Development and Application of Real-time and Interactive Software for Complex System

by

Hamidreza Soltani

A Thesis submitted in partial fulfilment for the requirements for the degree of Doctor of Philosophy at the University of Central Lancashire

October 2016

I declare that while registered as a candidate for the research degree, I have not been a registered candidate or enrolled student for another award of the University or other academic or professional institution.

I declare that no material contained in the thesis has been used in any other submission for an academic award and is solely my own work.

Signature of Candidate Hamidreza Soltani [illegible]

Type of Award **Doctor of Philosophy**

School **Physical Sciences and Computing**

Abstract

Soft materials have attracted considerable interest in recent years for predicting the characteristics of phase separation and self-assembly in nanoscale structures. A popular method for demonstrating and simulating the dynamic behaviour of particles (e.g. particle tracking) and to consider effects of simulation parameters is cell dynamic simulation (CDS). This is a cellular computerisation technique that can be used to investigate different aspects of morphological topographies of soft material systems. The acquisition of quantitative data from particles is a critical requirement in order to obtain a better understanding and of characterising their dynamic behaviour. To achieve this objective particle tracking methods considering quantitative data and focusing on different properties and components of particles is essential.

Despite the availability of various types of particle tracking used in experimental work, there is no method available to consider uniform computational data. In order to achieve accurate and efficient computational results for cell dynamic simulation method and particle tracking, two factors are essential: computing/calculating time-scale and simulation system size. Consequently, finding available computing algorithms and resources such as sequential algorithm for implementing a complex technique and achieving precise results is critical and rather expensive. Therefore, it is highly desirable to consider a parallel algorithm and programming model to solve time-consuming and massive computational processing issues. Hence, the gaps between the experimental and computational works and solving time consuming for expensive computational calculations need to be filled in order to investigate a uniform computational technique for particle tracking and significant enhancements in speed and execution times.

The work presented in this thesis details a new particle tracking method for integrating diblock copolymers in the form of spheres with a shear flow and a novel designed GPU-based parallel acceleration approach to cell dynamic simulation (CDS). In addition, the evaluation of parallel models and architectures (CPUs and GPUs) utilising the mixtures of application program interface, OpenMP and programming model, CUDA were developed. Finally, this study presents the performance enhancements achieved with GPU-CUDA of approximately ~2 times faster than multi-threading implementation and 13~14 times quicker than optimised sequential processing for the CDS computations/workloads respectively.

Contents

List of Figures

List of Tables

Acknowledgements

I am indebted and grateful to many people for the support, encouragement, advice and friendship that they have offered during my studies at the University of Central Lancashire. It is a great pleasure to acknowledge them here.

I would like to first express my sincere gratitude to my supervisors, Dr Dung Ly and Professor Waqar Ahmed for their vital guidance, patience and encouragement over the past years. Many thanks for your confidence in me and your enthusiasm.

Thanks go to all my colleagues and friends at University of Central Lancashire and Preston city for their help, and for reminding me that is fine to have fun occasionally.

My heartfelt and special gratitude to my other dear friends for their precious support and motivation, which drives me to give my best: Dr Maneea Eizadi, Dr Marjan Famili, Dr Sandeep Kadam and Mr Mouayed Rushdi, thanks to one and all.

I would like to acknowledge and thank my examiners Professor Dermot Brabazon and Dr Monika Gamza for being my thesis examiners, and also extend my sincere words of gratitude to Dr Tim Cawthorne, Mrs Fiona Mair, Dr Marco Pinna and Professor Derek Ward-Thompson for all their professional help, advice and consideration.

I also wish to acknowledge the University of Central Lancashire for partial financial assistance and support during my studies.

Last but not the least, a huge and special thanks to my family who mean world to me: my parents, grandmother and my brothers (Dr Mohammadreza Soltani and Dr Amirreza Soltani) for their support and love. This journey would not have been possible without the support of them. They kept me sane throughout this journey and inspired me to better myself. I consider myself the luckiest in the world to have such a supportive family, standing behind me with their love and support. I dedicate this thesis to my parents.

Research Output

- H. Soltani, D. Ly and W. Ahmed, "Computational Technique of Particle Tracking", *Fourth Annual Research Student Conference*, UCLAN, Preston, December 2014.

- H. Soltani, D. Ly and W. Ahmed, " Accelerating Cell Dynamic Simulation for 3D Diblock copolymer Sphere Morphology using GPU ", *GPU Technology Conference*, San Jose, USA, March 2015.

- H. Soltani, D. Ly and W. Ahmed, "Accelerating Cell Dynamics Simulation of Soft Materials using GPU - CUDA", *Journal of Materials today: proceeding 2016*.

- H. Soltani, D. Ly and W. Ahmed, "Parallel Implementation for Cell Dynamics Simulation of Diblock Copolymers based on Multi-core CPU and Many-core GPU", *Journal of Computational Physics*. (submitted)

Nomenclature

a	Phenomenological constant
b	Chain-length dependence to the free energy
D	Positive constant for diffusion coefficient
F	Free energy functional
f	Global volume fraction of A monomers in the diblock
$f(\psi)$	Map function
$F(\psi)$	Free energy functional
$H(\psi)$	Free energy function
M	Phenomenological mobility constant
N_A	Number of A monomers
N_B	Number of B monomers
N_x	The length of the x dimension of a simulation system
N_y	The length of the y dimension of a simulation system
N_z	The length of the z dimension of a simulation system
N_P	Number of detected points on the grid
r	Cell of lattice
t	Time
TG	Total size of lattice grid
u	Phenomenological constant
v	Phenomenological constant
Y_R_A	Y coordinate of particle A in reference time-step
Y_C_A	Y coordinate of particle A in current time-step
X_R_A	X coordinate of particle A in reference time-step
X_C_A	X coordinate of particle A in current time-step

Greek Letters

ψ	Order parameter
ϕ	Local volume fraction
γ	Shear flow
τ	Temperature-like parameter
Δt	Time-step
$\tilde{s}(f)$	Empirical fitting function

CUDA Terminology

Block	A collection of number of threads mapped to a streaming multi-processor.
Device	GPU or device computes a large amount of data parallelism.
Global Memory	Known as device RAM, it is the biggest memory on the GPU.
Grid	A collection of several blocks.
Host	CPU or host executes low volume or non-parallelised data.
Kernel	The parallel portion of a program that executes on the device.
Latency	It is the time needed to process an operation.
Occupancy	The ratio of the number of active warps to the maximum active warps.
Shared Memory	On chip memory with higher bandwidth and lower latency.
SMXs	A group of Streaming Multi-processors of GPU.
Throughput	How many operations can be processed per second in each SMX.

Abbreviations

AoS	Array of Structures
AVX	Advanced Vector eXtensions
CDS	Cell Dynamic Simulation
COM	Centre of Mass
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Device Random Access Memory
FLOP/s	Floating-Point Operations per second
GFLOP/s	10^9 FLOP/s
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Unit
HPC	High Performance Computing
HT	Hyper-Threading
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
PBCs	Periodic Boundary Conditions
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SMP	Symmetric Multi-processor
SMT	Simultaneous Multi-Threading
SoA	Structure of Arrays
SSE	Streaming SIMD Extensions
TFLOP/s	10^{12} FLOP/s
UMA	Uniform Memory Access

CHAPTER ONE

1 Introduction

By modelling nano and macro-scale structures using computational methods relational gap between the real world and the world of the laboratory can be bridged, providing useful insights regarding the evolution and dynamic behaviour of molecular systems. Computational simulation by modelling different experiments plays an important and essential role in today's technical, engineering and scientific research. This minimised the need for expensive, time consuming and sometimes hazardous experiments in order to obtain scientific data. In addition, computer simulation and modelling provides a corroboration and validation of theories into experimental works. Therefore, it is desirable and economically prudent to execute simulations as quickly as possible to investigate different properties of molecular systems.

