the same time, this is the reason that they are considered *multithreaded* SIMD processors.

The scheduling of warps is done by the **warp scheduler**, also shown in the diagram. This is the second level of scheduling after the GigaThread Engine. Because CUDA threads are by definition independent, the warp scheduler can schedule any warp at any time and does not have to finish executing one warp before moving on to the next one. This has important implications for memory accesses. Instead of having a large cache that makes memory operations fast for a few threads, GPUs hide memory latency by having many threads overlap their memory requests. If there are enough threads, there are always warps ready to execute when other warps are waiting for their data.

The implication for the programmer is that he or she should aim to spread the computational workload over a large number of threads. Otherwise, there might not enough work available for the warp scheduler to effectively hide memory latency.

4.2.4 Branch Divergence

As mentioned in the previous subsection, the lowest level of execution is the warp, which means that the execution progress is the same over all threads within a warp. While this is advantageous for parallel processing, the SIMD nature of warps does pose some unique challenges when the code contains **conditional branching**.

Conditional branching occurs when threads take on different paths of execution depending on a conditional statement. A conventional thread on the CPU simply evaluates the condition and follows wherever it leads. However, a warp executes the same SIMD instructions over all its threads, and therefore its threads cannot be in different parts of the code at the same time. The GPU solves this problem by using a “thread-active mask”. This mask enables the GPU to selectively prevent individual threads from executing SIMD instructions.

For example, if the warp encounters an if-else construct, the GPU creates a mask that disables the threads for which the if condition does not hold and then continues executing the statements in the if-block. When it exits the if-block, the GPU inverts the mask and enters the else-block. This process is called **branch divergence** and is illustrated in Figure 13. If the if-condition is satisfied for the entire warp however, only the if-block

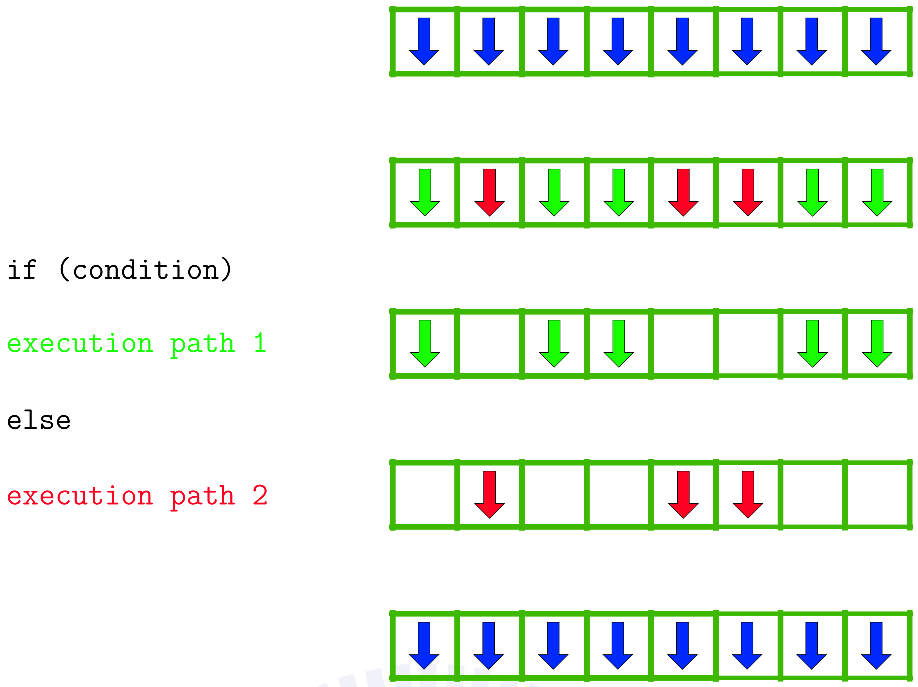


Figure 13: Branch divergence

is executed and the else-block is skipped. Conversely, when the if-condition evaluates to false for all threads in a warp, the GPU only executes the else-block.

It is important to understand that the GPU does not execute divergent branches across a warp in parallel. Instead, the different branches are executed sequentially with different masks. This means that the total number of instructions executed is the sum over all paths taken by branches in a given warp. Repeated branch divergence may cause only a few threads to be active of the potential 32 threads in a warp, resulting in a drastic drop in performance. A similar situation occurs in a loop that assigns a different number of iterations over the threads in a warp. Initially, all threads are active, but the threads with smaller workloads exit the loop before the others do. The rest of their time is then spent idling while waiting for the other threads to terminate the loop. Overall warp execution is thus limited by the rate of the slowest thread in the warp. The best strategy for maximal performance is therefore to keep the control flow at warp granularity as much as possible and distribute workloads evenly across threads in a warp.

4.3 The CUDA Memory Model

This section deals with the CUDA memory model. First, the concept of the memory hierarchy is explained. Then, the different CUDA memory types are introduced and placed into the CUDA memory hierarchy. Finally, atomic operations and optimal data access patterns are discussed in the last two subsections.

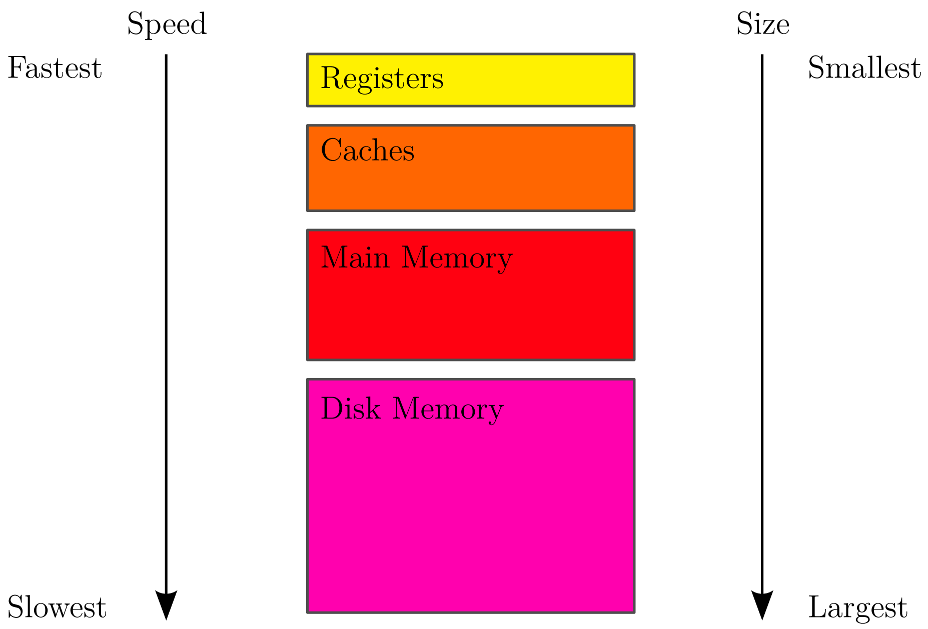


Figure 14: The CPU memory hierarchy

4.3.1 The Memory Hierarchy

Memory access is very important for high-performance computing because many computations tend to be “memory bound” [13]. This means that the runtime is not limited by the time spent processing data, but rather by the time spent transferring data between the processor and main memory. This is because dynamic random access memory (DRAM) chips operate at a much slower rate than processors. Faster memory technologies are available, but they are not suitable for storing main memory.

The reason for this is that there is a trade-off between latency and capacity. For the same cost, low-latency memory holds less data than higher-latency memory. Building a computer with only low-latency memory therefore severely limits the amount of memory available. On the other hand, a computer that relies solely on large-capacity memory chips would run unacceptably slow.

The solution is to use a hierarchy of memory types that places a large amount of high-latency memory at the bottom, and progressively faster, but smaller memories leading up to the processor (see Figure 14). The memory hierarchy is optimal for memory access patterns that display spatial and temporal locality. Spatial locality means that a program is likely to reference data elements that are close together, while temporal locality assumes that a program tends to operate on the same data within a certain time frame.

The memory hierarchy exploits locality by trickling a memory request from the CPU down the hierarchy until it finds the requested data. When it is found, the block of memory containing the data is propagated up to the highest level. Future requests to that block of memory are then readily serviced by fast memory structures, until the block is displaced by other memory blocks.

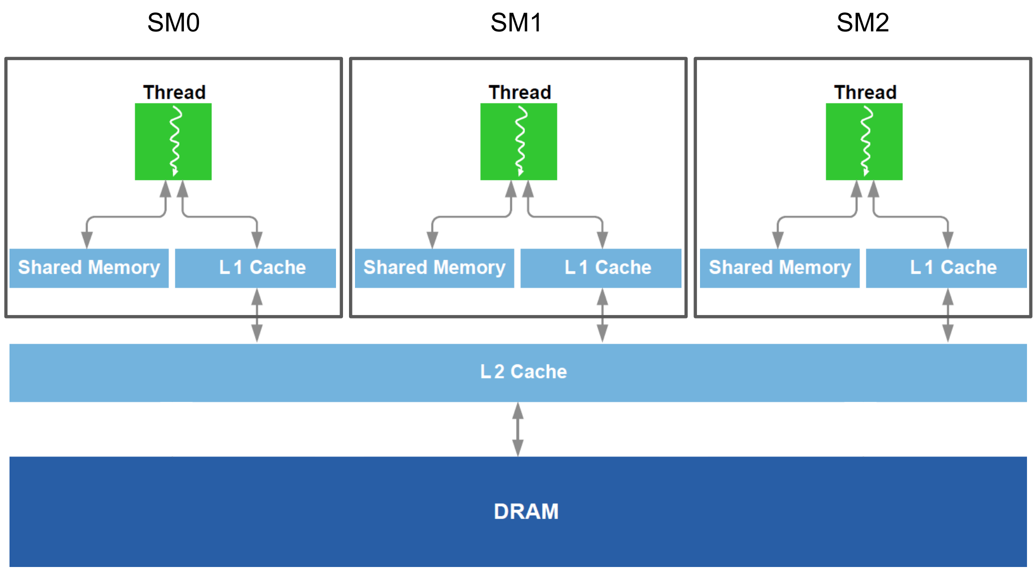


Figure 15: The CUDA memory hierarchy [17]

The layers between main memory and the processor are called “caches”. The cache closest to the CPU is called L1 cache, and the caches further down are labeled L2 cache and L3 cache respectively. These are usually made from static random access memory (SRAM) chips. The structure of a SRAM memory cell makes them much faster than DRAM memory, but also more expensive because they require more transistors per stored bit.

Inside processors, data is stored in registers. These are often implemented as fast SRAMs with multiple ports. They are the fastest type of memory available and are intimately bound with the computer architecture. Their role is not so much to store data as it is to temporarily hold variables for the processor to operate on. Therefore they are usually not regarded as part of the memory hierarchy, but rather as an integral component of the processor.

As mentioned in Section 4.2.3, GPUs rely less on caching and instead depend on massive parallelism to hide memory latency. Streaming multiprocessors, as their name suggests, were primarily designed to process long streams of data instead of operating repeatedly on the same piece of data.

Instead, caches in GPUs have more diverse roles. Moreover, the CUDA memory hierarchy contains caches that are are directly exposed to the programmer. This is in stark contrast to the L1 and L2 caches in CPUs, which follow rules that the programmer has no control over.

4.3.2 Global Memory and Local Memory

Global memory sits at the bottom of the GPU memory hierarchy (Figure 15) and is analogous to main memory in CPUs. Like CPU memory, global memory is composed of

DRAM chips. The name “global” refers to its scope; global memory is available to every SM and exists for the entire duration of a program. It is the largest memory present in the GPU, but also has the highest latency.

In order to reduce pressure on global memory, a L2 cache was introduced in the Fermi architecture [17, 13]. It is responsible for caching global memory accesses for all SMs. They are particularly significant for general-purpose applications because they significantly accelerate atomic operations in global memory (see Section 4.3.5).

Local memory is memory that is privately allocated to threads. It is primarily used when threads require more memory than what is available to them in the register file, a phenomenon called “register spilling”. The name is somewhat misleading because local memory is allocated on the same DRAM chips as global memory. Therefore, they can be considered as a private portion of DRAM. In order to speedup accesses to local memory, every SM has a L1 cache in addition to the GPU-wide L2 cache. Its main role is to reduce the performance penalty of register spilling.

4.3.3 Shared Memory

Every streaming multiprocessor contains a block of memory that is allocatable to thread blocks as a whole. It is called shared memory because it is accessible to all threads within a thread block. Shared memory is often used when different threads in the same thread block repeatedly access the same data or when threads need to share data. It is similar to the L1 cache, and is in fact located on the same physical SRAM.

The crucial difference however, is that it is a *programmable* cache. Clever use of shared memory can therefore be extremely useful to enhance performance. However, because of the parallel nature of thread execution in the GPU architecture, the programmer needs to be careful in dealing with data hazards. For instance, suppose thread A and B both store a value in shared memory and then want to read each other’s result. If no synchronization mechanisms are in place, the order in which thread A and B execute is arbitrary and thus thread A might incorrectly read shared memory *before* thread B has written out its value. This leads to undefined behavior and incorrect results. The solution is to insert **synchronization barriers** that ensure that all the threads in a block synchronize at a certain stage before proceeding. All the memory operations in shared memory before the barrier are then guaranteed to be visible after the barrier. In CUDA, thread synchronization is done with a call to `__syncthreads()`. It is considered bad practice however, to have many synchronization barriers. Excessive synchronization is bad for performance because it decreases the amount of thread-level parallelism that the GPU can exploit.

In the next chapter it will be discussed how shared memory was applied to the hybrid FHD/MD model.

4.3.4 Constant Memory

Like local memory, constant memory is located on the same DRAM as global memory. However, as the name suggests, constant memory is used for storing variables that remain constant during kernel execution. They are statically declared because their size has to be known at compile-time.

The advantage of using constant memory is that it is cached by a specialized per-SM constant memory cache. This cache has the property that if all threads in a warp request the same piece of data from constant memory, it is broadcast to all threads with a single instruction. In the context of the FHD/MD hybrid model it is therefore especially useful to store constant physical parameters, such as the ones listed in Table 3.

Constant memory is another example of how programmers have explicit control over the GPU's caching strategy. By designating variables as constant, programmers can ensure that using them in kernels comes at a low cost.

4.3.5 Atomic Operations

When multiple threads read and write the same location in global memory, a race condition can occur. This is when the outcome of the operation depends on the order of execution, leading to unwanted results.

For example, consider the volume density field v_{occ} being updated by threads A and B which are responsible for two different particles. When they want to increment the occupied volume stored in the same Eulerian cell, they need to perform a read-modify-write operation. Because it takes multiple instructions to complete such operations, there is no guarantee that other threads will not interfere. Suppose thread A reads v_{occ} first and calculates a new value. Meanwhile, thread B reads v_{occ} *before* thread A has written its new value to global memory. After thread A saves its updated value, it is overwritten by the value computed by thread B. The contribution by thread A is then erroneously discarded.

This type of data hazard is similar to the situation described earlier for shared memory. However, in this case the race condition takes place in global memory. Since the CUDA execution model precludes block-level synchronization, synchronization barriers are not an option for resolving the race condition. Instead, the CUDA API provides a number of **atomic operations** to tackle this type of problems. These are operations that are guaranteed to take place uninterrupted. They prevent race conditions by temporarily taking exclusive control of a piece of data, preventing other threads from accessing or editing it. Since the Fermi architecture, atomic operations are handled by the L2 cache, greatly accelerating atomic operations compared to previous generations.

4.3.6 Address Coalescing

The CUDA execution model is formulated using independent CUDA threads. However, this appearance is somewhat misleading, as CUDA threads are actually tightly bound together as warps (Section 4.2.3). This has implications for conditional branching (Section 4.2.4) as well as optimal memory access patterns.

GPUs are optimized to rapidly process sequential data. Therefore, memory requests to global memory are serviced by large chunks of data rather than individual words. The chunk size can be 32 bytes or 128 bytes, depending on the cache configuration [5]. Since the threads in a warp issue memory requests simultaneously, the performance of global memory depends strongly on the access pattern within warps.