In the last few decades particle tracking has been used extensively as one of the most popular ways for achieving the quantification of data and considering different properties of particles [1], including in: (a) tracking bacterial motion [2]; (b) studying intracellular tracking of pharmaceutical nano-carriers [3]; (c) genetic material tracking [4]; and (d) protein or lipid tracking [5]. Various studies have used different computational techniques for particle tracking for capturing the full benefits of quantitative data and understanding dynamic actions of particles [6]. Existing particle tracking methods have focused on tracking of experimental particles. The general approach involves the application image processing methods to detect experimental particles in images and transfer them to a readable form input for programs and track the detected particles. In 1995, Crocker and Grier developed an image processing algorithm for extracting quantitative data from experimental images [7]. Their technique could distinguish particles from the rest of the image. The computational technique of particle tracking velocimetry can be categorised into two levels: image relaxation and nearest neighbour search algorithm [8]. However, these two computational techniques follow the same model of particle tracking. Particle tracking model involves two main phases: image acquisition and tracking of the moving particles. Figure 1.1 shows a model of the particle tracking based on experimental work [6].

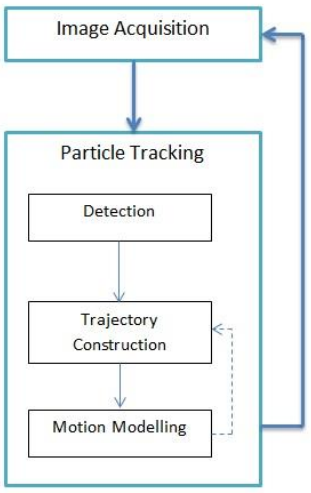


Figure 1.1: Particle tracking model based on experimental work [6].

In most particle tracking models, the data travel in one direction, from the detection part to the trajectory part, although in some particle tracking models the detection and trajectory parts are complementary and coupled to each other [9, 10]. In both situations, the motion modelling part aids the particle tracking phase by decreasing the vagueness and ambiguity of detected particles and global trajectory construction between frames. Having considered different methods for particle tracking in experimental work, it is essential to develop a method that can be used for uniform computational data. Computational data coupled with quantitative analysis and mathematical modelling helps to shed further light on a broad range of scientific issues [11].

A major challenge for particle tracking based on the cell dynamic simulation is the expense of computational works and long time frames involved, due to two fundamental constraints: time steps and experimental scale size. These limitations impact directly on the simulation results. To overcome these issues, a new parallel computational model is required. Interestingly, parallel programming model is the only method that can really improve and enhance the performance of applications and solve the problems identified. Graphics Processing Unit (GPU) has become the popular in recent years due to its broad suitability in different research fields for improving performance in various applications. Originally designed to process graphical data, GPUs were, later developed further for general purpose computing on graphics processing unit (GPGPU) allowing the operation of GPUs to process and compute non-graphical data. The application was extended to different fields of parallel programming and computational physics [12, 13], including calculations of long and short range order of nanostructure for diblock copolymers using CDS method.

Parallel processing can be executed either on Central Processing Unit (CPU) or GPU, but recently because of the performance improvements in the latter and the capability for processing graphic and non-graphical data, there has been increasing demand for executing parallel processing on GPUs [14]. GPUs produce a huge number of polygons at high speed to display photos. In addition, GPUs' floating point arithmetic is advanced and they can execute several computations efficiently rather than simply generating polygons [15]. An example of the combination between computational physics and parallel processing using GPUs involves solving the time-dependent Schrödinger equation for coherent electron transport in quasi two-dimensional electron gases. The numerical analysis of a Schrödinger equation is processed in about 280 seconds on a CPU (Intel Xenon CPU E5420 @ 2.50GHz), in comparison to 10 seconds for the GPU version (NVIDIA Tesla C1060 GPU). This result illustrates the significant time efficiency of GPUs [16]. As shown in Figure 1.2, GPU has more transistors than CPU [17], which increases number of processing cores and therefore computational capacity.

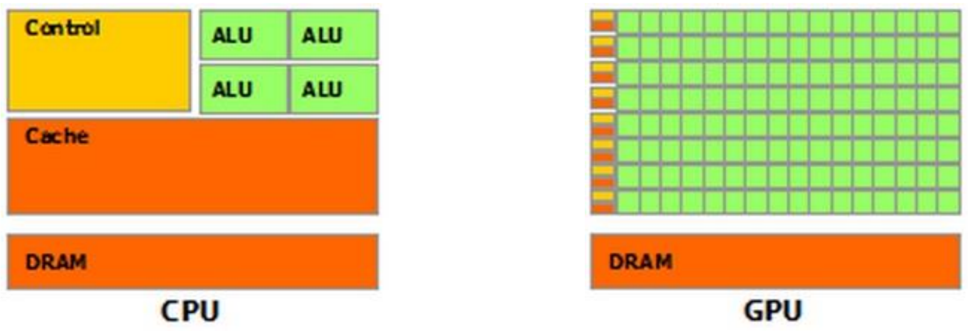


Figure 1.2: CPU and GPU architectures [17].

1.1 Motivation

Block copolymers are widely used in soft matter research for predicting the characteristics of phase separation and self-assembly into nanoscale structures. Different structures formed by polymers involve: (*i*) diblock copolymer; (*ii*) triblock copolymer; (*iii*) star triblock copolymer; and (*iv*) linear triblock copolymer [18]. Diblock copolymer is a nanoscale ordered structure based on the natural aptitude of micro-phase separation and self-assembly entailing several morphologies such as spheres, lamellar and cylinders. Diblock copolymer refers to chain molecules joined covalently and linked to a single macromolecule [19, 20]. Soft matter has been explored extensively theoretically and experimentally in many disciplines, such as chemistry and electronics, in order to understand different topological morphologies, short-range, long-range ordering of the microstructure, time evolution of ordered structures and the effects of various parameters [21, 22]. Experimentally evaluations of these issues are very costly, time consuming and problematic. Hence, a new computational technique is needed to model such a system and improve understanding of the time evolution of ordered structures and the effects of different system parameters on the properties of the diblock copolymer.

The main drawback to accurate results in computer simulation is the time-consuming and expensive nature of the computations with ordinary hardware and sequential programming. Indeed, the speed of scientific advances has not kept pace with the rapid developments in computers (e.g. supercomputer), and the peak floating-point performance of superfast computers. For instance, scientific advances approach only 5-20% of maximum computational power and performance. This difference between peak performance of computer and scientific programs leaves room for further study to enhancement of performance in different aspects significantly [23, 24]. Such investigations directly influence the cost and accuracy. Considerations of performance differences refer to the new idea of parallel computing model, which can improve and enhance performance of applications and solve important problems in various disciplines.

Simulation methods developed so far raise a number of questions. The purposes of this study are thus to obtain an improved understanding of the dynamic behaviour of particles with respect to time evolution via the computational method of particle tracking, and to overcome resource and performance limitations by proposing a novel GPU parallel algorithm and CPU multithreading method.

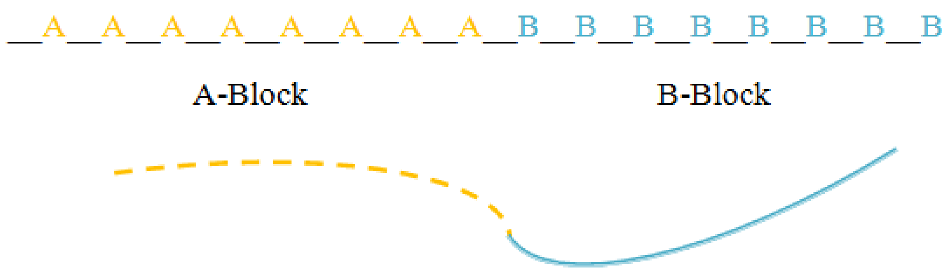


Figure 1.3: Schematic example of AB diblock copolymer.

1.2 Original Contributions in this Thesis

The work presented in this thesis makes the following novel contributions:

- Cell Dynamic Simulation (CDS) technique has been studied as a computational method to model phase separation of diblock copolymers, to investigate the effect of external fields, such as shear flow, and to simulate spherical morphology of diblock copolymer for comprehending the nontrivial behaviour of the spherical morphology of diblock copolymers.

- A new computational technique was developed during this study for detecting and tracking particles utilising results obtained from the cell dynamic simulation of a spherical phase of diblock copolymer under a shear flow. This new technique follows the same procedure for particle tracking including detection and tracking. However, detection of particles in this study is original and is not similar to any previous work, since the CDS output is numerical, and it would not be possible to utilise the same procedure in order to detect the particles while in tracking section. Other techniques have been combined such as nearest neighbour and linear assignment, with some novel methods in a unique way. However, the new method is more comprehensive and specifies not only the factors of the flow field, such as shear, but other properties such as particle number (labelling), while simultaneously and concurrently allowing the tracking of multiple particles.

- This study investigated different parallel computing architecture and programming model in both CPUs and GPUs. It comprises different aspects such as Flynn's Taxonomy model (SISD, SIMD, MISD, MIMD), restrictions and cost of parallel computing and different memory architecture of parallel computer such as distributed memory, shared memory and hybrid shared-distributed memory.

- The CDS method was implemented and optimised in C programming language and OpenMP multithreading shared memory architecture. Loop sharing method as one of the work-sharing constructs has been used to prevent race conditions, and to satisfy data dependencies different synchronisation constructs were used. In addition, the influence of multithreading to overcome time-consumption and expensive computation problems was investigated.

- A three-dimensional GPU-CUDA implementation of cell dynamic simulation for diblock copolymer based on the spatial decomposition method and block-cell link model was developed during this work. The spatial decomposition technique based on the block-cell link model was shown to be a suitable and appropriate choice for GPU-accelerated CDS simulation. By allocating enough resources for each data element, this method enhances system performance and the communication costs are reduced between each thread, ultimately decreasing the GPU's memory access time.

- A combination of different test cases and metrics according to throughput and latency of CPU and GPU for a given task to evaluate the performance and speed of parallel program were used. The performance comparison is based on a commodity NVIDIA Quadro K5000 graphics card and Intel Xeon E5-2420 CPU.

1.3 Overview of the Thesis

Following the introduction, the remainder of the thesis comprises of seven chapters.

Chapter 2 illustrates the cell dynamic simulation method, considerations of different parameters effects on CDS for spherical morphology and the simulation results for diblock copolymer sphere morphology under shear.

Chapter 3 explores the literature relevant to particle detection and tracking. It also presents a novel computational technique for particle detection and tracking and demonstrates dynamic movement and behaviour of one and more particles concurrently based on the novel method.

Chapter 4 reviews the concepts, performance analysis and computational throughput related to the areas of high performance computing and performance engineering. It specifically considers these techniques for evaluation between multi-core and many-core implementations in this study.

Chapter 5 details and reviews the concepts of past and present state-of-the-art in parallel software, hardware and computing, and addressed different parallel terminologies and programming models for different parallel memory architectures.

Chapter 6 details the evolution of graphics processing unit and investigates the GPU as many-core accelerator by considering different aspects of memory architecture and CUDA programming model.

Chapter 7 presents optimisation studies such as utilisation of SIMD, vectorisation and memory access patterns for performance improvement of the CDS baseline code. It proposes a new hybrid decomposition algorithm for multi-threading implementation on CPU and the results based on a new algorithm. The last section of chapter seven illustrates and details the algorithm, optimisation, and validation of CDS simulation scheme on GPU many-core hardware.

Chapter 8 concludes and summarises the work presented in this study, elucidates the limitations and provides scope for future research work.

CHAPTER TWO

2 Cell Dynamic Simulation

Over recent decades, numerous techniques have been used for modelling diblock copolymer, and making closer relationships between the real world and the laboratory. Based on the specific application, some of these techniques are flexible and scalable in terms of system parameters, however they lack preciseness. Other techniques such as self-consistent field theory and theoretically informed coarse-grained are more precise but not readily scalable [20]. Hence, a method is required that takes into account both accuracy and speed and makes for a closer relationship between the real world and the laboratory by modelling the behaviour of diblock copolymer on a large scale and prevent the size effect problem. To this end, CDS was identified as a suitable approach to compute and define mesoscopic self-assembled structure of diblock copolymers [25, 26].

CDS is a promising method and good example of a cellular automation in interface dynamics in phase-separating domain [27, 28]. It has been used in other systems and applications to model phase-separating dynamic, including micro-emulsions [29, 30]; cross linked polymer blends [31]; and binary blends containing surfactants [32]. The CDS equations model suggests that the CDS is a coarse-grained discretisation scheme. This scheme refers to the Ginzburg-Landau (TDGL) equation and Cahn-Hilliard Cook (CHC) equation which define all the simulation parameters for diblock copolymer and govern all differential equation for the time-evolution of order parameters, as explained in the following section. The main advantage of CDS technique compared to other molecular simulation methods is coarse-grained discretisation [33]. This chapter focuses on the CDS theory, considers the concepts of immiscible blends, and demonstrates simulation results for spherical morphology of diblock copolymer. An investigation of the external effects such as shear flow for three-dimensional structure of sphere forming is also presented.

2.1 Benchmark Description

This section describes the concepts of immiscible blend between two polymers and their morphology. The idea of mixing two polymers to produce a material with improved properties is well established, but it is nevertheless difficult to achieve practically with some polymers. The difficulty of mixable polymers is illustrated by the chicken soup scenario where chicken is one polymer and water is the other polymer. This situation, involves two phases: one phase is water and the other is chicken which is insoluble in water. The resultant mixture is thus phase-separated [34, 35]. When phase-separation of two blend polymers becomes beneficial and useful material, it is known as an immiscible blend. However, considering immiscible blend is not straight forward and needs a deep understanding of polymer structure. To address this issue, it is necessary to consider and investigate the physical structure at the nanoscale size. Consequently, physical systems that impulsively form different structures under various conditions are essential. To this end, block copolymers due to their ability to self-assemble into various nanostructures are one of the most valuable and popular materials.

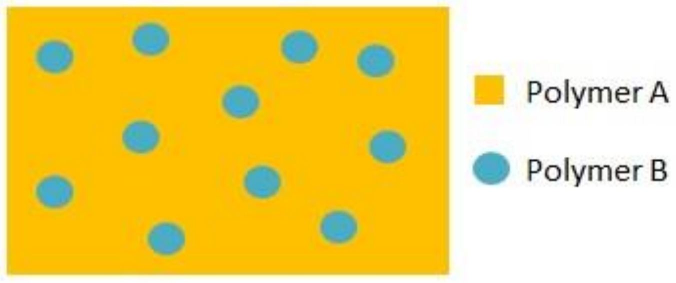


Figure 2.1: Immiscible blends of AB diblock copolymer.