Consider the scenario where a warp issues 4-byte memory requests from positions randomly scattered over global memory. In this case, every thread's memory request is handled by a separate transaction. If the chunk size is 32 bytes, only $4/32 = 12.5\%$ of memory traffic consists of the requested data. The other 87.5% is wasted bandwidth. Moreover, the memory accesses happen sequentially, which means that the warp has to wait for all 32 transactions to complete before continuing execution.

If consecutive threads access consecutive locations in memory however, the **address coalescing unit** (see Figure 12) recognizes this and coalesces them together into chunked data requests. Only four 32-byte memory transactions are then made with a bandwidth efficiency of 100%.

The programmer should therefore always attempt to structure his or her kernels so that sequential threads in a warp access sequential data items. Doing so will increase bandwidth utilization and reduce time spent on loading global memory. This is a direct consequence of the SIMD nature of GPU execution, and is another example of how writing GPU kernels for high performance requires technical knowledge of the underlying hardware.

V Design and Implementation of the GPU-Accelerated Simulation

Although performance is by definition the primary objective in high-performance computing, it is not the only aspect to take into consideration when writing writing scientific software. Other important factors include memory efficiency, scalability and modularity. These characteristics greatly extend the flexibility and applicability of high-performance applications. The first section below is dedicated to defining the design objectives of this work, as well as the design principles that help achieve those objectives.

The hybrid FHD/MD model consists of many different components working together to simulate the dynamic state of particles in a solvent. As a result, there were many different numerical techniques used in the implementation of the hybrid simulation. Fortunately, their GPU implementation can be described using a few key concepts. Section 5.2 discusses a scalable method to perform element-wise operations on the GPU. This is a general design strategy that applies to all parts of the model.

Section 5.3 on the other hand discusses the general strategy employed to deal with grid operations that cannot be formulated as simple element-wise operations. This section is especially relevant to the FHD model. The section after that deals with the most important algorithm for the particle-based model, which is the cell lists method for generating the neighbor list algorithm.

In Section 5.5, the GPU-accelerated interpolation from Lagrangian to Eulerian grids is presented. This operation is crucial to implementing the efficient calculation of occupied volume density fields, as described in Section 3.3. Since the coupling scheme is the novel component of the hybrid FHD/MD model, its GPU implementation is the most significant achievement of this work.

Lastly, the use of GPU-accelerated libraries is discussed in Section 5.6 for performing the fast Fourier transform (Section 3.4), as well as the generation of random numbers to sample stochastic variables at high throughputs (Section 3.6).

5.1 Design Objectives and Principles

5.1.1 Performance

The first priority in this work was to maximize performance. There are many ways to define performance, but the most consistent and reliable metric is execution time [13], defined as the time that the computer takes to complete a task from start to end. Smaller execution times mean higher performance and thus the performance of a program or subroutine is defined to be the reciprocal of the execution time:

$$\text{performance} = \frac{1}{\text{execution time}} \tag{V.1}$$

To compare the performance of two different programs that perform the same computation, the ratio of their execution times is used to quantify the speedup of one implementation over another. For example, X is n times faster than Y means that:

$$n = \frac{\text{execution time}_{\text{Y}}}{\text{execution time}_{\text{X}}} \tag{V.2}$$

With the definition of performance as the reciprocal of execution time in (V.1), this means that "X is n times fast than Y" is equivalent to saying "the performance of X is n times the performance of Y":

$$n = \frac{\text{execution time}_{\text{Y}}}{\text{execution time}_{\text{X}}} = \frac{\frac{1}{\text{performance}_{\text{Y}}}}{\frac{1}{\text{performance}_{\text{X}}}} = \frac{\text{performance}_{\text{X}}}{\text{performance}_{\text{Y}}} \tag{V.3}$$

Measuring Performance

Performance is thus measured in execution time. In this work I have measured the runtime of CPU subroutines using the high-resolution clock in the Boost.Chrono library [3]. This library allows the user to record certain time points from a clock and then calculate the time elapsed between different points. The execution time of a subroutine was thus measured by recording a time point directly before calling the subroutine and directly after the subroutine has completed.

On the other hand, measuring GPU execution time is less straightforward. CUDA kernel calls are asynchronous, which means that invoking a kernel returns to the caller immediately, without any guarantees with regards to the execution status of the kernel. Instead, the kernel is added to a queue, also called a **CUDA stream**. The asynchronous nature of kernel calls is designed to allow CPU execution to continue unhindered simultaneously with GPU workloads. However, this also means that CPU timers are not appropriate tools for measuring GPU performance.

Fortunately, the CUDA API provides **CUDA events** to time GPU applications. A CUDA event is essentially a time stamp that can be insert into the CUDA stream. Since the GPU itself records the moment that the event occurs, CUDA events are provide most direct method to measure execution time on the GPU.

Arithmetic Intensity

Writing efficient programs is an incredibly multifaceted and complex issue. However, there are a few important high-level principles when it comes to writing efficient GPU programs. One of them is maximizing **arithmetic intensity**. Informally, it is defined as:

$$\text{arithmetic intensity} = \frac{\text{computation}}{\text{memory}}$$

“Computation” refers to the number of floating-point operations executed and “memory” stands for the time spent on memory operations.

As mentioned in Chapter IV, GPUs possess massive arrays of computational units. However, this potential goes to waste when streaming multiprocessors are starved of data to operate on. Therefore, the programmer should aim to maximize the amount of computation relative to the time spent on memory.

This does not mean that memory *operations* as such should be minimized. Instead, it means taking advantage of the CUDA memory model described in Section 4.3 to prefer the use of fast memory and cache structures—like shared and constant memory—over slower memory where possible.

5.1.2 Memory Efficiency

In contrast to CPUs, GPUs are prepackaged with their own dedicated graphics memory. This makes it impossible to expand on-board GPU memory. Moreover, GPUs do not support advanced memory techniques such as memory paging, to manage more memory than what is available on the graphics card. As a result, GPU memory is considered to be a scarce resource.

Therefore, the algorithms were designed to run without imposing a heavy burden on the GPU memory. For example, intermediate results are stored in designated buffers that are constantly reused to keep allocated memory to a minimum.

5.1.3 Scalability

System sizes in molecular dynamics vary from a few thousand particles to millions of particles and more. Moreover, there is a rich variety of size-dependent phenomena that occur at the nanoscale. As a result, different types of simulations may require wildly different grid sizes and spatial resolutions. It is therefore important that the hybrid simulation is able to a handle a large variety of problem sizes. Moreover, the performance should not deteriorate for larger workloads. This property is known as scalability, and it was an important factor in the design of the hybrid simulation.

5.1.4 Modularity

The hybrid modeling framework is an area of active research. Extensive changes to the model are expected in the near-future. Accordingly, the source code was written with an

Listing 3: Daxpy kernel with grid-stride loop

```
__global__
void grid_stride_daxpy(int n, double a, double *X, double *Y)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < n;
         i = i + blockDim.x * gridDim.x)
    {
    Y[i] = a * X[i] + Y[i];
    }
}
```

emphasis on modularity. In particular, C++ language features such as object-oriented programming and templating were employed to facilitate future modifications.

5.2 The Grid-Stride Loop

Consider the kernel (Listing 2) first introduced in Section 4.2.2:

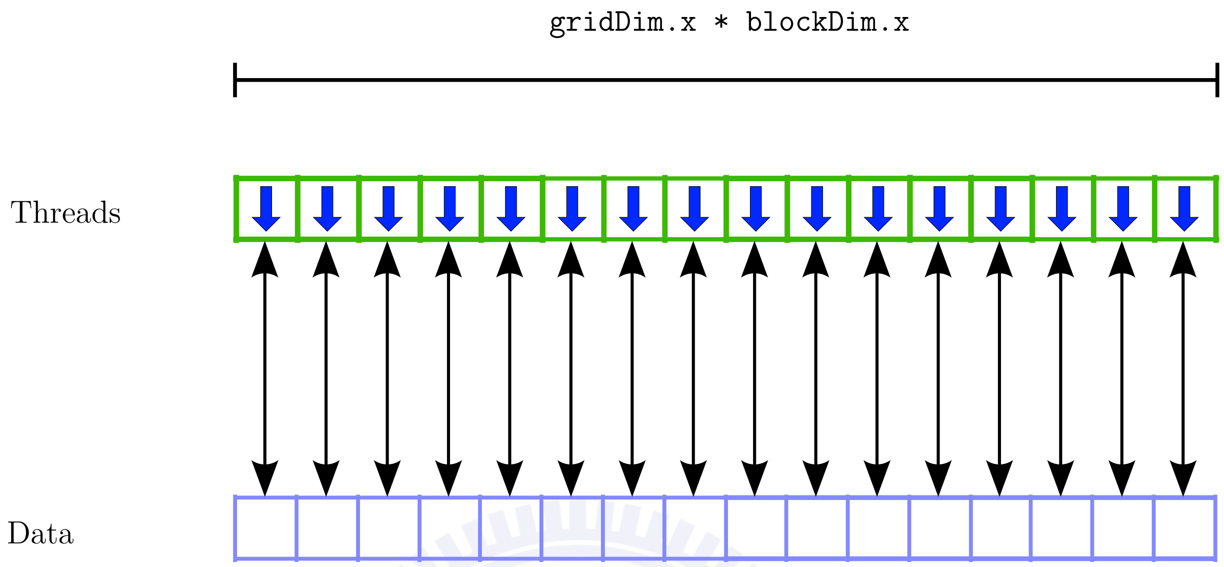
Listing 2: Daxpy kernel in CUDA C/C++

```
__global__
void daxpy(int n, double a, double *X, double *Y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) Y[i] = a * X[i] + Y[i];
}
```

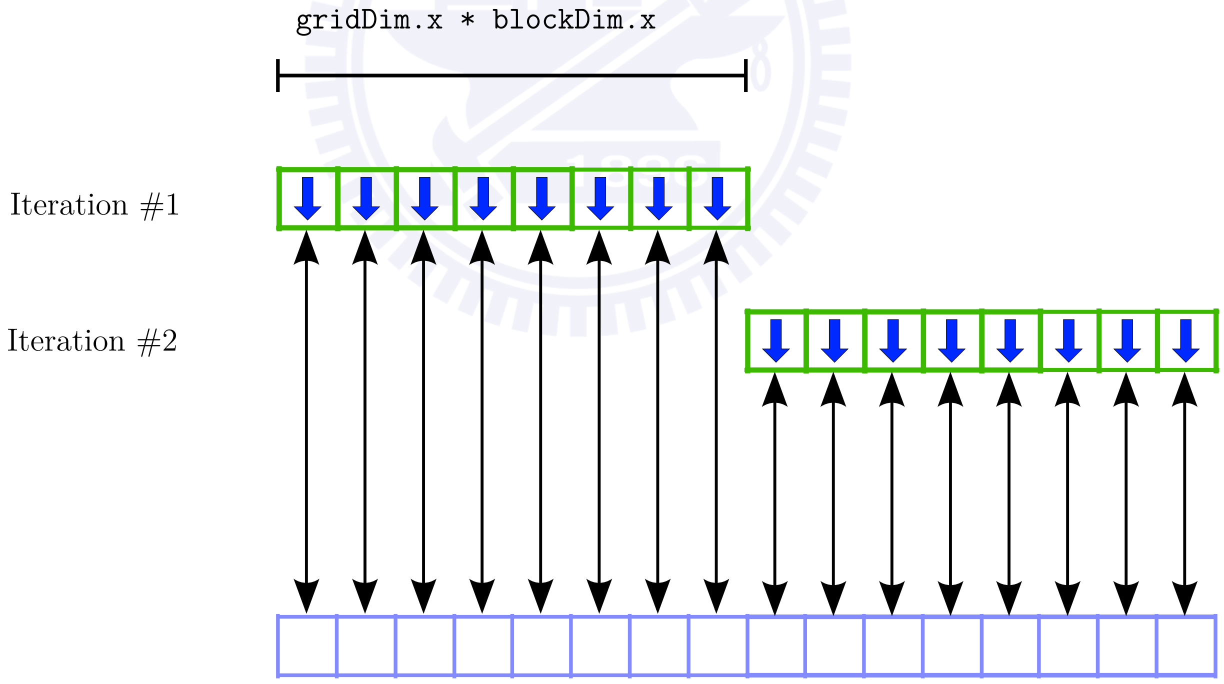
The goal was to parallelize the vector operation $\mathbf{Y} = \mathbf{Y} + a\mathbf{X}$. This kernel eliminates the loop that a CPU function would use and instead invokes a thread for every element of $\mathbf{X}$. The mapping of threads to elements processed thus has a one-to-one correspondence (Figure 16a). Because it requires creating one large grid to process all elements in a single pass, this can be considered a "monolithic kernel".

A more scalable kernel on the other hand, is made by implementing a **grid-stride loop** [11], as demonstrated in Listing 3. This is a loop that proceeds by taking steps of size `blockDim.x * gridDim.x`. This is the number of total threads in the grid, hence the name *grid-stride* loop. The loop condition ensures that the loop terminates when it is out of bounds and is similar to the if condition in the monolithic kernel. The mapping of threads to vector elements is visualized in Figure 16b.

There are several advantages to using a grid-stride loop. First of all, the grid-stride loop ensures that the kernel works for any size `n`. This is not always the case for the regular kernel because it is limited by maximum grid size. This kernel eliminates those



(a) Monolithic kernels have a one-to-one correspondence between threads and data



(b) Grid-stride kernels loop over the data

Figure 16: Monolithic versus grid-stride kernel

Listing 4: Template for element-wise kernel

```
template <class FunctionObjectType>
__global__
void element_wise_kernel(int n, FunctionObjectType fun_obj)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < n;
         i = i + blockDim.x * gridDim.x)
    {
    fun_obj(i);
    }
}
```

limitations and is therefore **scalable**.

Additionally, the creation and destruction of threads comes with some overhead. Reusing threads by processing multiple elements with each thread amortizes the cost of thread creation. Moreover, since the scalable kernel works correctly for any grid configuration, the number of blocks can be independently tuned to achieve the fastest runtime. The grid-stride loop therefore enhances **performance**.

It turns out that the grid-stride loop is thus a very useful kernel design pattern. Therefore, I have used it extensively for calculations that had a simple element-wise correspondence between data and threads. However, implementing a grid-stride kernel for every such operation would be quite bothersome. Instead, templating and object-oriented programming were used to write a single kernel that could support any arbitrary operation (Listing 4). The structure is identical to Listing 3, but this template kernel accepts a *function object* instead. This is an object that contains the data that would ordinarily be passed as arguments and has a member function `operator()` that executes a single iteration of the loop body, such as line 8 in Listing 3. It accepts a single argument `i` because the loop body needs to know which data to access.

The reason that Listing 4 is written as a template is so that it can be compiled for any function object type. During compilation, the compiler will instantiate a version of the template by inserting the appropriate function object type.

For example, the function object for daxpy is given in Listing 5. The member variables contain the data that corresponds to the arguments in Listing 3. The constructor initializes them to user-supplied arguments. Finally, `operator()` corresponds to the loop body that the function object expresses. Here, it is the same as line 8 in Listing 3. The qualifier `__device__` indicates that the function resides in the GPU.

Defining `num_blocks` and `num_threads` as the optimal grid and block dimensions respectively, the invocation of Listing 3 would be:

```
grid_stride_daxpy <<< num_blocks, num_threads >>> (n, a, Y, X);
```

Listing 5: Declaration of daxpy function object type

```
struct DaxpyFunction
{
        //member variables
        double a;
        double *X, *Y;

        //constructor
        DaxpyFunction(double _a, double *_X, double *_Y)
        : a(_a), X(_X), Y(_Y) {}

        //member function
        __device__
        operator() (int i)
        {
            Y[i] = a * X[i] + Y[i];
        }
}
```

Using the templating approach on the other hand:

```
DaxpyFunction fun_obj(a, X, Y);
element_wise_kernel <<< num_blocks, num_threads >>> (n, fun_obj);
```

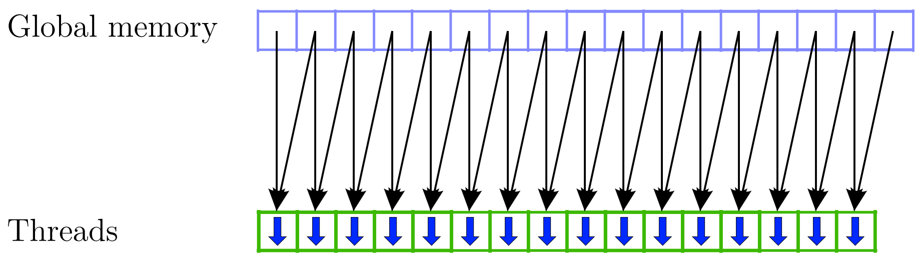
When the compiler encounters these lines it will generate a kernel with `DaxpyFunction` as the function object type. The same applies to any simple element-wise operation expressed as a function object and used with the same template kernel. This shows how templates can be used to construct kernels in a flexible manner.