In Figure 2.1, polymers A and B are mixed together but did not form a *blend*. Instead, polymer B split from polymer A and made spherical spots/forms as an immiscible blend. In real applications, the mixture of two polymers (e.g. polystyrene and polybutadiene) can make the spherical phase-separation of immiscible blends which can be seen by an electron microscope. Although other types of immiscible blends exist, such as lamellar phase separation, consideration of other types of phase separation is beyond the scope of this study as we only investigate the dynamic behaviour of spherical morphology of diblock copolymers.

The concept of morphology refers to the form and arrangement of two phases in immiscible blend. The most important parameters affecting the morphology of an immiscible blend are: (*i*) the volume of two polymers; (*ii*) temperature; and (*iii*) external fields, such as shear flow or electric field. For instance, if the volume of polymer A is greater than polymer B, polymer B will be divided into spheres. In this situation, polymer A is the major component with polymer B being minor. Figure 2.2 illustrates the relative volume of polymer B and polymer A in the immiscible blend with respect to time evolution.

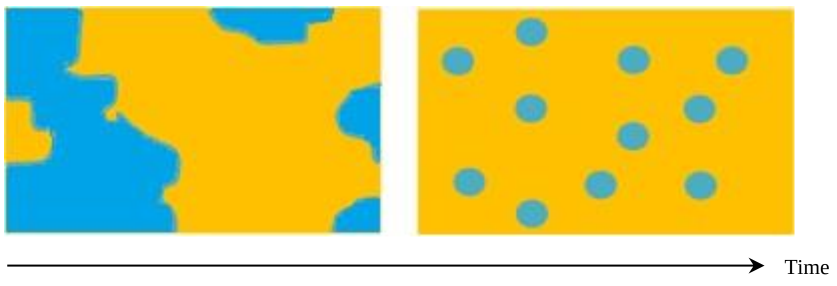


Figure 2.2: Relative volume of polymer B and polymer A in the immiscible blend.

Another important factor that affects the morphology of an immiscible blend is the external stress. Figure 2.3 shows changing morphology for two polymers A and B under flow in one direction.

Figure 2.3: Morphological change from spheres into lamellar under an external field (shear flow).

2.2 Cell Dynamic Method

Cell dynamics simulation is a cellular automation method of ordered structure shaped in diblock copolymer dissolves applied to consider the influences of simulation properties on the morphological details and kinetics of ordering structures from the preliminary disordered level [33]. The simulation properties are connected to the parameters in the Gingzburg-Landau free energy in the Cahn-Hilliard-Cook equation [18, 33]. In fact, the CDS equations are a coarse-grained discretisation offering a promising opportunity and capability to investigate the micro-phase separation details in systems that are comparable with experimental domains in terms of size.

An order parameter in cell dynamic simulation method is defined as $\psi(t,r)$. Order parameter is discretised on a lattice by getting values of $\psi(t,r)$ in cell r and time t [33, 36]. Order parameter is the difference between the local volume fractions of A and B monomers. The following equation defines the compound order parameter for AB diblock copolymer:

$$\begin{gathered}\psi = \phi_A - \phi_B + (1-2f), \\ f = N_A/(N_A+N_B).\end{gathered} \tag{2.1}$$

Here ϕ_A and ϕ_B are local volume fractions of monomer A and monomer B, and f refers to the number fraction of monomer A in a diblock copolymer which can be calculated from the second part of the equation. N_A, N_B are the total number of monomers respective to A and B blocks. The evolution and progress of order parameter for each discrete cell can be calculated by equation 2.2:

$$\psi(t+1,r) = f(\psi(t,r)). \tag{2.2}$$

In equation 2.2, $f(\psi)$ refers to the map function, which mimics the tendency of the values of order parameters not being zero during the order-disorder transition [27]. By considering coarse-grained discretisation as a main benefit of cell dynamic simulation technique, the time evolution of the order parameter can be shown by a Cahn-Hilliard-Cook (CHC) equation [18, 33]:

$$\frac{\partial\psi}{\partial t} = M\nabla^2\left(\frac{\delta F(\psi)}{\delta\psi}\right). \tag{2.3}$$

Where $F(\psi)$ is a free energy functional described by the Ginzburg-Landua (TDGL) equation. This free energy is used for defining short-range and long-range contribution of the copolymer

[33, 37]. M is a phenomenological mobility constant which is assumed to be $M = 1$ for unity throughout the simulation.

The following equations describe free energy functional, including short-range and long-range interactions terms of the copolymer [25, 33, 37].

$$F(\psi) = F_S(\psi) + F_L(\psi).$$

$$F_S(\psi) = \int dr\left[H(\psi) + \frac{D}{2}(\nabla\psi(r))^2\right],$$

$$F_L(\psi) = \frac{b}{2}\int dr\int dr' G(r-r')\psi(r)\psi(r'), \tag{2.4}$$

$$F(\psi) = \int dr\left[H(\psi) + \frac{D}{2}(\nabla\psi(r))^2\right] +$$

$$\frac{b}{2}\int dr\int dr' G(r-r')\psi(r)\psi(r').$$

Where $F_S(\psi)$ is short-range interaction term and $F_L(\psi)$ is long-range interaction term. D refers to the positive constant which acts as a diffusion coefficient. b is a value of chain-length dependence to the free energy [27]. It should be noted the term $\frac{D}{2}(\nabla\psi(r))^2$ in free energy functional is necessary to make an interface for A and B monomers. Here Green's function $G(r-r')$ for the Laplace equation satisfies $\nabla^2 G(r-r') = -\delta(r-r')$, making approximations $\psi(t+1, r) - \psi(t, r) \approx \partial\psi/\partial t$. $H(\psi)$ refers to the free energy which is the same as Landau free energy equation [25, 38, 39].

$$H(\psi) = \left[-\frac{\tau}{2} + \frac{a}{2}(1-2f)^2\right]\psi(r)^2 + \frac{v}{3}(1-2f)\psi(r)^3 + \frac{u}{4}\psi(r)^4, \tag{2.5}$$

here τ is a temperature-like variable and a, v, and u refer to the phenomenological constants [33], which can be related to molecular properties and characteristics. In fact, these are multifaceted and complex functions, and since we are considering the general phenomenology of diblock copolymer, we choose just approximate and estimated values for these constants [40, 41, 42]. According to Ohta and Kawasaki [40], $\tau = -\tau' + a(1-2f)^2$ and D can be written in terms of degree of polymerisation N and the Flory-Huggins factor χ. In fact, τ' is related to the Flory-Huggins parameter which contrariwise depends on the temperature. Therefore, the parameter τ' can be stated in terms of physical parameters:

$$\tau' = -\frac{1}{2N}\left(N\chi - \frac{\tilde{s}(f)}{4f^2(1-f)^2} \right). \tag{2.6}$$

Equation 2.6 presents the physical parameter of temperature, where $\tilde{s}(f)$ refers to the empirical fitting function which is considered to be of the order of 1. χ refers to the measurement of the comparative strength of the repulsion between different types of segment to the attraction between the same types of segment [40, 41]. *N* indicates the total degree of polymerisation, and can be expressed as $N = N_A + N_B$.

According to the preceding equations the numerical solution of equation 2.3 for cell dynamic simulation in a cubic lattice can be written as [27, 40, 43]:

$$\psi(t+1,n) = \psi(t,n) - \{\langle\langle\Gamma(t,n)\rangle\rangle - \Gamma(t,n) + b\psi(t,n)\}. \tag{2.7}$$

In addition, a shear flow term is added to consider the movements of diblock copolymer [27, 44]:

$$\begin{aligned}\psi(t+1,n) = \psi(t,n) - \{\langle\langle\Gamma(t,n)\rangle\rangle - \Gamma(t,n) + b\psi(t,n) + \\ \frac{1}{2}\gamma y[\psi(n_x+1,n_y,n_z,t) - \psi(n_x-1,n_y,n_z,t)]\},\end{aligned} \tag{2.8}$$

Where $\langle\langle\rangle\rangle$ refers to the discrete Laplacian for calculating the average in the neighbourhood cells excluding for the centre cell $=\langle\langle X\rangle\rangle - X$. n is the total number of grid points $n = n_x \times n_y \times n_z$. γ and y refer to shear flow rate and dimensionless coordinate. It should be noted that we have applied shear flow just along X axis, therefore $v_x = \gamma$. The function $\Gamma(t,n)$ is defined by:

$$\Gamma(t,n) = f(\psi(t,n)) - \psi(t,n) + D[\langle\langle\psi(t,n)\rangle\rangle - \psi(t,n)], \tag{2.9}$$

As mentioned earlier, $\langle\langle X\rangle\rangle - X$ refers to isotropised discrete Laplacian with a number of X [27, 18]. Hence, for a three-dimensional grid (cubic) lattice can be calculated as [33, 45]:

$$\langle\langle\psi(t,r)\rangle\rangle = \frac{6}{80}\sum_{NN}\psi(t,r) + \frac{3}{80}\sum_{NNN}\psi(t,r) + \frac{1}{80}\sum_{NNNN}\psi(t,r). \tag{2.10}$$

Where *NN* refers to the nearest neighbours, *NNN* next-nearest neighbours and *NNNN* next-next-nearest neighbours of a grid point (i, j, k).

Finally, the equations mentioned enable the identification of the map function [25, 33, 37] which is related to the functional derivative of free energy in equation 2.5.

$$f(\psi) = \left[1 + \tau - a(1-2f)^2\right] \psi - v(1-2f)\psi^2 - u\psi^3. \tag{2.10}$$

2.3 Simulation Results

This section presents the simulation results of spherical phase diblock copolymer under shear flow based on the cell dynamic method. Cell dynamic method in this context is divided into five main calculations: (*i*) calculations of periodic boundary conditions; (*ii*) calculations of first isotropised discrete Laplacian; (*iii*) calculations of map function and free energy functional; (*iv*) calculations of second isotropised discrete Laplacian of the free energy functional; and (v) the time evolution of the order parameters calculations. In the first step, calculations of periodic boundary conditions are divided into three sub-levels for calculating PBCs for each dimension (x, y, z) of the system respectively. In the second step, calculation of first isotropised discrete Laplacian is taken into account, which is $\langle\langle\Psi\rangle\rangle - \psi$ in free energy functional term. In the third step, calculation of map function and free energy functional are considered: $f(\psi) = \left[1 + \tau - a(1-2f)^2\right] \psi - v(1-2f)\psi^2 - u\psi^3$ and $\Gamma(t,n) = f(\psi(t,n)) - \psi(t,n) + D[\langle\langle\Psi(t,n)\rangle\rangle - \Psi(t,n)]$.

After calculating map function and free energy functional, calculation of second isotropic Laplacian operator of the free energy functional is the fourth step considered, which is $\langle\langle\Gamma(t,n)\rangle\rangle - \Gamma(t,n)$ in time evolution of order parameters equation. The last step refers to the calculations of time evolution of the order parameters based on the definition of the fields in a cubic lattice/system, together with a precise and appropriate discretisation of the isotropic Laplacian which is $\psi(t+1,n) = \psi(t,n) - \{\langle\langle\Gamma(t,n)\rangle\rangle - \Gamma(t,n) + b\psi(t,n)\}$.

Some parts of the implementation source-code of the cell dynamics simulation method is presented in Appendix A.

The main practical barrier encountered in this process is the time-consuming and expensive nature of such computations, as discussed in chapter seven. Table 2.1 presents the system parameters used to generate spherical phase of diblock copolymer in cell dynamic method [18]. The simulations were executed on a $64 \times 64 \times 64$ lattice system size for up to 1000,000 time-steps without a shear and 300,000 time-steps more after applying a shear to achieve stable and perfect system. An initial random disordered ψ is between $[-0.9, 0.9]$. It should be noted that there is a difference between an equilibrium and stable system: a system has equilibrium at certain time-steps (no-more changes after that specific time-step), but it still has defects; while a stable system is a perfect one without any defects.

CDS system parameters	D	a	b	u	v	f	τ
Spherical system	0.5	1.5	0.01	0.38	2.3	0.4	0.20

Table 2.1: System parameters used in cell dynamic method for spherical morphology.

Figure 2.4 illustrates the phase diagram of diblock copolymer for spherical morphology with different shear rate, and different time-step. It can be seen that at lower shear rate (0.0005) the system is completely spherical, with a hexagonal order. At 0.001 shear rate the system is spherical but not completely ordered, and at higher shear rate (0.005) the spheres are lengthened to ellipsoids and cylinders (particles are mixing together). Additional increase in shear transforms the entire system from spheres to cylinders. Increasing the shear rate has a direct impact on the sphere to cylinder transition.