5.3 Grid Tiling

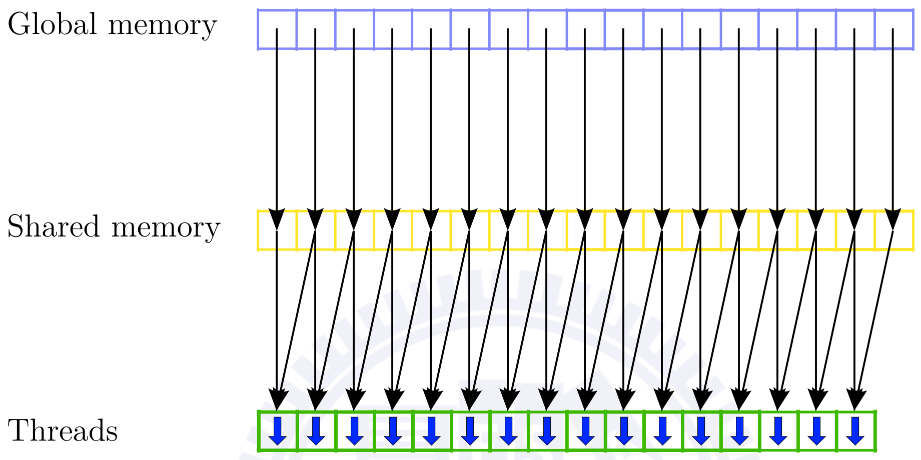
Section 3.2 introduced the staggered grid that puts scalar values at cell centers and vector components at cell faces. Vector fields are conceptually decomposed into three scalar grids that are shifted from the scalar grid so that their cell centers coincide with the cell faces of the unshifted grid. This decomposition is not only helpful theoretically however, it is also efficient from a computational standpoint to store a vector field as separate scalar grids in global memory.

The reason for this is that operations like the gradient and divergence can then be formulated as computations that have a straightforward correspondence between vector component grids and scalar grids. This improves the regularity of memory access patterns and is therefore better suited for the GPU.

The formulations of finite differences for the staggered grid were also discussed in Section 3.2. Essentially, both the gradient and divergence operations express central



(a) Data flow for naive forward sum



(b) Data flow for forward sum with shared memory

Figure 17: Naive forward sum versus forward sum using shared memory

finite differences as weighted forward and backward sums respectively. Forward sum here means that the computation of cell (i, j, k) depends on the current cell and the cell one further along the x-, y- or z-axis, i.e. at $(i+1, j, k)$, $(i, j+1, k)$ or $(i, j, k+1)$, and vice versa for the backward sum.

Furthermore, the decomposition of vector fields means that the forward or backward sum has to be considered only one direction at a time. The finite difference kernels thus need to compute forward or backward sums in one direction at high throughput.

Consider the x-derivative of the gradient operator, with finite difference formula is given by (III.14a). The straightforward GPU approach is to assign a thread to each cell and then compute its forward sum. A naive implementation of the forward sum loads the current cell's value and the forward cell's value from global memory (Figure 17a). The important observation here is that every data element in global memory is accessed twice by two different threads. This is a wasteful memory access pattern because multiple, expensive global memory reads are performed on the same data.

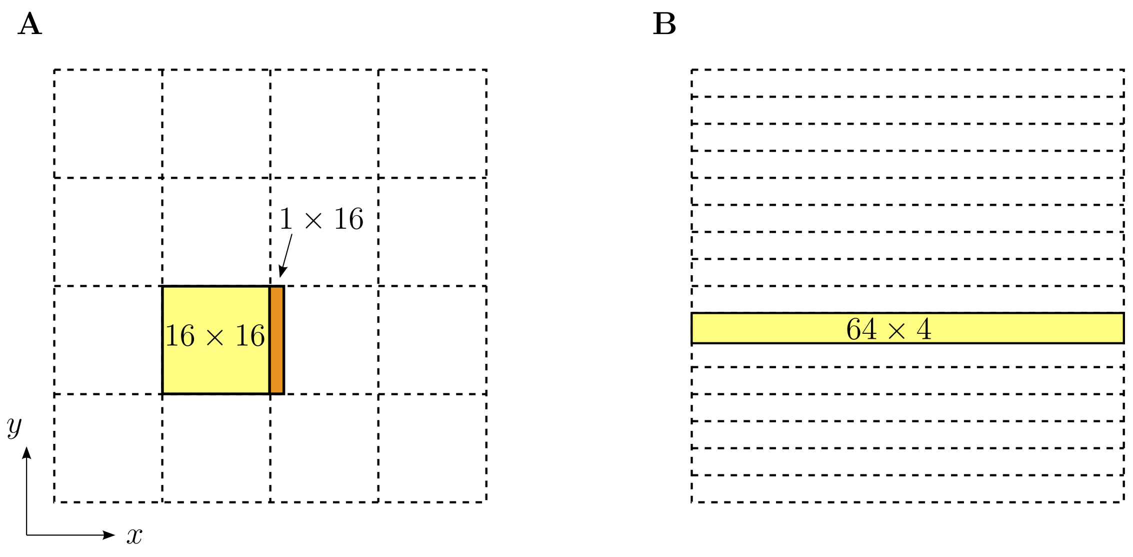


Figure 18: Tiling strategies for x-derivative

5.3.1 Tiling Strategy

The repeated global memory operations can be avoided by a **tiling** approach. This involves loading a portion of the data, or a tile, into shared memory, and then reusing data from the tile. The additional shared memory operations are offset by reducing the number of global memory operations, which are much slower. The memory access pattern then roughly looks like Figure 17b.

Shared memory is a limited resource that is local to a thread block. So the efficiency of this approach depends for a great deal on the tiling strategy. Figure 18 shows two different tiling strategies for a 64^3 grid. Both use 256 threads per thread block, but the tile aspect ratio is very different. Strategy A splits up the xy-plane into a regular grid of 16×16 squares. However, as the figure shows, the shared memory tile actually has dimensions 17×16. This is because the threads at the right edge of the tile need an additional column of data to compute their forward sums. This extra column is called a "halo" and introduces a few inefficiencies. First of all, the elements in the halo are once again loaded twice from global memory. Furthermore, the global memory loads are not coalesced because loading 17 consecutive elements from global memory uses up only a portion of the potential bandwidth compared to loading 32 consecutive elements.

Tiling strategy B on the hand splits the domain into a series of rectangular 64×4 stripes. The tile contains all the points in the x-direction and thus avoids any overlap with other tiles. Therefore no data element is requested twice from global memory. Moreover, the tile is filled using only perfectly coalesced memory accesses. This strategy is thus clearly superior to strategy A.

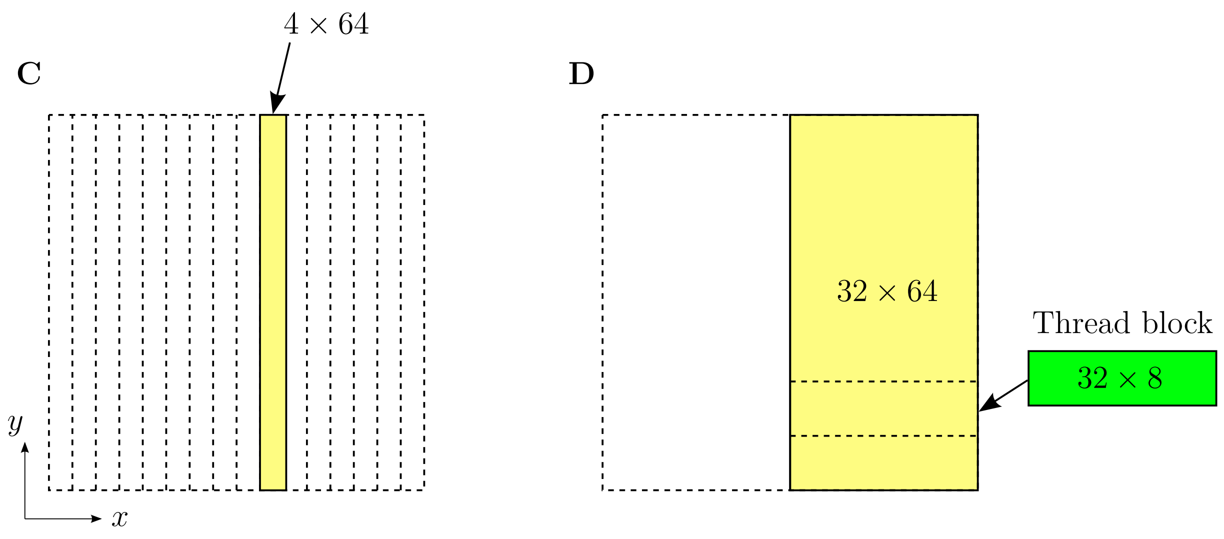


Figure 19: Tiling strategies for y-derivative

5.3.2 The Block-Stride Loop

For the y- and z-derivatives, a similar tiling approach is attempted as tiling strategy B (Figure 19). Unfortunately, because there are only four consecutive data elements per row (x is the innermost dimension), the memory accesses are no longer coalesced. Instead, every warp load only uses 12.5% of the total memory bandwidth.

To regain the efficiency of coalesced memory loads, the width of the tile is increased to 32 (strategy D). Every warp load then once again accesses 32 consecutive elements in global memory. However, a one-to-one mapping of threads to tile elements would require a thread block of $4 \times 64 = 2048$ threads, which is higher than the maximum block size of 1024.

This is resolved by mapping threads to multiple tile elements. The kernel then contains a loop that processes the tile one block-sized portion at a time. Since the loop jumps by the block size, it is a **block-stride** loop. Similarly to how the grid-stride loop decouples the grid size from the problem size, the block-stride loop decouples the block size from the tile size. Tile sizes and block size become independent execution parameters that can be tuned for maximal performance.

5.3.3 The Tile-Stride Loop

The tile size is still explicitly linked to the problem size however. As the dimensions of the box are increased, the tile size has to be adjusted accordingly. Arbitrarily large box sizes require arbitrarily large tiles that do not fit into shared memory at all. For intermediate box sizes, like a $256 \times 256 \times 256$ box, the resulting tiles may still fit into the shared memory, but it limits the number of thread blocks that a streaming multiprocessor can

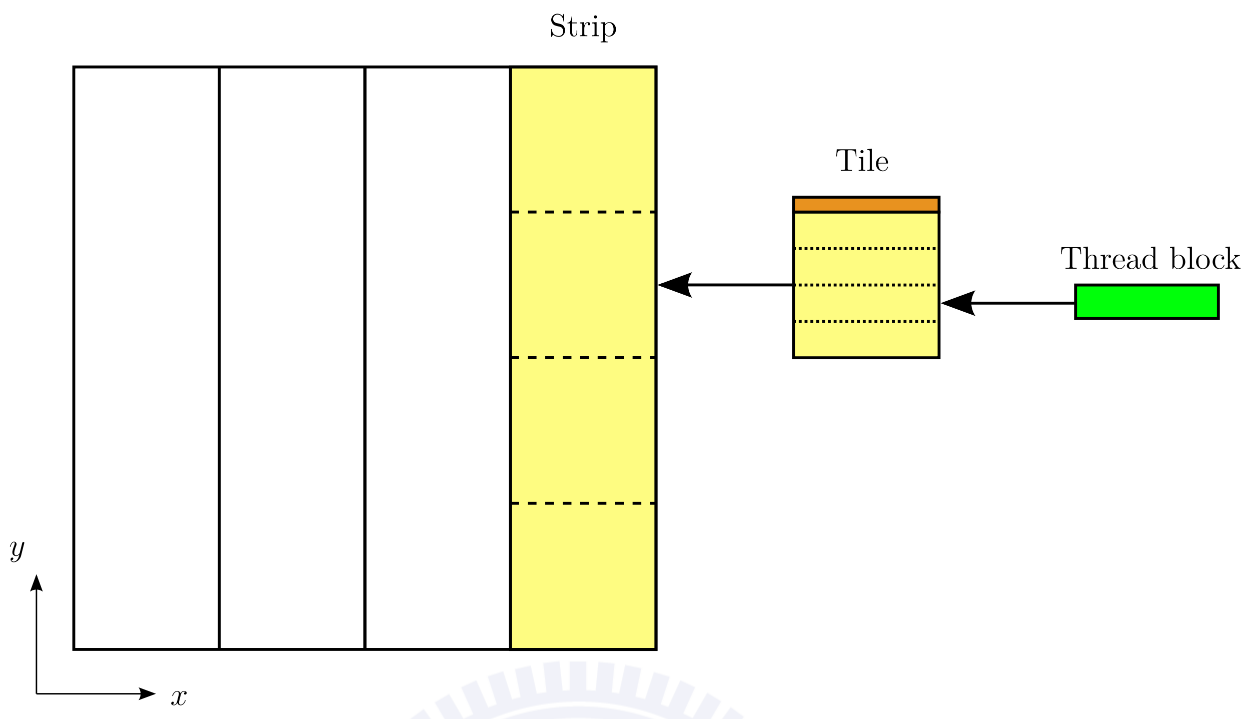


Figure 20: Tile-stride loop

process simultaneously. This results in less active threads per SM and severely undermines performance.

Therefore, I implemented a **tile-stride loop** to decouple the tile size from the problem size (Figure 20). In analogy with the grid-stride and block-stride loop, the tile-stride loop emulates a conceptually larger tile, or "stripe", using smaller tiles that loop over the stripe. In order to preserve the memory access properties of the original tiling strategy, the tiles for the forward sum iterate in *backward* steps. The reason for this is that the bottommost row (for the y-derivative) stored in shared memory can then serve as the halo for next tile. Only the very first tile iteration requires an extra memory access that is repeated because it cannot take it from the previous iteration. Fortunately, this is a small price to pay for a completely scalable kernel that accesses the grid values only once elsewhere.

5.3.4 Summary

The grid-stride loop was also employed so that the number of blocks was not dependent on the problem size. There are therefore a total of three within the kernel. The outermost loop ensures that the thread blocks iterate over all stripes. The tile-stride loop then splits the strip into tiles that fit into shared memory. Finally, the innermost loop processes the tile in block-sized portions. Altogether, this looped structure enables the user to independently tune grid, block and tile size for maximal performance.

The gradient and divergence operators make use of forward and backward sum kernels respectively. The multiplication of a scalar and vector fields, also introduced in Section 3.2.5, is calculated using forward sums as well and thus accesses the data in the same way as the gradient operator. These subroutines that have the same data access pattern but perform different sums are implemented by function objects, analogously to Section 5.2.

5.4 Neighbor List Generation

The GPU-accelerated neighbor list generation was implemented based largely on the cell lists algorithm (Section 3.5) of the GPU molecular dynamics software HOOMD-blue [2, 9]. A brief high-level description of the procedure will be given here. More details can be found in the cited papers.

5.4.1 Construction of Cell Lists

The lists were implemented on the GPU using two arrays. The first array `celln` is of size `Nc`, the total number of particle grid cells, and records the number of particles stored in each cell list. The second array `celldata` holds a list of particles for each cell. It contains `Nc * Np_max` elements, where `Np_max` is the maximum number of particles per cell list. The data in `celldata` is arranged as `Nc` blocks of `Np_max`-sized cell lists. For example, suppose the maximum cell list size is equal to `5` and the cell with global index `3` contains 2 particles with indices `5` and `17`. The particle count is stored as `celln[3]` = 2, while the particle indices are stored in the cell list list starting at `celldata[3 * 5 = 15]`. Therefore, `celldata[15] = 5` and `celldata[16] = 17`.