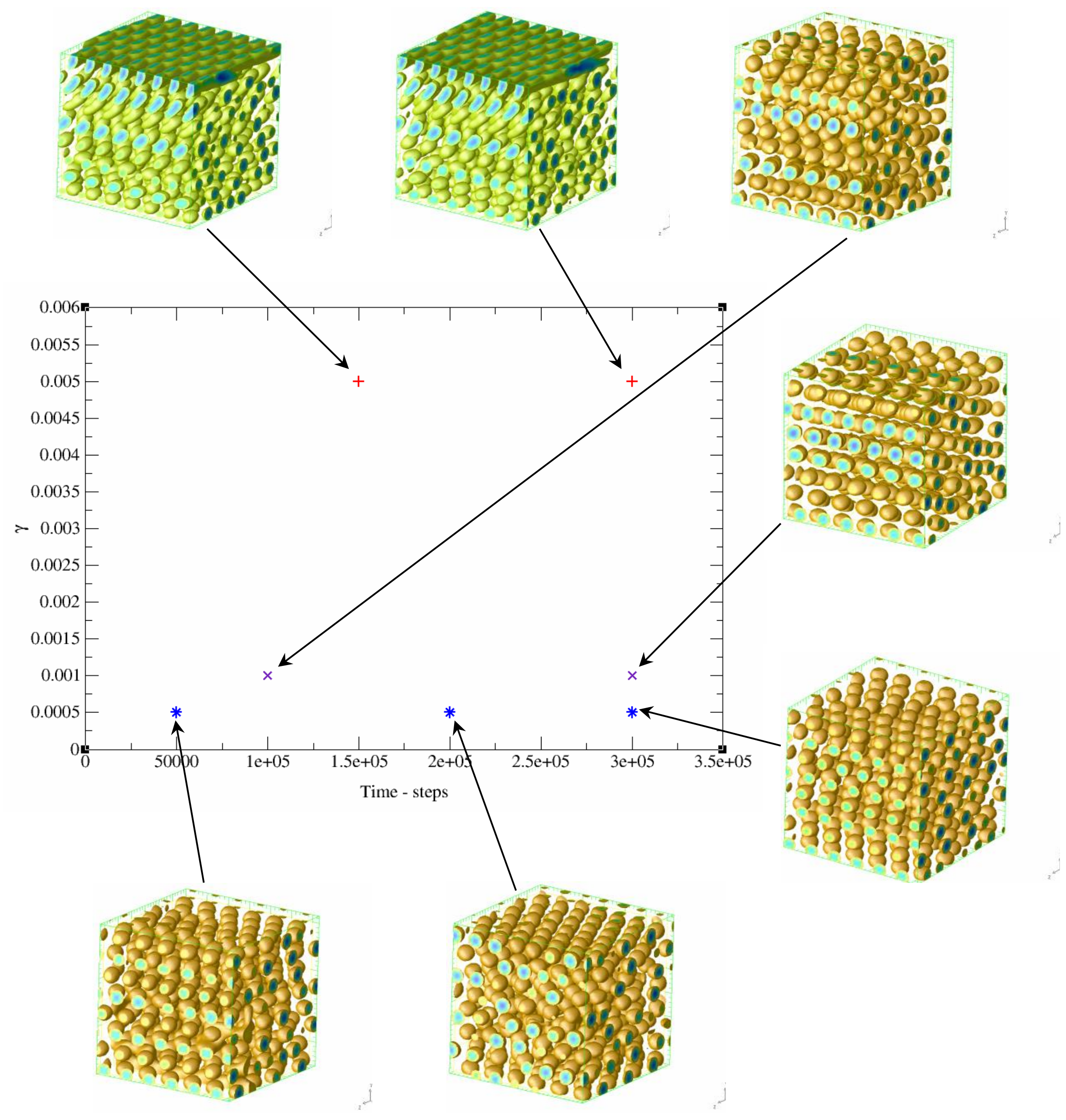


Figure 2.4: Phase diagram of diblock copolymer morphologies under different shear flow (γ) and evolution of time steps: $*)$ Perfect Spheres, $\times)$ Spheres, $+)$ ellipsoid and Cylinder.

The following figures present the kinetics of transition of spherical diblock copolymer with the same system parameters as shown in table 2.1 except the shear rate is 0.0003. By considering the Figure 2.4 and the following figures we can comprehend that when the shear flow is between 0.001 and 0.0001 the system obtains the spherical morphology with a hexagonal order (stable and perfect system).

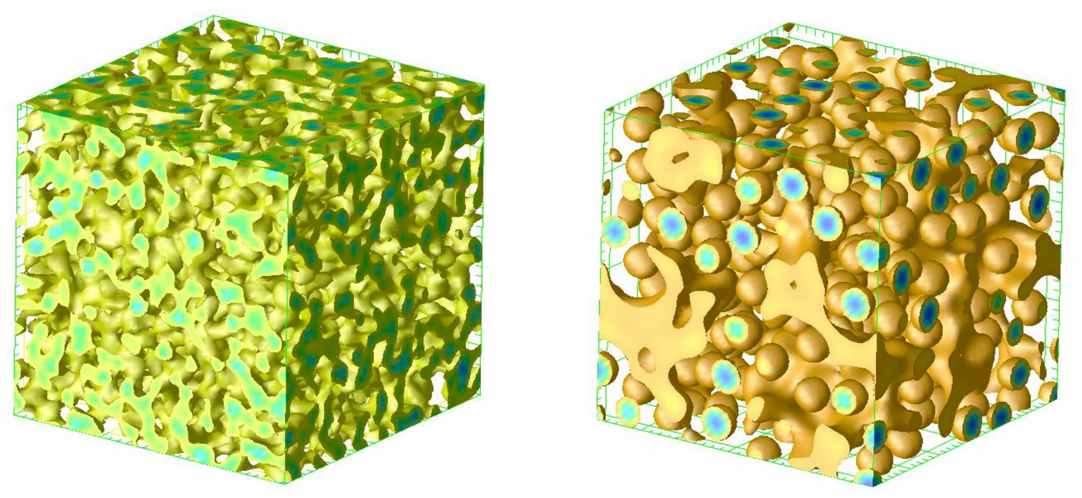


Figure 2.5: Spherical morphology produces by cell dynamic simulation method from initial stage (left) to last stage (right) without shear flow.

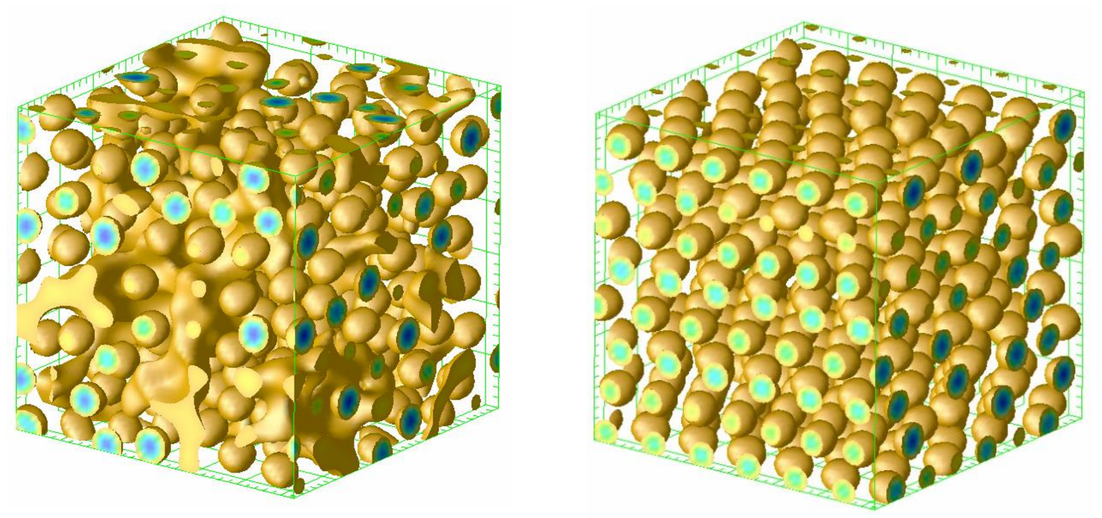


Figure 2.6: Spherical phase of block copolymer under a shear flow from initial time-step (left) to last time-step (right).

Shear γ	Morphology
0.0001	Spheres ($*$)
0.0003	Spheres ($*$)
0.0005	Spheres ($*$)
0.001	Spheres ($\times$)
0.005	Spheres & Cylinders (+)
0.008	Cylinders

Table 2.2: Summary of morphology results for diblock copolymer system under different shear flows.

Investigation of perfectly ordered structure of diblock copolymer system for spherical morphology under different shear rates and time steps yielded the result shown in Figures 2.7 and 2.8, showing the polygon structure of diblock copolymer from right side view, and Figure 2.9, which presents the transition from pentagonal to hexagonal structure in yz - {011} plane (right side view).

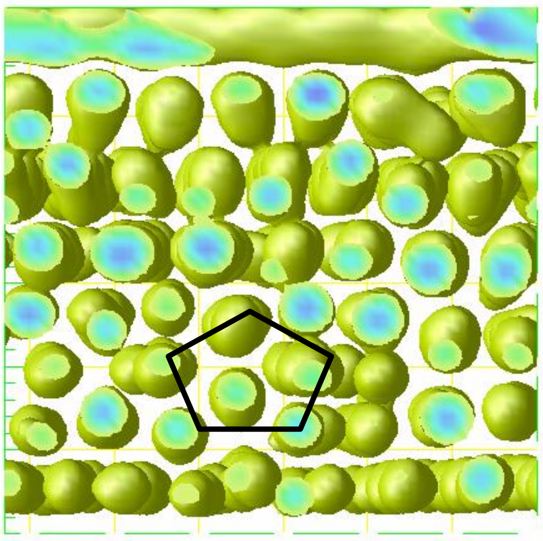


Figure 2.7: Polygon structure of diblock copolymer in time-step 300 with 0.005 shear flow rate.

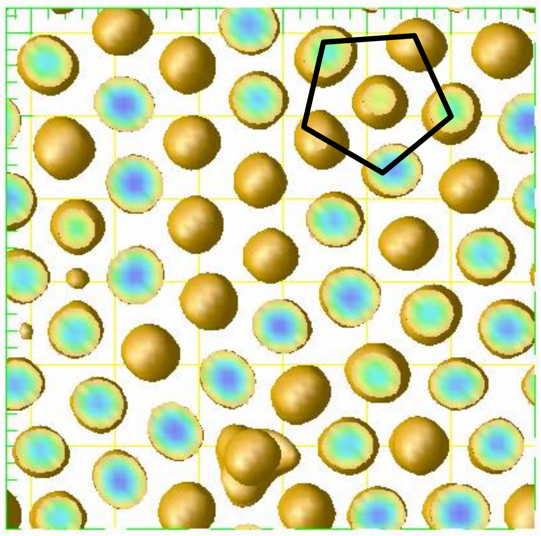


Figure 2.8: Polygon and pentagonal structures of diblock copolymer in time-step 300 with 0.001 shear rate.

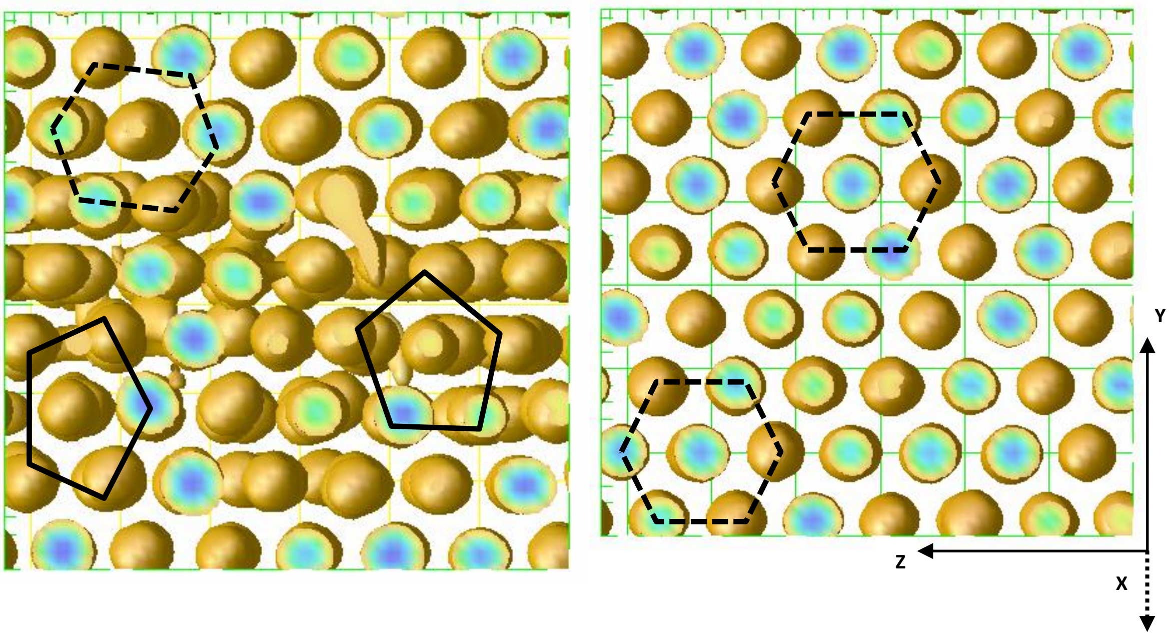


Figure 2.9: Pentagonal and hexagonal structures of diblock copolymer with 0.0003 shear rate.

In Figure 2.9, the right image shows the pentagonal and hexagonal structures in time-step 100 and the image on the left illustrates the perfect hexagonal order structure in time-step 300. The solid lines refer to the pentagonal order and the dash lines indicate perfect hexagonal order. The dot arrow is the direction of shear.

2.4 Summary

The theory of cell dynamic simulation along with a consideration of the Ginzburg-Landau (TDGL) and Cahn-Hilliard Cook (CHC) equations for free energy and time evolution have been presented in this chapter. Examples of the use of cell dynamic simulation as a powerful scheme in other applications and systems were included, such as micro-emulsions [30], cross-linked polymer blends [31], and binary blends containing surfactants [32]. However, in this chapter the main focus was on the application of the CDS scheme to model micro-phase separated structures in diblock copolymer and to simulate the spherical phase diblock copolymer under shear flow. The benefits of CDS compared to other molecular dynamics methods have been highlighted. The concept of immiscible blend and morphology, and how CDS method as a cellular automation can help to understand the dynamic behaviour of copolymers in immiscible blend situations, has been discussed briefly. To this end, the spherical morphology of diblock copolymer under a shear flow was studied in order to obtain a better understanding the effect of external fields and also to comprehend the dynamic behaviour of the spherical morphology of diblock copolymers in different time-steps. Dynamic density functional theory [38, 46], molecular dynamics simulation method [47], or particle-based Langevin dynamics simulation method [48] have also been mentioned in relation to the spherical morphology of diblock copolymer under shear flow. However, these were limited to small domains or cubic system sizes and did not allow for a greater insight into the kinetics of spherical morphology of diblock copolymer on the scale of nanostructures and particle tracking. The results obtained for cell dynamics simulation for diblock copolymers under a shear flow in large systems can provide a benchmark for subsequent investigations.

CHAPTER THREE

3 Multiple Particle Tracking

Particle tracking methods have been studied to comprehend the mobile behaviour of particles. During the last twenty years particle tracking has been used widely as one of the most well-known ways for achieving the quantitative data and considering different properties of particles [1], including: (*i*) tracking bacterial motion; (*ii*) studying intracellular tracking of pharmaceutical nano-carriers; (*iii*) genetic material tracking; and (*iv*) protein or lipid tracking. Various scientific contexts, according to their aims, have applied different computational techniques of particle tracking to capture the full benefits of quantitative data and understanding dynamic actions of particles [6].

Existing particle tracking methods have considered tracking experimental particles. The general approach to this point is to apply image processing methods to detect experimental particles in images and transfer them into a readable form input for programs and track the detected particles. In some image processing techniques, light points/pixels with high intensity and dark points with low intensity are coded differently to distinguish between the particles and background, and to reduce the number of irrelevant pixels to a minimum. Figure 3.1 presents the percentage of expanding interest in particle and cell tracking between 1970- 2010. It should be noted that the rate of percentage is measured based on the number of publications per year for the indicated mixtures of words in the title and/or abstracts in the PubMed database [49], (National Library of Medicine and National Institutes of Health, USA).