To construct the cell lists data structures, `celln` is first initialized to zero because all cell lists are empty at first. Then, one thread per particle is launched on the GPU. Every thread loads its position from global memory and calculates its cell coordinates using equation (III.38). The global index `gid` is then computed through (III.9). Next, the counter at `celln[gid]` is **atomically** incremented. The atomic operation to `celln[gid]` returns the value that was stored there *prior* to the operation. Using this value as the cell list index ensures that particles belonging to the same cell do not overwrite each other's place in the cell list. For example the first thread to atomically increment a particular cell's counter sees `0`, while the next thread sees `1` because of the addition by the first thread.

For the example given above, the thread responsible for particle `5` was the first to atomically increment `celln[3]`. The returned value was `0` and therefore the thread knew that it was the first particle to be added to the cell list. Accordingly, particle index `5` was saved to `celldata[3 * 5 + 0 = 15]`. Next, the thread assigned to particle `17` read the value 1 from celln[3] because it had already been incremented by particle 5. Therefore, the thread wrote its particle index to celldata[3 * 5 + 1 = 16].

5.4.2 Construction of Neighbor Lists

CPU implementations usually maintain a list containing only unique neighboring pairs of particles. For example, neighboring particles i and j will be saved as (i, j), but the converse pair (j, i) is not stored. This is because CPU algorithms take advantage of Newton's third law to iterate over i, calculate the pairwise force once and add it to the forces of particles i *and* j. However, on the GPU this would mean that every pairwise force calculation performs an atomic read-modify-write on a scattered location j, which is a very inefficient memory operation.

Therefore, the threads are assigned to calculate forces for a single particle only. Accordingly, the neighbor list should store neighbor pairs for both of the particles involved so that every particle has full knowledge of its neighbors. The data structures used are completely analogous to the cell lists: neighborn is an array of size Np that stores the number of neighbors for every particle, and array neighbordata stores Np blocks of size neighbor_max to store a neighbor list of maximum neighbor_max neighbors per particle.

Similarly to the cell lists, the construction of the neighbor lists starts by initializing the neighbor counters neighborn to zero. Every particle is assigned a single thread that first calculates the cell that it belongs to (see previous subsection) and then, using the data structures celln and celldata, iterates over the particles in the same cell and the neighboring cells (Section 3.5 and Figure 9). The thread checks every potential neighbor by testing whether the interparticle distance is less than R_{cutoff}. Confirmed neighbors are finally added to neighbor list neighbordata using atomic increments of neighborn in analogy with the cell lists.

5.4.3 List Overflow

One important issue that the two previous subsections glossed over is what happens when the number of particles in a cell list exceeds Np_max, or when the number of neighbors in a neighbor list exceeds neighbor_max. In that case, the block of data allocated to each list in celldata or neighbordata is not enough to hold the largest list. In other words, there is a **list overflow**.

In order to detect an overflow, the variables global_Np_max and global_neighbor_max are introduced. Before generating lists, these global variables are initialized to Np_max and neighbor_max respectively. Whenever a thread experiences overflow, it writes the overpopulated list size to the global maximum using an atomicMax operation. This is an atomic operation that writes the list size to the global maximum only if it is larger than the current global maximum.

After constructing a list for the first time, its global variable is probed to check whether `global_Np_max > Np_max` or `global_neighbor_max > neighbor_max`. If an overflow has occurred, the list size is adjusted accordingly by setting `Np_max = global_Np_max` or `neighbor_max = global_neighbor_max` and reallocating the array `celldata` or `neighbordata` with the updated maximum list size. The counter data structure `celln` or `neighborn` is reinitialized to zero and the list construction is repeated. After generating the list for the second time, the global variable does not have to be checked again because the list size has been set to the maximum size encountered in the previous attempt.

5.5 The Interpolation Scheme from Lagrangian to Eulerian Grids

This section describes the implementation of the occupied volume interpolation from the Lagrangian grid to the Eulerian grid described in Section 3.3. For simplicity, it is assumed that particles all have the same coupling radius R_{coupling}. The method can easily be extended to multiple particle types, but it is not discussed in this work.

5.5.1 Original Implementation

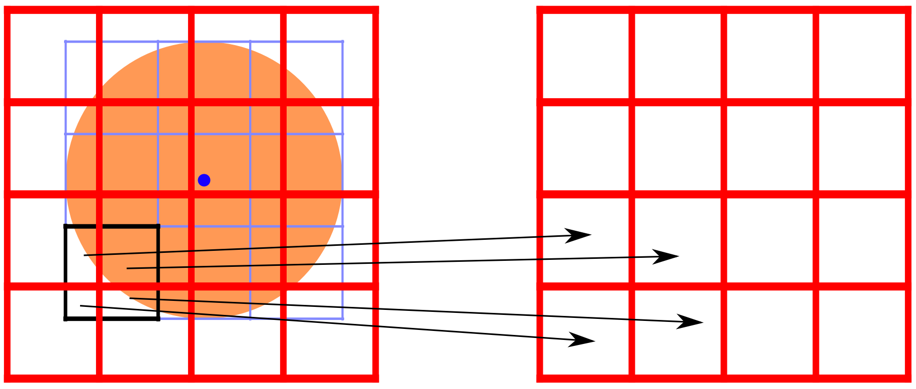
Conceptually, each particle i is associated with a discrete Lagrangian occupied volume field $V^i_{i,j,k}$. Since it is assumed that all particles have the same coupling radius and have the same fixed position within their Lagrangian grid, this field is identical for all particles. Moreover, most of the cells have zero occupied volume because the particle fills only a small portion of the entire spatial domain. This sparse data structure is therefore stored using a **compact representation**. The occupied volume of all *nonzero* Lagrangian cells is stored in `V_occ_lag`, and the position relative to the Lagrangian particle position is stored in `R_relative`. Each cell's position in the Eulerian grid can then be calculated by adding the relative position vector to the Eulerian position vector of the particle.

The original subroutine in FORTRAN 90 (Listing 6) is a straightforward implementation of the serial procedure outlined in 3.3. It consists of a nested loop that loops over all particles and all nonzero Lagrangian cells. Since it is assumed that all particles have the same coupling radius and position within the Lagrangian grid, the same compact data structure is used throughout the subroutine. Figure 21a shows the data pattern for an individual Lagrangian cell. This pattern repeats for all nonzero Lagrangian cells, as illustrated in Figure 21b. Every particle interpolates its occupied volume according to this data flow, but to different memory locations in `V_occ` depending on their position in the Eulerian grid.

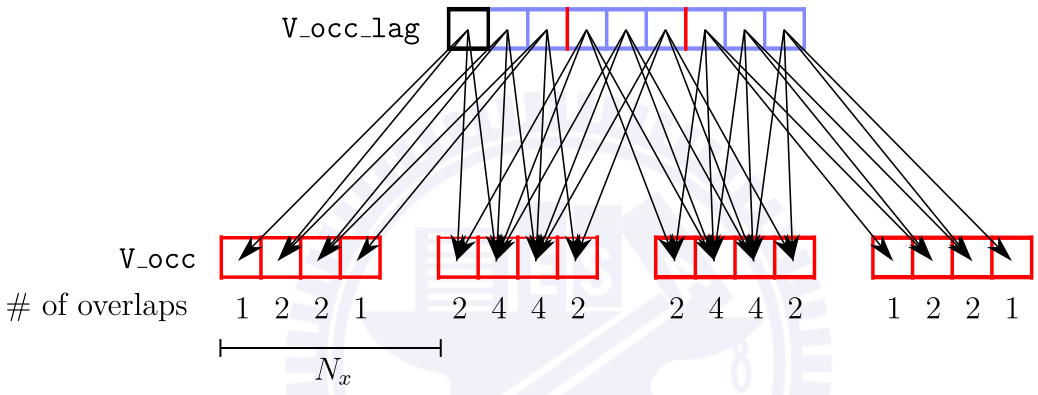
Figure 21b exposes an important consequence of this procedure: most Eulerian cells are repeatedly written to. In fact, every arrow represents a read-modify-write operation

Listing 6: Original implementation of occupied volume interpolation

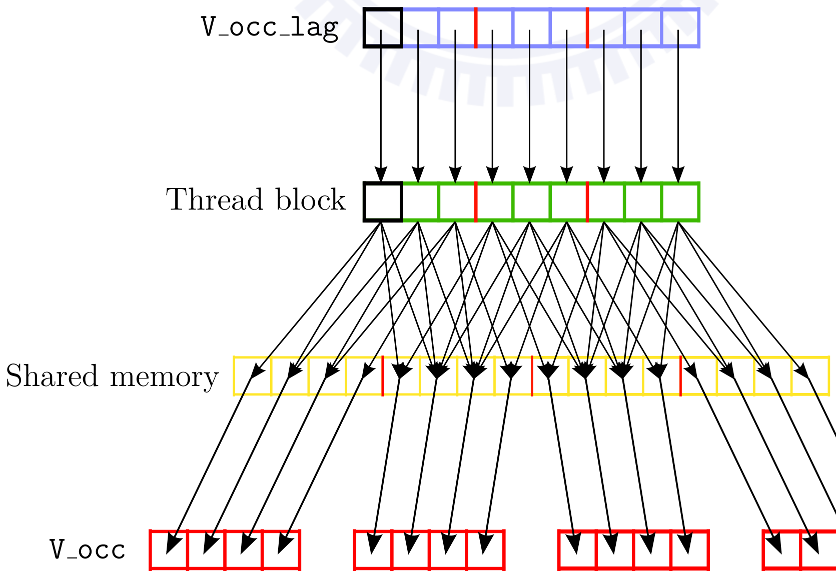
```fortran
SUBROUTINE GET_OCCUPIED_VOLUME(R)
! Particle position vectors are given by R(3, Np)

! Initialize occupied volume field to zero
V_occ = 0

! Loop over particles
DO i = 1, Np

    ! Loop over Lagrangian cells
    DO j = 1, num_lagrangian_cells

        ! Add relative position to particle position
        ! R_relative(:, j) contains the relative position
        ! of the j-th Lagrangian cell
        R_cell(:) = R_relative(:, j) + R(:, i)

        ! Calculate overlap of Lagrangian cell with Eulerian cell
        ! V_overlap is the calculated overlapping volume
        ! idx_overlap is the corresponding Eulerian cell index
        CALL GET_OVERLAP_CELLS(R_cell, idx_overlap, V_overlap)

        ! Loop over overlapping Eulerian cells
        DO k = 1, 8
            ! Eulerian index
            eul_idx = idx_overlap(k)
            ! Overlapping volume fraction
            fraction = V_overlap(k) / Vc
            ! Add contribution to occupied volume field
            ! V_occ_lag(j) contains the occupied volume
            ! of the j-th Lagrangian cell
            V_occ(eul_idx) = V_occ(eul_idx) &
            + fraction * V_occ_lag(j)
        ENDDO
    ENDDO
ENDDO
END SUBROUTINE GET_OCCUPIED_VOLUME
```



(a) Interpolation from Lagrangian to Eulerian grid



(b) CPU data flow



(c) GPU data flow

Figure 21: Lagrangian to Eulerian grid interpolation

because the interpolated occupied volume value is added to the Eulerian value that was already stored there. The Eulerian cells within the particle are modified four times in total (eight times in three dimensions). Repeated modifications are not necessarily detrimental for performance, as long as the program displays a high locality of reference, because then the L1 cache provides fast access to the Eulerian cells. However, since the Lagrangian cells are processed one by one, the program alternates between two locations in memory that are relatively far away because consecutive cells in the y-direction are N_x cells apart in memory. In three dimensions, the spatial locality is even worse because in that case eight cells need to be accessed in four scattered memory locations.

5.5.2 GPU Implementation

Figure 21a shows a particle and only its nonzero Lagrangian cells, leaving out the rest of the Lagrangian grid. As mentioned in the beginning of this section, the fact that the particle occupies just a localized portion of the grid is explicitly taken advantage of to compress the Lagrangian grid. The key observation in the GPU implementation is that the collection of Eulerian cells overlapping the nonzero Lagrangian cells *also* constitute a small, confined space within the entire Eulerian grid. In fact, Figure 21a shows only those Eulerian cells, even though the Eulerian grid generally extends much further.

The main idea in the GPU implementation is to take advantage of this fact by interpolating first to a *compact representation of the Eulerian cells* in shared memory and only then scattering the intermediate data to the full Eulerian grid in global memory. This method performs the interpolation in fast shared memory and avoids repeated accesses to global memory. The data flow of the GPU implementation is visualized in Figure 21c. Because multiple particles can overlap the same Eulerian cells, the global memory modifications in the last step are performed using atomic operations.

Even though the shared memory is allocated to Eulerian cells, the threads are initially **not** mapped to shared memory. Instead, they are mapped to Lagrangian cells. The reason is that every Lagrangian cell interpolates to exactly eight overlapping Eulerian cells. From the perspective of the compressed Eulerian grid however, each Eulerian cell overlaps with at least one Lagrangian cell and at most eight Lagrangian cells. If the threads were mapped to Eulerian cells, each thread would therefore need to perform between one and eight interpolations. This results in an imbalanced workload, and because of branch divergence (see 4.2.4), this means worse performance.

Another way to look at it is that the number of Lagrangian cells occupied by a particle is always less than the number of Eulerian cells that they overlap. Mapping threads to the Lagrangian cells thus requires less threads per particle, while taking the same time to interpolate as the Eulerian-mapped threads because of branch divergence. The interpolation steps are illustrated for two dimensions in Figure 22.

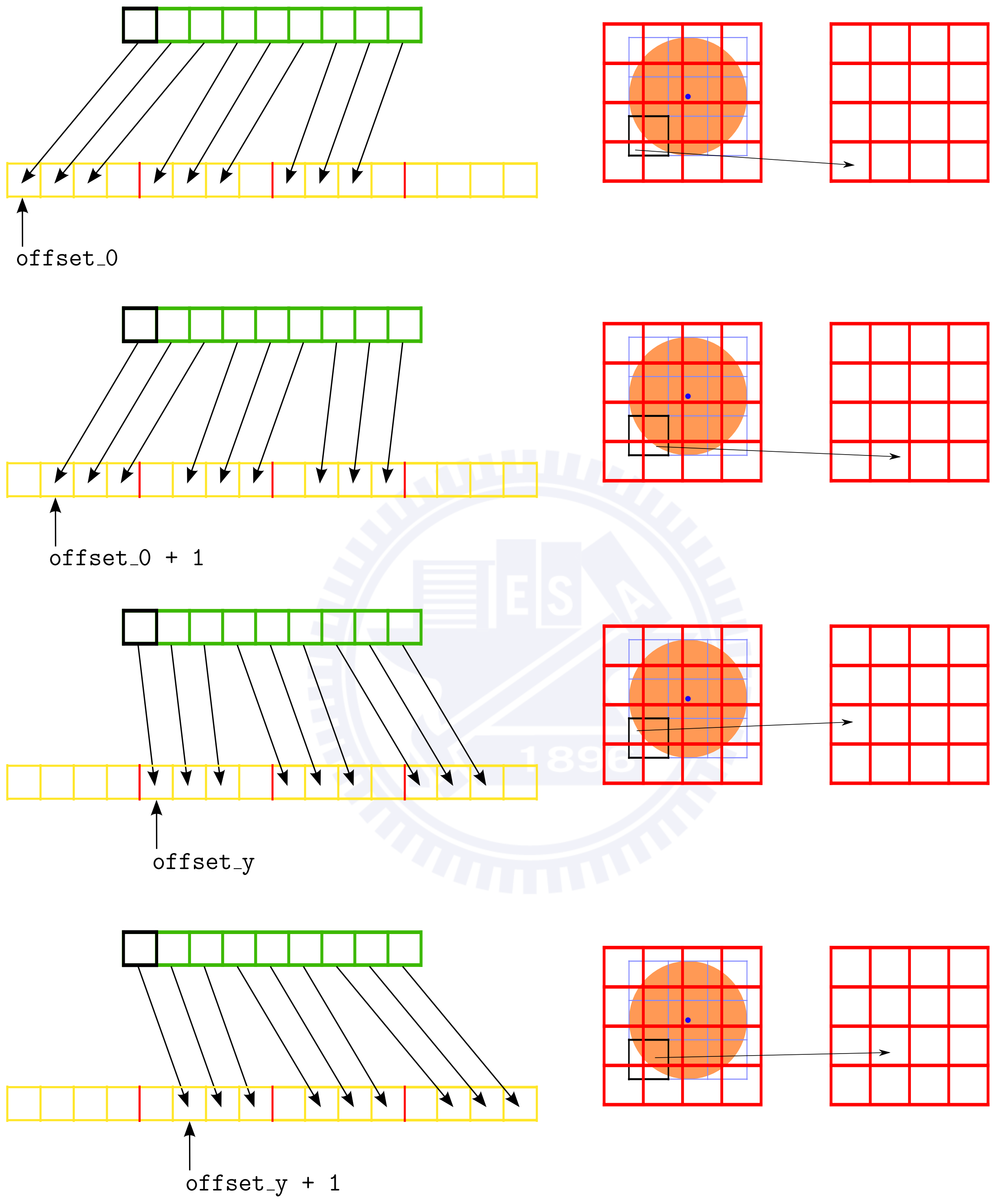


Figure 22: Lagrangian-mapped interpolation. The threads (green) interpolate their Lagrangian occupied volumes to shared memory (yellow). The corresponding cells are shown on the right from the Lagrangian grid (blue) to the Eulerian grid (red).

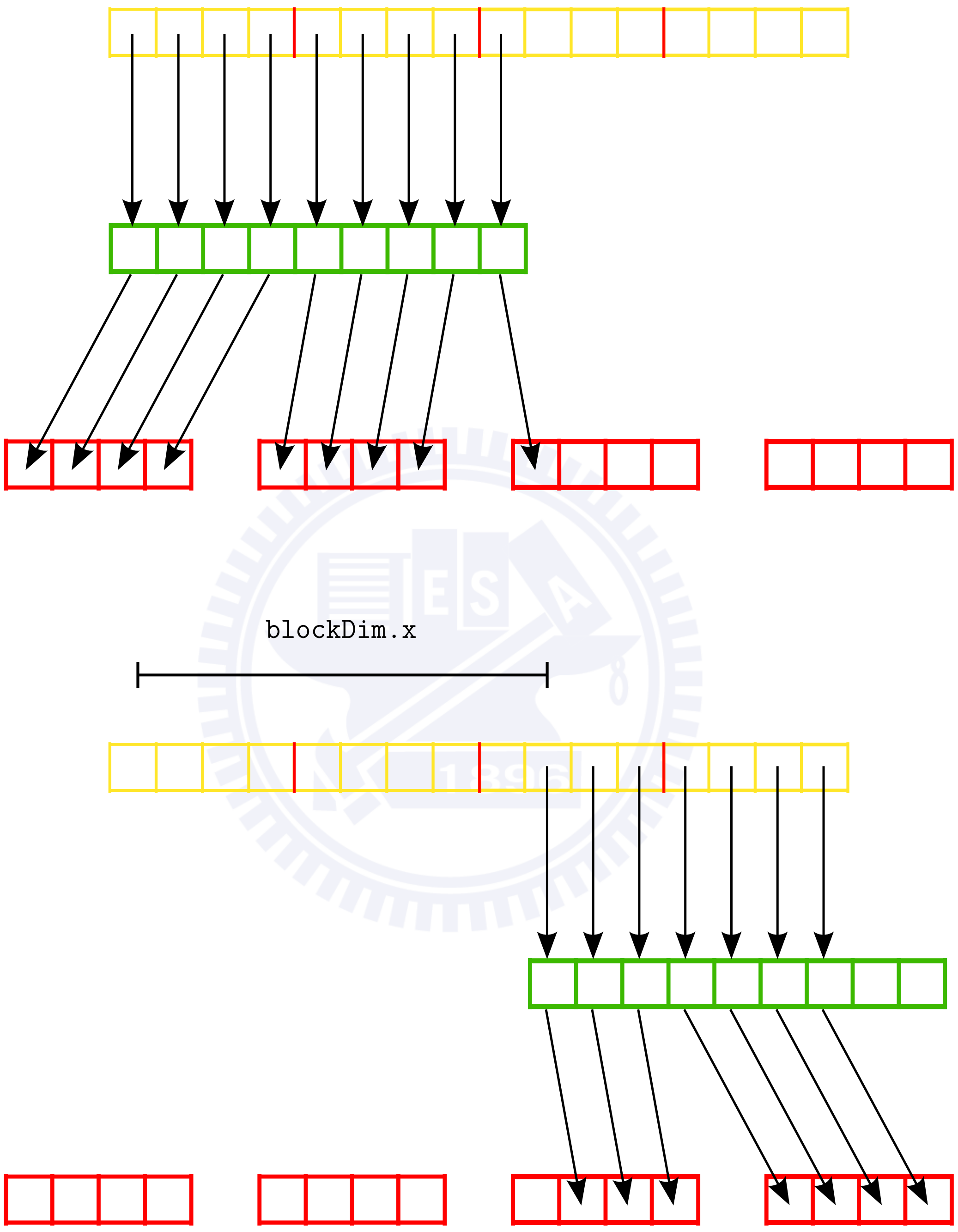


Figure 23: Eulerian-mapped write to global memory. The threads (green) store the interpolated Eulerian values in shared memory (yellow) to global memory (red) using a block-stride loop.

Once the interpolation is complete, the threads are remapped (Figure 23) in order to store the results in shared memory to global memory. In this stage the threads correspond to Eulerian cells instead of Lagrangian cells. Because of the initial Lagrangian mapping of threads however, there are more Eulerian cells than there are threads inside the thread block. A **block-stride loop** is therefore necessary to loop over the Eulerian cells.

The GPU interpolation algorithm thus does not have a constant thread mapping throughout its execution, but actually passes through a **two-stage mapping**. In the first stage, the threads are mapped to Lagrangian cells in order to achieve the highest density of computation during the interpolation scheme. The threads then switch to an Eulerian mapping in the second stage to transfer the data created in the first stage to global memory.

The main challenge that arises when implementing this algorithm is directing the flow from the compact Lagrangian representation to the compact Eulerian representation. Moreover, the example discussed so far explains how the inner loop of Listing 6 is unrolled by the GPU algorithm. In practice, the outer loop over particles is also unrolled by concatenating the Lagrangian-mapped threads of all particles to make up the computational grid. Inside a given thread block, the shared memory is then also partitioned into regions reserved for different particles. This further complicates the mapping of Lagrangian cells to Eulerian cells.

In fact, the amount of computation and communication required for a Lagrangian-mapped thread to figure out what location in shared memory to interpolate to is significant enough that it would undo any performance benefits that the present GPU algorithm offers. Therefore, instead of calculating the Lagrangian-to-Eulerian correspondence on the fly, I decided to compute it beforehand. Because the role of this data is only to facilitate fast execution and does not have any physical meaning by itself, it is referred to as **auxiliary data**. The threads then access this data during execution so that the interpolation scheme can proceed smoothly.

5.5.3 Auxiliary Data for Interpolation

There are two paths in the data flow that are guided by auxiliary data. The first is the interpolation from the Lagrangian cells to the Eulerian cells in shared memory. The second is in the second stage when the values of the compact Eulerian cells in shared memory are scattered to global memory. Accordingly, two types of auxiliary data were defined. The first type describes the correspondence between the compact Lagrangian representation and the compact Eulerian representation. The second type of data on the other hand is responsible for mapping the compact shared memory to the full Eulerian grid in global memory. Its role is therefore analogous to storing the relative positions of nonzero Lagrangian cells in the CPU implementation. The difference however, is that the "decompression" happens from a compact Eulerian array instead of Lagrangian.

Now consider a single thread in the first data path. Each Lagrangian cell interpolates to eight Eulerian cells[1]. Therefore, the thread writes to eight locations in shared memory. The Eulerian cells are laid out in shared memory as contiguous stripes along the x-direction (Figure 22). Therefore, the eight Eulerian cells appear as four groups of two adjacent cells. Each thread thus needs to know four locations in shared memory, with the other locations exactly one data element further.

The first type of auxiliary data therefore consists of four integer shared memory offsets per thread. They are denoted as:

1. `offset_0` corresponds to the Eulerian cell overlapping the $(0, 0, 0)$-corner[2].
2. `offset_y` corresponds to the Eulerian cell overlapping the $(0, d_y, 0)$-corner.
3. `offset_z` corresponds to the Eulerian cell overlapping the $(0, 0, d_z)$-corner.
4. `offset_yz` corresponds to the Eulerian cell overlapping the $(0, d_y, d_z)$-corner.

The offsets are indicated for the two-dimensional case in Figure 22.

In the second stage, the compact Eulerian cells are mapped to the full Eulerian grid. This is less straightforward than deriving the absolute positions of the compact Lagrangian cells in the CPU implementation. Lagrangian cells are fixed with respect to the particle, while the Eulerian cells are not. The positions of overlapping Eulerian cells are therefore also not fixed relative to the particle. As a result, the second data type needs to define Eulerian positions relative to a **reference Eulerian cell**.

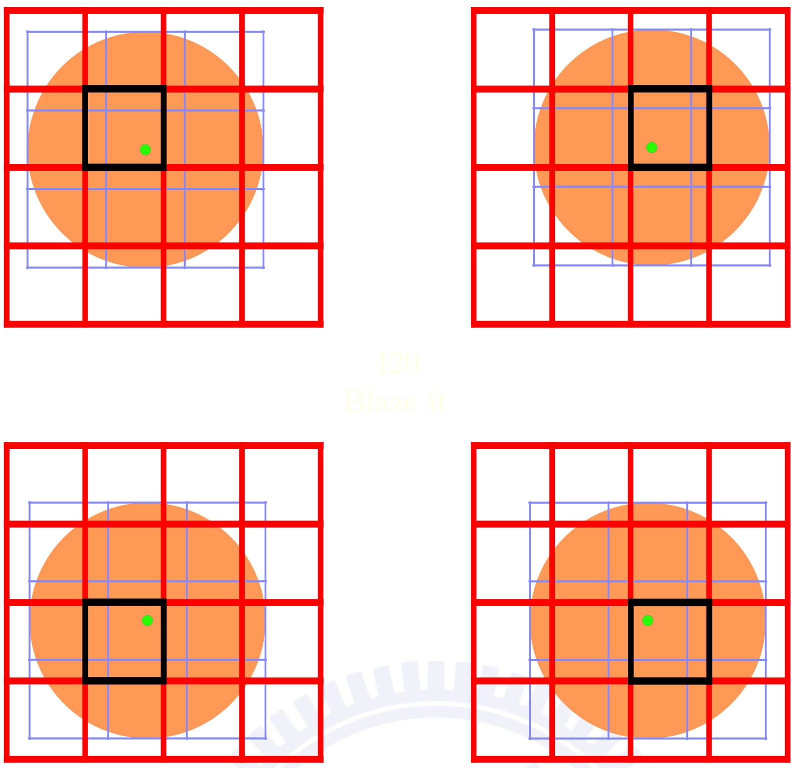
The defining property of the reference cell is that it is constant for all possible particle positions that correspond to a single Lagrangian-to-Eulerian correspondence pattern. Figure 24a shows that this cell is **not** necessarily the Eulerian cell that contains the cell center. The reference cell changes even as the correspondence pattern remains the same.

Instead, I have defined the reference cell as the Eulerian cell that contains the $(0, 0, 0)$-corner of the Lagrangian cell that the particle center is located in. Figure 24b shows how this reference cell is remains constant for the same correspondence pattern, even as the particle center moves into different Eulerian cells.

The second type of auxiliary data stores the positions of the compact Eulerian cells relative to the reference cell. During the Eulerian stage, the threads can then derive the reference cell from their particle's position and add the precomputed relative position to obtain their absolute location in global memory. To save on memory and computation, the relative position is saved as three integers instead of floating point numbers:

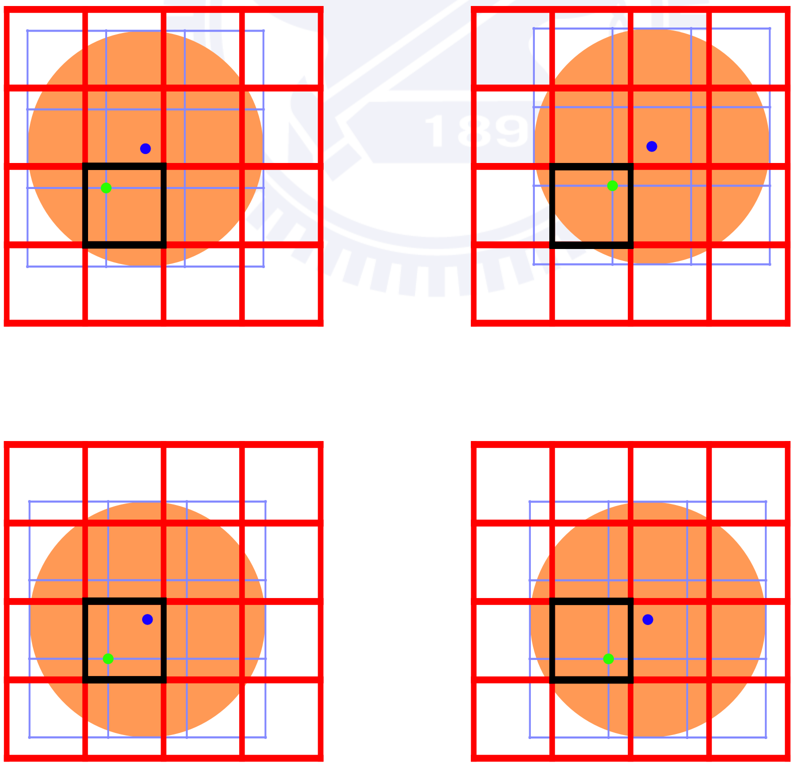
[1]When the Lagrangian and Eulerian grids are perfectly aligned in one or more dimensions, the number of overlapping cells is less. This corresponds to a special case where one of the overlapping volumes evaluates exactly to zero, so the general algorithm still applies.

[2]The definitions of the cell corners are found in Section 3.2.1.



120
Blaze it

(a) Eulerian reference cell contains particle center



(b) Eulerian reference cell contains Lagrangian cell corner

Figure 24: Eulerian reference cell definitions. The green dot indicates the Lagrangian point that determines the reference cell.

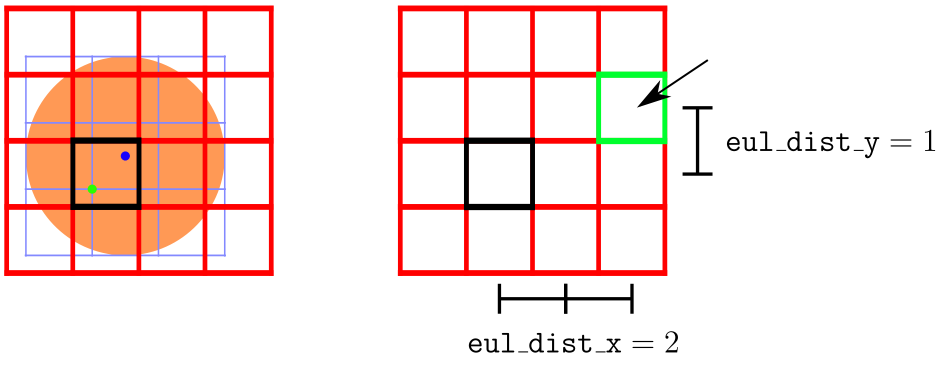


Figure 25: Position of Eulerian cell (green) relative to reference cell (black)

1. `eul_dist_x`: distance from reference Eulerian cell along x-direction.

2. `eul_dist_y`: distance from reference Eulerian cell along y-direction.

3. `eul_dist_z`: distance from reference Eulerian cell along z-direction.

Figure 25 gives an example for an Eulerian cell in the two-dimensional case.

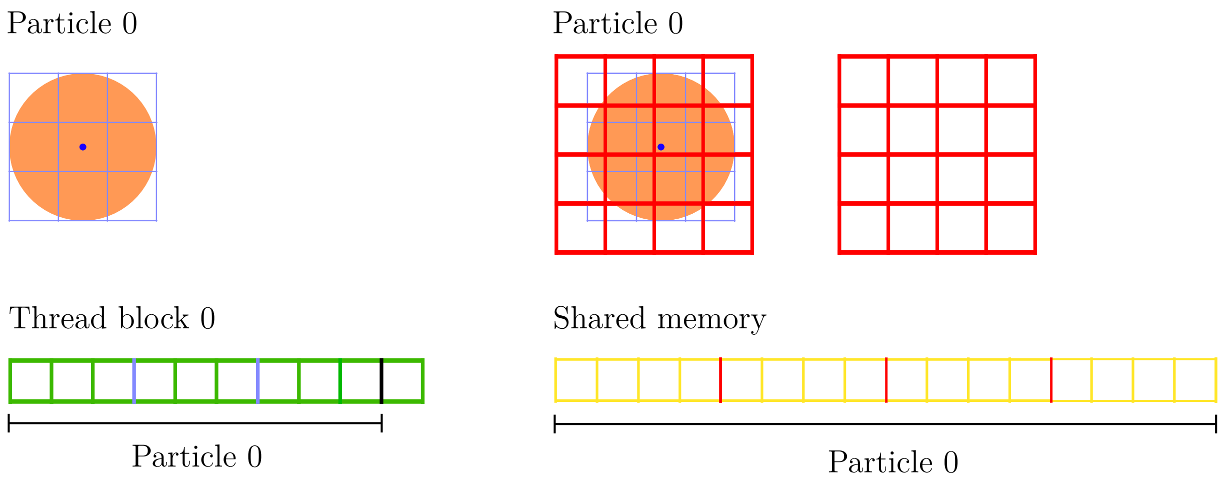
At first sight, it may seem relatively straightforward to compute the auxiliary data because the thread groups and shared memory partitions responsible for different particles are chained together. In the two-dimensional example discussed so far, 9 threads and 16 shared data elements are allocated per particle. Therefore, the shared memory offsets should simply repeat every 9 threads with an increment 16 data elements. Likewise, the relative Eulerian positions repeat every 16 Eulerian cells as well. However, this regular pattern breaks down when the block boundary does not overlap with the particle boundary. This means that the Lagrangian cells belonging to a single particle are allocated to a different thread blocks. This could potentially be avoided by moving that particle entirely to the next thread block, but that would introduce inactive threads at the tail end of every thread block. Moreover, in the general case, the number of Lagrangian cells in a single particle might exceed the number of threads in a thread block, which means that there is no choice but to spread out a single particle over different thread blocks.

The goal is thus to fill up every single thread block with Lagrangian cells, even across block boundaries. The procedure is as follows:

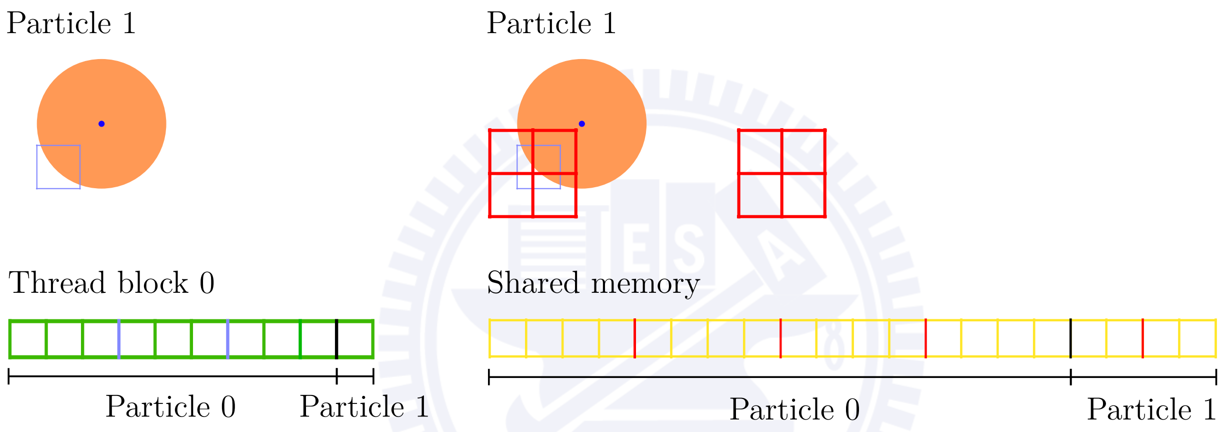
1. Initialize particle index $i = 0$ and thread block index $j = 0$.

2. Allocate the remaining free threads in block j to the remaining Lagrangian cells of particle i.

3. Compute the overlapping Eulerian cells with the newly allocated Lagrangian cells and reserve the required shared memory. Calculate and store the shared memory offsets `offset_0`, `offset_y`, `offset_z` and `offset_yz`.



(a) Left: allocation of threads to particle 0. Right: allocation of shared memory to particle 0.



(b) Left: allocation of threads to particle 1. Right: allocation of shared memory to particle 1.

Figure 26: Allocation of threads and shared memory in thread block 0. Steps 2 and 3 are illustrated for the first two particles.

4. Find the reference Eulerian cell and save the relative distances `eul_dist_x`, `eul_dist_y`, `eul_dist_z` for each overlapping Eulerian cell.

5. Repeat steps 2 to 4 while incrementing the particle index i until thread block j is completely filled.

6. When the thread block is full, increment thread block index j and return to step 2. If there are Lagrangian cells remaining for the current particle i, move them to the next thread block, otherwise increment i.

7. The procedure terminates in step 5 or step 6 when all particles have been treated.

This algorithm is illustrated step by step in Figure 26 for a two-dimensional case with 9 Lagrangian cells and a block size of 10. Since only one thread remains in thread block 0 for particle 1, the rest is moved to the next thread block, shown in Figure 27. The

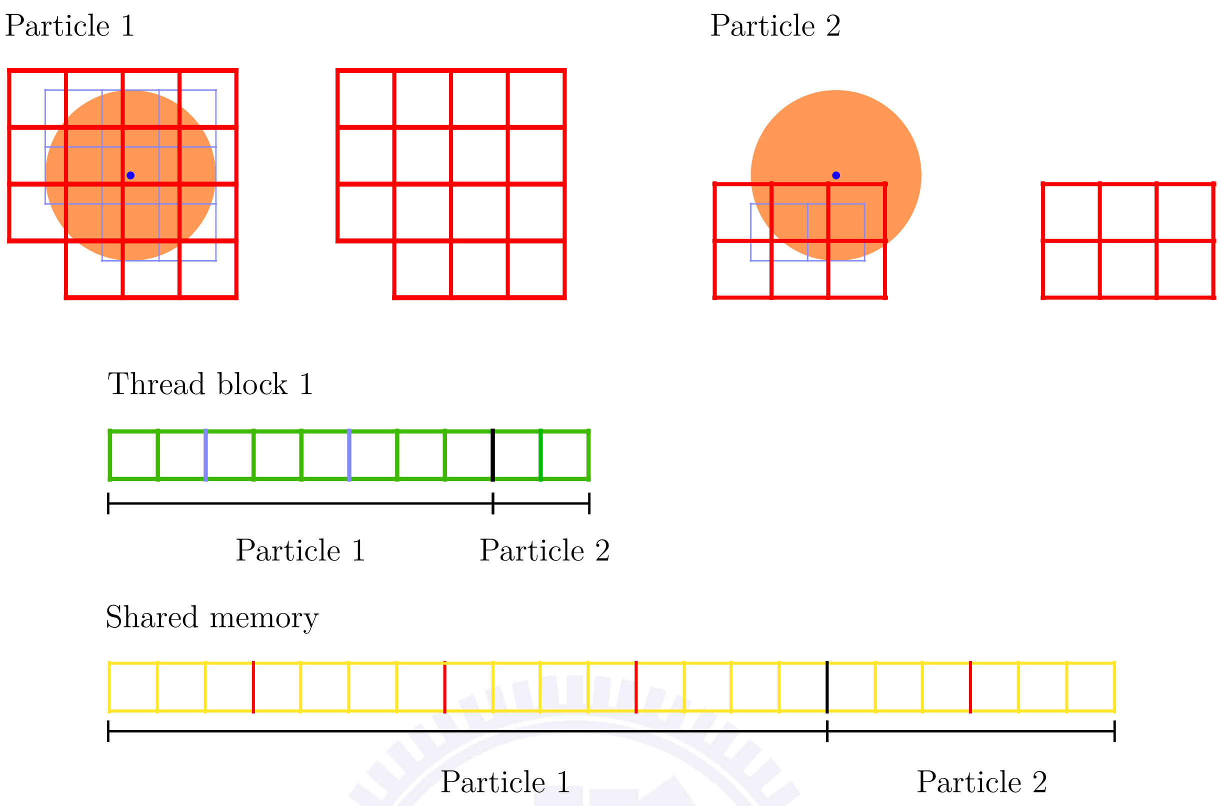


Figure 27: Allocation of threads and shared memory in thread block 1

figures reveal that the mapping pattern shifts from block to block. The total required shared memory is also not the same in thread block 0 and 1. This simple example thus demonstrates how thread block boundaries cause irregular allocation patterns, which in turn creates the need for auxiliary data.

It should be noted that the "allocation" of threads and shared memory should not be interpreted literally. In the preprocessing stage no actual shared memory or threads are physically allocated to particles. Rather, the preprocessing algorithm simulates how the threads *should* be allocated during the occupied volume interpolation and then calculates the corresponding auxiliary data. During kernel execution, the auxiliary data enforces the allocation pattern that was precomputed by the procedure above.

The potential downside of this scheme that the amount of auxiliary data makes the algorithm very costly in terms of memory usage. Every Lagrangian cell requires four integers of auxiliary data and every Eulerian cell requires three. This means that the space complexity of auxiliary data scales with the number of particles times the number of Lagrangian cells per particle. For the simulation parameters used in this work, every particle has 19 Lagrangian nonzero cells, which means that approximately 266 bytes are stored per particle if 16-bit integers are used. By comparison, storing the position of a particle consumes only 24 bytes using double precision floating-point numbers.

Fortunately, it not true that *all* thread blocks have different auxiliary data. Continuing on with the two-dimensional example above, figures 26 and 27 show that the particle boundary (black) is shifted relative to the block boundary by one thread when moving from thread block 0 to thread block 1. After 10 thread blocks, the particle boundary coincides with the block boundary again. Therefore, the mapping pattern for block 10 is exactly the same as for thread block 0. Likewise, the mapping pattern of the next block is the same as thread block 1 and so on. This means that there are only 10 unique mapping patterns that continuously repeat. Formally, the number of unique blocks is given by the formula[3]:

$$\#\text{ unique blocks} = \frac{\text{lcm}(\#\text{ Lagrangian cells per particle}, \#\text{ threads per thread block})}{\#\text{ threads per thread block}} \tag{V.4}$$

As a result, the mapping patterns only need to be stored for a constant number of blocks. For example, suppose a block size of 512 is used and there are 19 Lagrangian cells per particle. Since 19 and 512 do not have any common prime factors, the least common multiple is simply the product $19 \times 512 = 9728$. This means that there are 19 unique blocks that cover exactly 512 particles. The total size of the auxiliary data is then approximately 133 KiB, regardless of the number of particles.

The preprocessing procedure outlined above is modified slightly to take advantage of this fact. Specifically, the loop termination in step 7 should happen when all unique blocks have been calculated, instead of stopping when all particles have been exhausted.

5.5.4 Summary

The novel feature of the hybrid model is the Lagrangian grid that collocates with particles. This grid enables fast interpolation to and from the Eulerian grid that the fluid resides in. As a result, the occupied volume field does not have to be explicitly integrated at each timestep, but can be interpolated from the Lagrangian grid. Moreover, because particles by definition inhabit a spatially confined volume, the Lagrangian grid can be compressed into a compact data structure that stores only those cells with nonzero occupied volume, along with their position relative to the particle.

The original CPU implementation loops over the compacted Lagrangian cells and projects the cells directly onto the Eulerian field. This results in a poor cache performance because Eulerian cells are repeatedly modified at different locations in memory. The GPU implementation avoids this by mapping a compact Lagrangian data structure to a compact Eulerian data structure stored in shared memory. This algorithm exploits the fact that a particle's Lagrangian cells interpolate only to a small subset of the Eulerian grid, just like how a particle only occupies a small subset of the Lagrangian cells in the Lagrangian

[3]Here, lcm stands for least common multiple.

grid.

In other words, I have inserted a staging area between the compact Lagrangian data structure and the full Eulerian grid. The staging area is a compact representation of the Eulerian cells that the particle interpolates to. This method ensures an efficient interpolation step because the staging area resides in fast shared memory. The second step is to map the compact Eulerian cells to the Eulerian grid. In this stage, the CUDA threads correspond to Eulerian cells instead of Lagrangian cells. Therefore, throughout their lifetime, the CUDA threads go through two distinct stages: a Lagrangian-mapped interpolation step, where threads perform the interpolation for individual Lagrangian cells, and an Eulerian-mapped writing step, where threads are responsible for mapping individual Eulerian cells to the uncompressed Eulerian grid. Hence, a **two-stage mapping** scheme was employed.

The use of compact data structures means that the one-to-one correspondence between data elements is lost. Therefore, the data flow in the GPU algorithm needs to be carefully managed. Instead of deriving the mapping function on the fly, I decided to precompute **auxiliary data** on a per-thread block basis that the threads use during kernel execution to correctly route their data. The total memory overhead is minimized by exploiting the fact that there is only a limited number of unique mapping patterns that is continuously repeated over the computational grid. Therefore, the auxiliary data is only saved for a set of unique thread blocks. As a result, the amount of auxiliary data generated is independent of the problem size.

5.6 GPU-Accelerated Libraries

Two GPU-accelerated libraries were used in the implementation of the hybrid simulation. These are the NVIDIA CUDA Fast Fourier Transform (cuFFT) and the NVIDIA CUDA Random Number Generation (cuRAND) libraries. Both are part of the CUDA toolkit [16].

The cuFFT library was used to perform the fast Fourier transform, discussed in Section 3.4. It is a highly optimized library that performs FFT workloads up to ten times faster than CPU libraries.

Modeling the stochastic variables discussed in Chapter II requires sampling a large number of normally distributed variables (Section 3.6). The cuRAND library provides device functions to generate random numbers inside GPU kernels, enabling massively parallel sampling. Every thread possesses its own random number generator (RNG). To save the state of the RNG, the data type `curandState_t` is used. The number of `curandState_t` data objects that have to be allocated in global memory depends on the number of threads that are generating random numbers.

Here, another advantage of the grid-stride loop is exposed. Because the grid size is constant, the amount of memory allocated for `curandState_t` objects is also constant. Every thread reuses its RNG to generate as many random numbers as the problem requires. Therefore, the number of RNGs is independent of the problem size. The grid-stride loop therefore also helps with memory efficiency when generating random numbers.

VI Results and Discussion

6.1 Experimental Setup

The graphics card used in this work is the NVIDIA Tesla K80 containing two GK210 chips. These chips function as two independent GPU devices with identical technical specifications. Despite the availability of a second GPU, this work focused solely on porting the original implementation to a single GPU. The card was connected to the system with a 16-lane PCI Express 3.0 bus. All experiments were executed with double-precision floating-point arithmetic and with error-correcting code enabled. A summary of the technical specifications is found in Table 4.

The CPU used to run the FORTRAN implementation is an Intel Xeon E5-2687W v3 processor with 20 cores. However, the CPU code was written without any explicit parallelism and therefore ran only on a single core. Table 5 provides an overview of the CPU and memory configuration.

All programs were compiled and executed on a machine running the CentOS (release 6.7) operating system. The host code, which includes both FORTRAN and C++ code, was generated using the GNU Compiler Collection[7] (version 4.4.7). The CUDA Toolkit (version 7.5) provided GPU compilation tools, as well as GPU-accelerated libraries [16]. In addition to the C++ standard library and GPU-accelerated libraries, several Boost libraries [3] were employed to implement smart pointers, unordered sets and timers (Section 5.1.1).

6.2 Fluctuating Hydrodynamics Performance

Table 6 summarizes the achieved runtimes for a selection of the most important FHD subroutines. The gradient and divergence operators (Section 3.2), implemented using tile-stride kernels (Section 5.3), are approximately 20 to 30 times faster than their CPU counterparts. The divergence operator takes more time to complete than the gradient operator because tile-strided kernels only compute the derivative for a single dimension. Therefore, the calculation of the divergence involves first performing backward sums for each of the three vector derivatives, and then summing the derivatives. The CPU version on the other hand calculates the divergence on a per-element basis, which is why the speedup is not as large as for the gradient operator.

[1]Since shared memory and L1 cache both exist on the same physical SRAM, they can be allocated differently. For the GK210, there are three configurations: 16 KiB–48 KiB shared–L1, 32 KiB–32 KiB shared–L1 and 48 KiB–16 KiB shared–L1.

Table 4: Tesla K80 technical specifications

Tesla K80	
GK210 chips	2
GK210	
Streaming multiprocessors	13
CUDA cores	$13 \times 192 = 2496$
GPU maximum clock rate	824 MHz
Memory size	GDDR5 12 GiB
Memory clock rate	2505 MHz
Memory bus width	384 bit
L2 cache	1.5 MiB
Streaming multiprocessor	
CUDA cores	192
Registers	65 536
Shared memory	16 KiB–48 KiB [1]
L1 cache	16 KiB–48 KiB

Table 5: System hardware configuration

Intel Xeon E5-2687W v3	
CPU clock rate	3.1 GHz
L1 cache	
Data	32 KiB
Instruction	32 KiB
L2 cache	256 KiB
L3 cache	25 MiB
RAM	
Memory	DDR4 62 GiB
Memory clock rate	2133 MHz

Table 6: Profiling of fluctuating hydrodynamics subroutines

Grid size	CUDA runtime (ms)	FORTRAN runtime (ms)	Speedup
		Gradient	
128^3	0.76	18.32	24.14
256^3	5.79	146.51	25.32
512^3	38.91	1172.12	30.14
		Divergence	
128^3	1.10	18.73	17.07
256^3	7.92	151.63	19.16
512^3	54.63	1181.92	21.64
		Convective flux	
128^3	5.98	116.23	19.43
256^3	38.56	943.64	24.42
512^3	305.39	13 414.97	43.93
		Viscous flux	
128^3	6.04	115.68	19.17
256^3	40.09	969.78	24.19
512^3	326.17	11 877.48	36.42
		Poisson's equation	
128^3	2.50	888.16	355.70
256^3	18.50	11 710.57	633.17
512^3	183.46	136 084.32	742.11
		Fluctuating mass flux	
128^3	0.86	350.38	408.54
256^3	5.86	2806.78	479.80
512^3	47.71	22 440.34	470.38

A common metric for analyzing the performance of data-intensive kernels is the "effective bandwidth" [5]. This is the total amount of bytes that was either read or written to global memory, divided by the kernel runtime. Formally:

$$\text{effective bandwidth} = \frac{\text{bytes read} + \text{bytes written}}{\text{elapsed time}} \tag{VI.1}$$

If a kernel has a higher bandwidth than another kernel, that means it performs better However, this by itself does not give us not more information than just looking at the runtime. The advantage of using bandwidth as a metric is that it can be compared against the theoretical peak bandwidth of the device. This theoretical peak bandwidth is computed from the memory hardware specifications as:

$$\text{theoretical bandwidth} = 2 \times \text{clock speed} \times \text{bus width} \tag{VI.2}$$

The factor 2 comes from the fact that GDDR5 is a "double data rate" type of RAM. This means that two data transfers happen on every clock cycle. For the Tesla K80 (Table 4), the theoretical peak bandwidth is $2 \times 2505\,\text{MHz} \times 384\,\text{bit} \approx 224\,\text{GiB/s}$. This is a hard limit on the effective bandwidth that can be realized. It is important to realize however, that the theoretical peak bandwidth literally disregards all sources of overhead and thus can never be achieved by a kernel in practice. The rule of thumb is that utilizing between 60% and 80% of the theoretical peak bandwidth is considered efficient [5]. Another important caveat is that enabling error-correcting code (ECC) lowers memory performance. All the profiling data was collected with ECC enabled, so the peak bandwidth derived above should be corrected to account for that. I therefore conducted bandwidth tests with and without ECC and found that the bandwidth penalty is around 10%. The corrected peak bandwidth is thus 202 GiB/s.

A closer look at the gradient and divergence performance (Table 7) reveals that the DRAM bandwidth utilization is fairly high in general, ranging from 55% to 75%. For both operations, the bandwidth improves as the problem size grows larger. The reason for this is that more data means that there are more threads issuing memory transactions to saturate the device and thus the memory latency is hidden more effectively. Also, a larger proportion of time is spent processing data instead of creating and initializing threads. Moreover, every tile-strided kernel starts with a memory access that is repeated later on, as explained in Section 5.3.3. The proportion of repeated accesses to nonrepeated accesses is small for larger grid sizes, and thus the overall performance goes up. This also explains why the recorded speedups over the CPU implementation (Table 6) increase with the grid size.

The contribution of convective and viscous fluxes to the momentum density accumulation (Section 2.1) is calculated by applying the divergence operator to second-order tensors. The numerical implementation was not discussed in detail, but it is based on the

Table 7: Effective bandwidth of gradient and divergence operations

Grid size	Data (MiB)	Runtime (ms)	Bandwidth (GiB/s)	Ratio to peak (%)
		Gradient		
128^3	96	0.76	123.57	61.30
256^3	768	5.79	129.64	64.31
512^3	6144	38.91	154.22	76.51
		Divergence		
128^3	128	1.10	113.92	56.52
256^3	1024	7.92	126.34	62.68
512^3	8192	54.63	146.45	72.65

same tiling strategy and tile-strided loop as discussed in Section 5.3, though the computation is more complex and more data is processed. This is reflected in an approximately six times larger runtime than the divergence operator on vector fields for the same grid size. Similarly to the gradient and divergence operators, the speedup relative to the CPU implementation is higher for larger problem sizes. Comparing the runtimes over different grid sizes reveals that the CPU version fails to scale linearly as the grid size increases from 256^3 to 512^3 (an eight-fold increase in grid points). On the other hand, the GPU scales slightly better when going from 256^3 to 512^3 than the step before it. The reasons for an improved bandwidth at larger problem sizes is the same as for the gradient and divergence operators discussed above.

The solution of Poisson's equation (Section 3.4.3) provides the largest acceleration encountered for any particular subroutine. A speedup of almost three orders of magnitude is achieved for the 512^3 grid. While this is a spectacular result, comparing its runtime to the tile-strided kernels discussed so far shows that it is situated between the gradient/divergence and tensor operators. This makes sense because the amount of data handled is less than the tensor flux calculations, but the complexity of the algorithm is higher than that of the gradient/divergence operators. Therefore, the large jump in performance can be largely attributed to an inefficient CPU implementation. Both implementations do not scale linearly because the fast Fourier transform is a $O(N\log N)$ algorithm, as opposed to the tile-stride kernels.

Lastly, the profiling of the fluctuating mass flux calculations (Section 2.3) was included because the subroutine involves almost exclusively the generation of random numbers. As a result, it provides the most direct method for measuring the performance of random number generation. The amount of memory traffic is similar to the gradient operator,

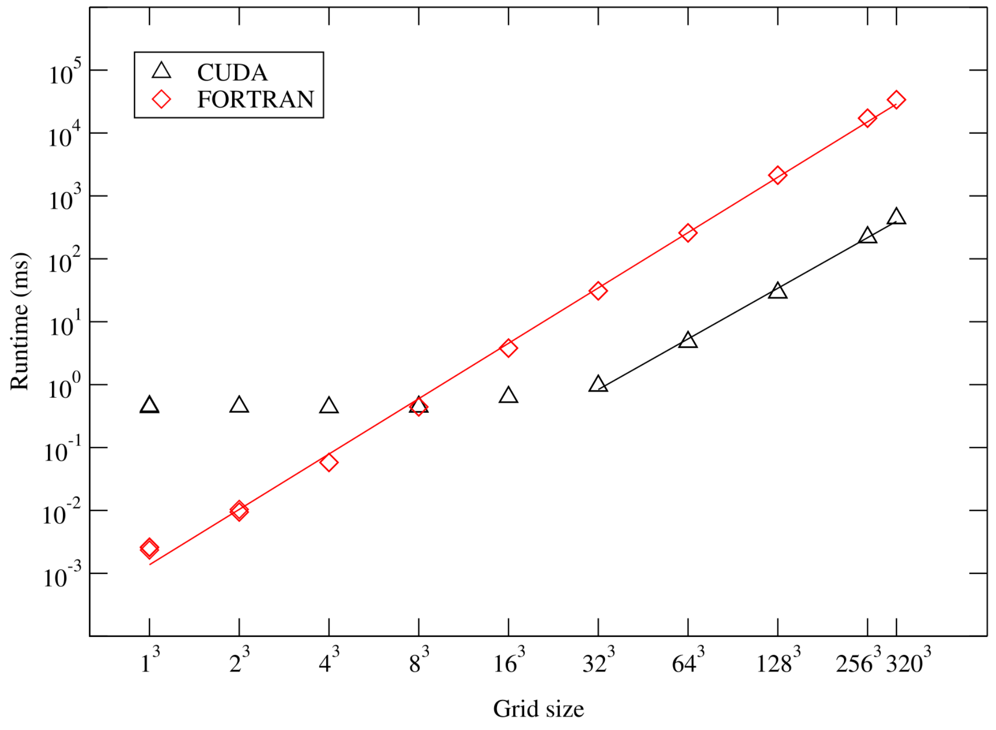


Figure 28: Runtime complexity of fluctuating hydrodynamics calculations

which is why the runtime is comparable as well. Both implementations scale linearly, resulting in a roughly constant speedup of 400. Therefore, although the CPU implementation does not perform worse for large grid sizes, it cannot compete with the GPU implementation.

The calculation of the FHD derivatives was also profiled as a whole. The computed quantities are $\partial \mathbf{g}/\partial t$, $\partial \rho^A/\partial t$ and $\partial \rho^B/\partial t$, and therefore the subroutine spans steps 9–10 and 13–14 in the procedure outlined in Section 3.7. It is representative of the overall GPU acceleration achieved for the FHD simulation. The computational time is plotted against the grid size on a log–log scale in Figure 28. The configurations profiled range from the smallest possible grid, containing a single cell, to the largest size (352^3) that the Tesla K80 could accommodate.

The figure shows that there are two distinct regions in the GPU runtime data. For box dimensions of 16^3 and smaller, the GPU takes approximately half a millisecond to compute the derivatives, regardless of the grid size. In this region, the computational time is dominated by memory latency and parallel overheads because there are not enough threads active to hide memory latency. From 32^3 upwards, the data sets are large enough to saturate the device and the trend stabilizes to a linear relationship with respect to the problem size. The CPU on the other hand scales linearly for all problem sizes because

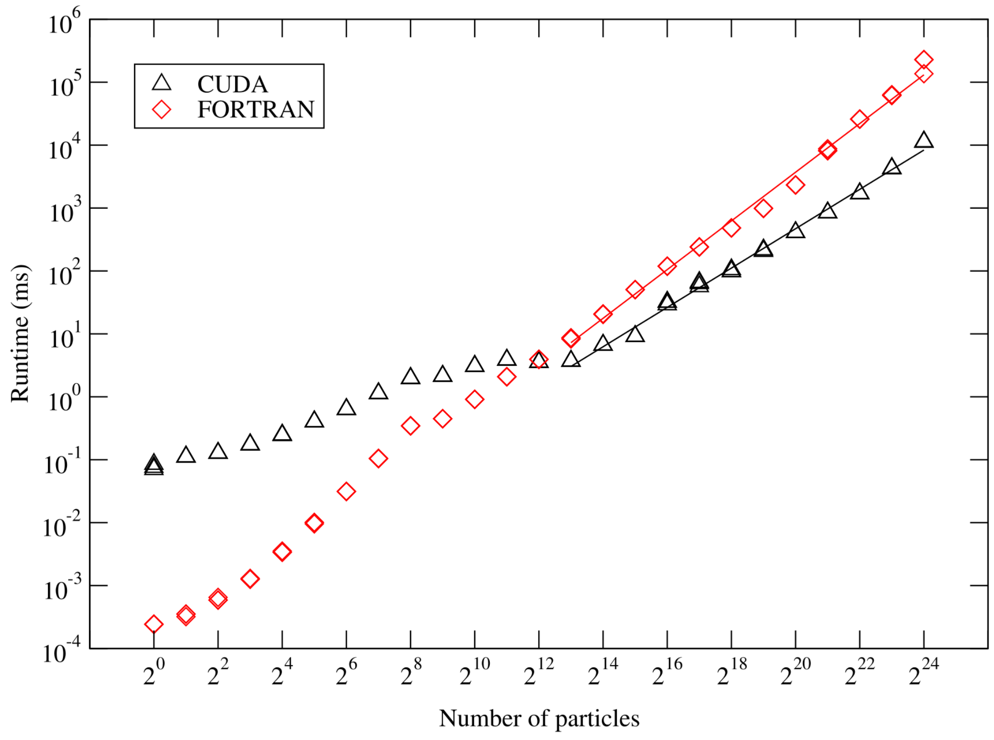


Figure 29: Runtime complexity of neighbor list calculation

it does not suffer from the same overheads as the GPU. As a result, it outperforms the CPU for small grid sizes. However, at grid size 8^3, the CPU runtime rises above the GPU runtime. This is the point where the parallel processing power of the GPU starts to have an edge over the CPU, also called the **break-even point**. Finally, in the linear region, a performance gap of approximately two orders of magnitude is established. The experimental data is consistent with the observation made earlier that the amount of utilized bandwidth for tile-strided loops increases with the data size. Small problem sizes do not provide enough parallelism for the GPU to exploit, resulting in a poor performance compared to the CPU. Larger problems on the other hand benefit greatly from the GPU architecture because then the parallel overhead is just a small fraction of the total computational time and all the available GPU hardware can be put to use. As a result, the massively parallel structure of the GPU leads a much greater performance than the CPU for large data sets.

6.3 Neighbor List Performance

The runtime of the particle-based simulation is dominated by neighbor list calculations. The original implementation of the hybrid model employs a brute-force $O(N_p^2)$ algorithm to construct a neighbor list. Up until this work, many-particle simulations of the hybrid model had not yet been attempted so this was not an issue. For larger system sizes however, it is imperative to employ more sophisticated algorithms such as the cell lists algorithm (Section 3.5). It would have been unfair to compare the original brute-force algorithm on the CPU to the cell lists algorithm on the GPU because of the different asymptotic runtime complexity. I have therefore implemented the cell lists algorithm in FORTRAN using linked lists [1].

Figure 29 thus plots the runtimes of CUDA and FORTRAN implementations of the cell lists algorithm in function of the number of particles, with both axes in logarithmic scales. The smallest system consists of a single particle. The subsequent system sizes double the number of particles up until the limit of GPU memory is reached at 2^{24} particles. The cutoff radius was 2.5σ, and the box size was scaled to keep a constant particle density of $0.4\sigma^{-3}$ for 2^8 particles and above. Smaller box sizes were not possible for the lower range of particle numbers because the cell lists algorithm requires that the box length is at least three times the cutoff radius. All initial particle positions were randomly sampled over the simulation box.

The CPU initially outperforms the GPU by more than three orders of magnitude when there is just a single particle. However, the runtime then grows steeply up to a system size of 2^8 particles. After this initial climb, the CPU implementation displays a stable growth with respect to the system size. The two distinct segments in the runtime complexity are a direct result of the sparsity in the first 8 experiments. The average number of neighbors per particle is initially very low and then grows quickly until the specified particle density of $0.4\sigma^{-3}$ is reached at 2^8 particles. From this point on, the density of particles remains constant and thus the curve stabilizes.

The CUDA implementation on the other hand is not as much affected by the different particle densities. Similarly to the FHD simulation, the predominant bottlenecks are the low device saturation and large latencies involved with launching GPU workloads. As a result, the relatively large initial computational time increases only slightly prior to the break-even point at 4096 particles. After that the particles generate a large enough workload that the runtime increases linearly in the number of particles, with a slope of 1.0387 obtained by linear regression. From visual inspection, it is clear that the FORTRAN version does not scale as linearly for the same data points. Indeed, curve fitting derives a coefficient of 1.2854. As a result, an initial 2.3 times speedup for 2^{13} particles gradually increases to a 17.5 times speedup at 2^{24} particles.

Overall, the GPU advantage is modest compared to the two orders of magnitude difference in the FHD simulation. This is because particles may have varying numbers of neighbors, which results in an imbalanced workload over threads. Moreover, the data access patterns are not fixed as in the FHD model. This results in scattered reads and therefore less efficient bandwidth utilization. Advanced techniques such as particle sorting and texture caching may further accelerate the neighbor list search [2, 9].

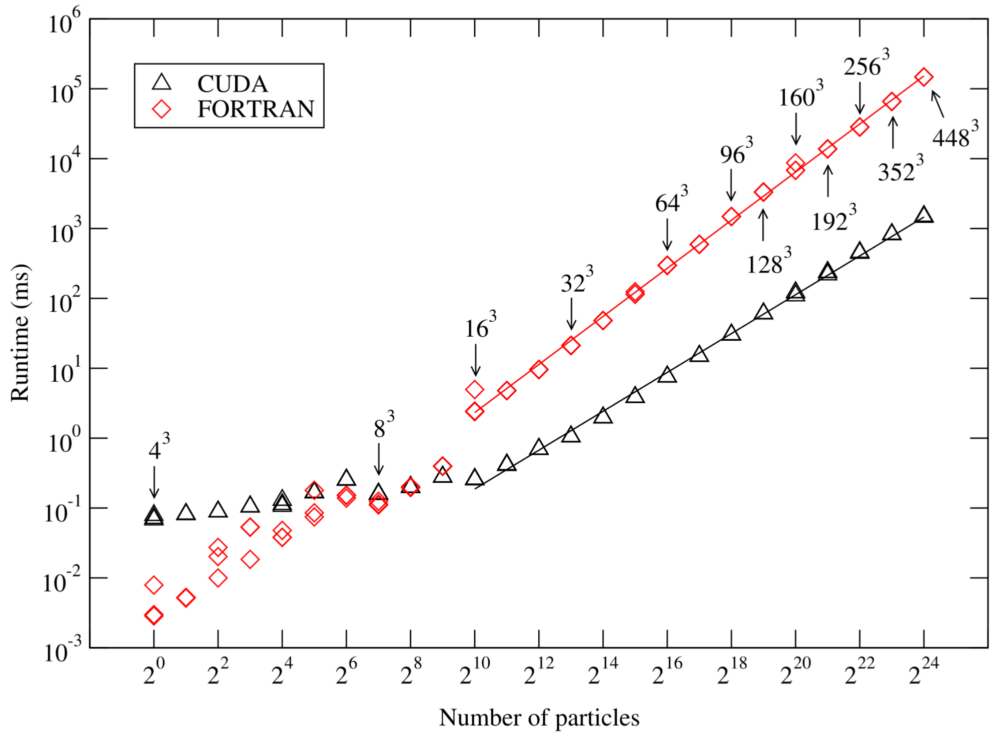


Figure 30: Runtime complexity of occupied volume calculation

6.4 Occupied Volume Interpolation Performance

The computational times for occupied volume interpolation (Section 5.5) is plotted against the number of particles in Figure 30 on a log–log scale. The grid size was scaled with the number of particles, as indicated on the graph, to ensure a relatively constant particle density. The problem sizes profiled range from a single particle to 2^{24}—approximately 16.8 million—particles.

Similarly to the FHD simulations, the CPU takes less time than the GPU when the total workload is small. However, in this case, the CPU advantage is less pronounced and does not exceed two orders of magnitude. The computational times quickly approach each other in the 32–128 range and converge for 256-particle systems. From 1024 particles and above, both implementations stabilize and display a constant scaling behavior. Fitting straight lines in this region results in a slope of 1.1431 for the CPU and 0.92366 for the GPU. This shows that the CPU scales slightly worse than the ideal linear scaling behavior (slope equal to one), while the GPU scales slightly better. This translates into an increasing performance gap, starting from one order of magnitude at the start of regression curve and ending with a 100-fold speedup for the largest system size.

The reason for the subpar CPU performance is that the CPU implementation suffers from low spatial locality, resulting in frequent cache misses. This is effect is amplified

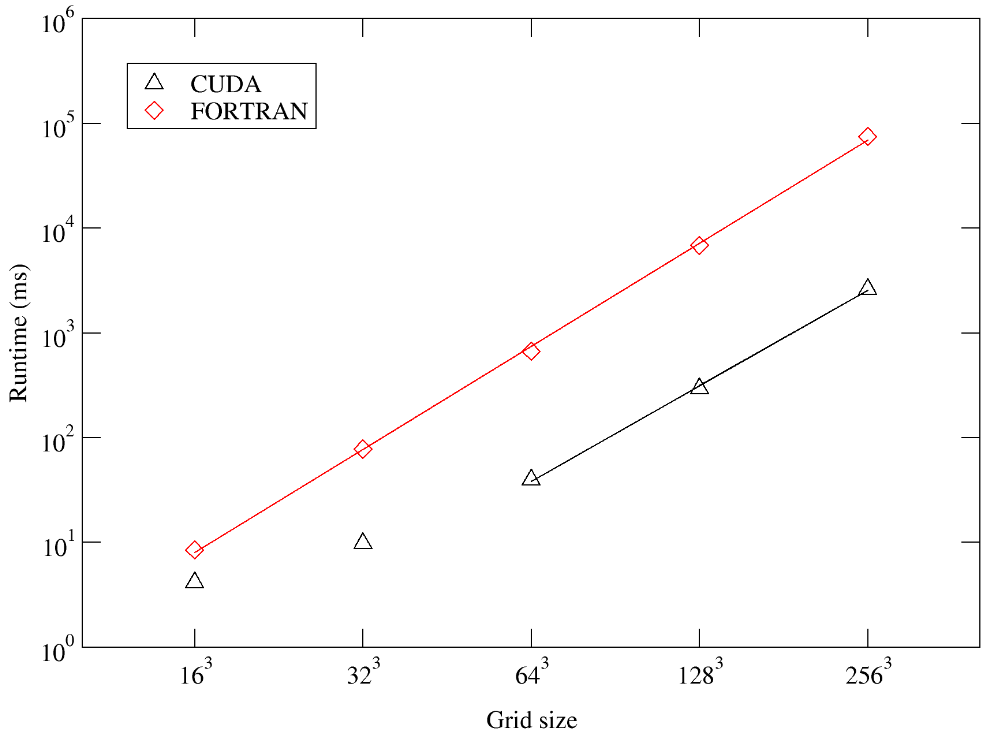


Figure 31: Runtime complexity of simulation timestep

as the grid size grows larger, which explains the nonideal scaling. On the other hand, the GPU algorithm presented in this work ensures that the access pattern to the GPU's shared memory cache is independent from the problem size. As a result, larger data sets do not adversely affect the performance of the interpolation step. To the contrary, a larger number of particles allows the GPU to utilize more of its available memory bandwidth and therefore the performance improves with the problem size.

In conclusion, the data demonstrates that the GPU implementation of the Lagrangian to Eulerian interpolation vastly outperforms the CPU for systems containing at least a thousand particles. Since MD simulations typically involve tens of thousands of particles or more, the novel GPU algorithm presented in this work is the best option for most practical applications. Larger system sizes benefit from an even greater performance boost because the GPU implementation scales very well, whereas the CPU struggles to maintain a linear runtime complexity.

6.5 Simulation Performance

Finally, whole-system simulations were profiled as well. This includes all the equations described in Chapter II and numerically implemented as specified by Chapter III. In the simulation experiments, the spatial domain was subdivided into field grids with the number of divisions across the side equal to powers of two. The same constant particle density and cutoff radius were used as in Section 6.3. The particles were assigned a radius of 5 Å, and therefore the coupling radius R_{coupling} and sigma parameter σ were equal to 5 Å and 10 Å respectively. The cell widths d_x, d_y and d_z were chosen to be equal to 5 Å as well. Because the box size has to be at least three times the cutoff radius, 16^3 was the smallest valid grid size under these conditions. On the other hand, 256^3 was the largest simulation that was achievable with the Tesla K80. Half of the particles were given a positive elementary charge, and the other half a negative elementary charge. Finally, the particles were initially placed in a cubic grid exactly 10 Å apart.

Every configuration ran for 1000 iterations to obtain an accurate average of the time taken to compute a single timestep. The results are shown in Figure 31. Like previous figures, the data is plotted on a log–log scale. At the smallest problem size simulated, with a grid size of 16^3 and 208 particles, both the CUDA and FORTRAN runtimes have the same order of magnitude–between one and ten milliseconds–although even at this configuration the GPU is twice as fast as the CPU. This indicates that taken together, the GPU will outperform the CPU for all except the smallest problem sizes. From this point on, the CPU runtime maintains a roughly linear behavior, up until the largest system with 838 862 simulated particles within a 256^3 grid. On the other hand, the gradual rise of the GPU runtime indicates that the device is not saturated for systems smaller than 64^3. The linear region thus comprises of the last 3 system sizes.

Although both runtime are very close to scaling perfectly linearly and thus the curves run approximately parallel, both implementation are slightly worse than linear. This is not surprising because not all algorithms included in the timestep calculation scale linearly, e.g. the fast Fourier transform. However, the deviation is stronger for the FORTRAN version, leading to a 16.9-fold GPU acceleration at 64^3 that accumulates into a 28.6-fold speedup at 256^3.

Overall, the GPU has better performance than the CPU for system sizes that have a grid size of at least 16^3. The decisive factor in achieving the speedup for small system sizes is not the particle-based model, as the 208-particle system is not past the break-even point for neighbor list calculations (see Section 6.3). Rather, the GPU-acceleration of field calculations initially drives the overall speedup, with the other components running just fast enough not to be a bottleneck. As the system sizes grow larger, the benefits from faster neighbor list calculations and occupied volume calculations start to take effect as well. This finally results into a saturated GPU device starting at around 64^3 and 13 108 particles, with accelerations ranging between 15 times and 30 times speedup that get larger for bigger systems.

VII Conclusion

In this work, I demonstrated that all parts of the hybrid FHD/MD can be ported into the GPU architecture and a significant speedup can be achieved in all components of the computational model. For numerical calculations of field variables, such as the gradient and divergence operators, a newly designed scheme of high data throughput using internal shared memory is employed. Specifically, I showed that using grid-strided and tile-strided kernels ensures that execution parameters can be tuned to achieve the highest performance for arbitrary system sizes as well as any GPU hardware configuration. Furthermore, the GPU-accelerated libraries cuFFT and cuRAND were integrated seamlessly into the hybrid modeling framework for fast Fourier transform operations and generating random numbers, respectively. For particulate variables, the cell list algorithm for identifying neighboring particles was implemented in the GPU architecture to eliminate the severe bottleneck associated with brute-force generation of the neighbor list.

The interpolation of occupied volume fields was greatly accelerated by introducing a staging area between the compact Lagrangian grid and the Eulerian grid. The nonzero Lagrangian occupied volume values are interpolated to a compact Eulerian data structure in shared memory before storing the interpolated values to global memory. This data flow ensures that repeated modifications of Eulerian values are performed in fast shared memory and the number of accesses to global memory is minimized. Moreover, I designed a two-stage mapping scheme that guarantees a balanced workload over all threads. This novel algorithm incurs an additional cost in the form of auxiliary data. In particular, the process of interpolating between compact Eulerian and Lagrangian data structures, as well as the mapping from shared memory to global memory, both need extra data. In a naive approach, the extra memory required scales with the number of particles. However, I have exploited repetitions in the mapping patterns so that the total memory overhead is reduced to a small amount of data whose size is independent from the problem size. Therefore, the GPU-accelerated interpolation scheme presented in this work is a highly efficient algorithm that uses shared memory as a staging area, balances its work equally over all threads and requires only a small and constant amount of auxiliary data.

In conclusion, this work shows that the GPU architecture is a suitable computational platform for the hybrid FHD/MD model. Its unique coupling scheme can be implemented efficiently on the GPU with the use of shared memory. As a result, the spatial and temporal scales that are within reach of the hybrid model have been greatly expanded beyond the practical limitations of the original CPU implementation. In future work, the scalability of the FHD/MD model may be further improved by decomposing larger simulations over multiple GPUs.

Bibliography

[1] M. P. Allen and D. J. Tildesley. “Computer simulation of liquids”. Oxford: Clarendon Press, 1987.

[2] J. A. Anderson, C. D. Lorenz, and A. Travesset. “General purpose molecular dynamics simulations fully implemented on graphics processing units”. In: “Journal of Computational Physics” 227.10 (2008), pp. 5342–5359.

[3] Boost Community. “Boost C++ Libraries”. URL: `http://www.boost.org/` (visited on 07/25/2016).

[4] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. “CHARMM: A program for macromolecular energy, minimization, and dynamics calculations”. In: “Journal of Computational Chemistry” 4.2 (Jan. 1983), pp. 187–217.

[5] J. Cheng, M. Grossman, and T. McKercher. “Professional CUDA C Programming”. Indianapolis, Indiana: John Wiley & Sons, Inc., 2014.

[6] W. M. Deen. “Analysis of transport phenomena”. Oxford: Oxford University Press, 2013.

[7] Free Software Foundation Inc. “GNU Compiler Collection”. 2016. URL: `https://gcc.gnu.org/`.

[8] D. Frenkel and B. Smit. “Understanding molecular simulation: from algorithms to applications”. New York: Academic Press, 2002.

[9] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer. “Strong scaling of general-purpose molecular dynamics simulations on GPUs”. In: “Computer Physics Communications” 192 (2015), pp. 97–107.

[10] A. Grama, A. Gupta, G. Karypis, and V. Kumar. “Introduction to Parallel Computing; 2nd Edition”. Upper Saddle River, New Jersey: Addison-Wesley, 2003.

[11] M. Harris. “CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops”. 2013. URL: `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/` (visited on 07/25/2016).

[12] M. J. Harris. “Fast Fluid Dynamics Simulation on the GPU”. In: “GPU gems: programming techniques, tips, and tricks for real-time graphics”. Upper Saddle River, New Jersey: Addison-Wesley, 2004.

[13] J. L. Hennessy and D. A. Patterson. “Computer architecture: a quantitative approach”. Boston: Elsevier, 2012.

[14] D. Kirk and W.-M. W. Hwu. “Programming Massively Parallel Processors: A Hands-on Approach”. Boston: Elsevier, 2010.

[15] NVIDIA Corporation. “CUDA C Programming Guide”. September. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 07/25/2016).

[16] NVIDIA Corporation. “CUDA Toolkit”. 2016. URL: https://developer.nvidia.com/cuda-toolkit (visited on 07/26/2016).

[17] NVIDIA Corporation. “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi”. In: (2009). URL: http://www.nvidia.com.tw/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[18] R. A. X. Persson, N. K. Voulgarakis, and J.-W. Chu. “Dynamic mesoscale model of dipolar fluids via fluctuating hydrodynamics”. In: “Journal of Chemical Physics” 141.17 (2014).

[19] S. Popinet and S. Zaleski. “A front-tracking algorithm for accurate representation of surface tension”. In: “International Journal for Numerical Methods in Fluids” 30.6 (1999), pp. 775–793.

[20] J. Sanders and E. Kandor. “CUDA By Example”. Upper Saddle River, New Jersey: Addison-Wesley, 2010.

[21] N. K. Voulgarakis and J.-W. Chu. “Bridging fluctuating hydrodynamics and molecular dynamics simulations of fluids”. In: “Journal of Chemical Physics” 130.13 (2009).

[22] N. K. Voulgarakis, B. Z. Shang, and J.-W. Chu. “Linking hydrophobicity and hydrodynamics by the hybrid fluctuating hydrodynamics and molecular dynamics methodologies”. In: “Physical Review E - Statistical, Nonlinear, and Soft Matter Physics” 88.2 (2013).