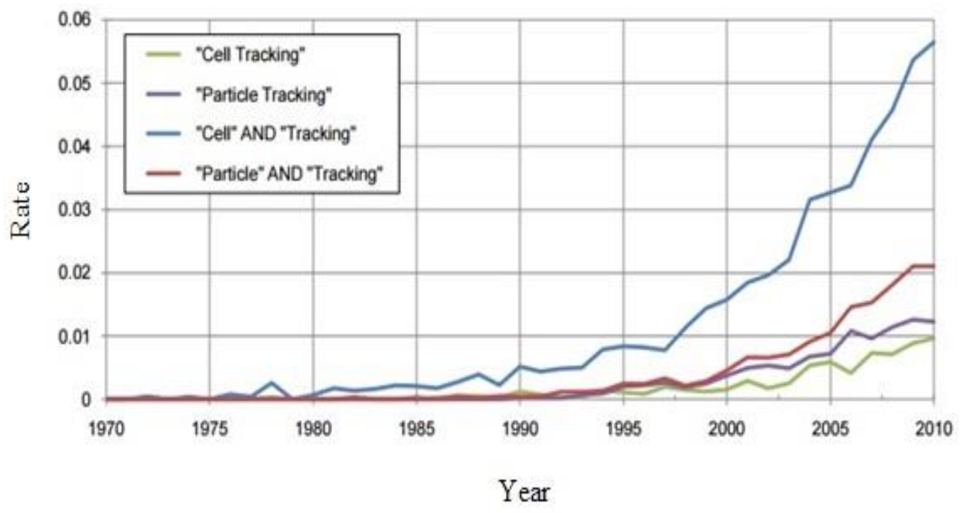


Figure 3.1: Growing interest in particle tracking and cell tracking rates (%) in different research fields over three decades [49].

Particle imaging velocimetry and particle tracking velocimetry are the two most famous computational techniques used for particle tracking [50]. Standard processes for particle imaging velocimetry are divided into two levels: algorithms for detecting particles on images, which is mainly related to standard cross-correlation method [51]; and techniques for tracking the particles based on each frame of a time lapse sequence of images. Typically, the first part of experimental particle tracking model refers to the image processing technique used to optimise and distinguish between particles and background, and any acquisition failure of optimised images will have a direct effect on particle tracking.

In 1995 Crocker and Grier developed an image processing algorithm for extracting quantitative data from experimental images [7]. This algorithm distinguishes the particles from the rest of the image. Particle tracking velocimetry as a computational technique can be categorised: image relaxation and nearest neighbour search algorithm [52]. However, these two computational techniques followed the same model of particle tracking. Particle tracking model involves two main phases: the image acquisition and the tracking the moving particles.

The main purpose of particle detection is to understand numerical representation of the positions and different components of image features [10]. In image acquisition stage, all images with their local intensity are considered with different neighbours who have different intensity levels. In this part, cross-correlation technique performs well for different particle intensities. According to Kean and Adrian (1995), it is necessary to consider the ratio between image size and particle displacement. In fact, the versatility in choosing the image size and location of successive particles (displacement) allows a better spatial resolution with more efficient and effective matching/paring of particle images. Therefore, by the optimal implementation of the cross-correlation method (considering appropriate ratio between image size and particle displacement) no pair of particles will be lost and the spatial resolution can be enhanced [51]. Alternatively, local nearest neighbour can be used for detecting particles when the ratio between particle movement/displacement and mean nearest neighbour is less than a specific number, which depends on the number of detected points from one frame to another of images. Based on the local nearest method, Crocker and Grier developed a particle tracking tool which is one of the most well-known and widespread tracking packages [53]. Local nearest method executes well when the ratio between particle movement and mean nearest neighbour is not large. If the numbers of candidate particle are high enough or insufficient then the local nearest neighbour method will break down. Candidate particle refers to the concerned particle in different time-lapse. To overcome of this issue, multiple hypotheses tracking (MHT) was introduced [54]. In multiple hypothesis tracking, the locations of particles are specified in every frame, and according to the velocity of particles the next position of particle is predicted. In this method all particle routes as well as the whole expected particle behaviour are created using the trajectories of all particles. Hence, by considering multiple hypothesis tracking, acceptable tracking probabilities can be maintained for particle tracking in experimental works [54].

Recently the global nearest neighbour method has been used for particle tracking for cell biology applications [50]. This method applies an accurate mathematical model, the linear assignment problem [55], to deliver efficient and effective answers to the issues mentioned previously. The algorithm first considers the detected particles through a time-lapse image sequence and then makes the connection between all detected particles in each frame of images consecutively by using the global nearest neighbour technique. In contrast to the local nearest neighbour method, the global nearest neighbour obtains accurate solutions by decreasing the complexity of computational achievements of particle tracking and solving the frame-to-frame correspondence issues in the whole trajectory. The key point of global nearest neighbour method is motion prediction under high density circumstances. In the model, motion level by predicting particle position from primary frame to the target frame reduces the computational complexity, which helps the trajectory construction between frames. Based on the global nearest neighbour there are different types of possible ways to achieve particle motion between frames, such as formulating unique motion model for each particle [56], or predicting general particle motion between frames [57].

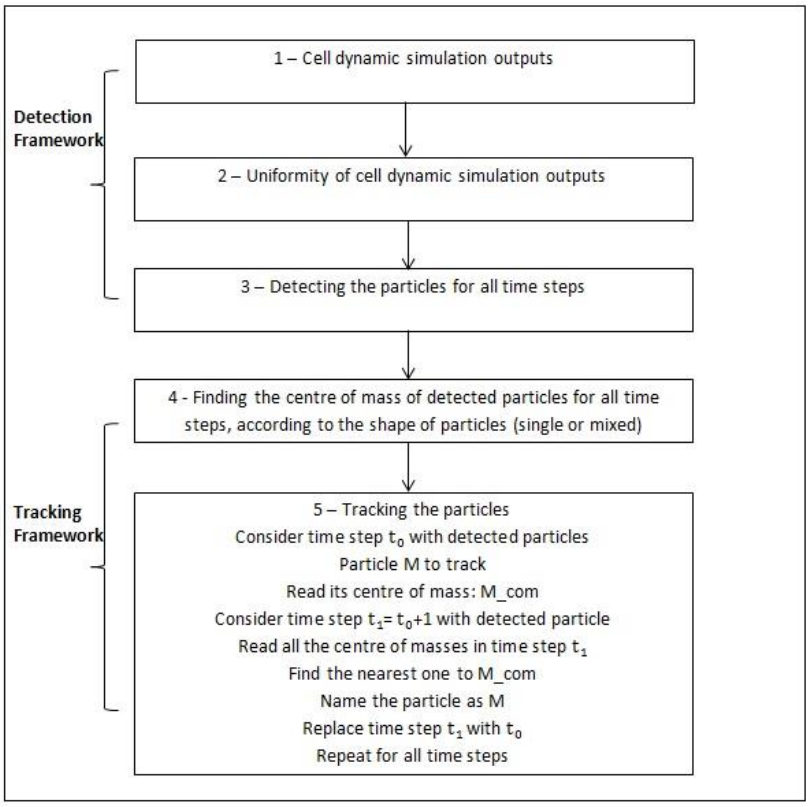
The direct outcomes after applying particle tracking on different context of sciences refers to the data in text files format which contains a sequence of coordinates showing the location of tracked particle at each time-lapse [49]. Computational analyses help to acknowledge and understand different aspects of these types of data. Motility analysis is one of the primary computational analysis which helps to rebuild the trajectories of detected particles from a sequence of coordinates, calculating the entire distance travelled by the considered particle, and finding the distance from the start to end point of detected particle [58]. The other well-known computational analysis for particle tracking is velocity analysis. Velocity (i.e. the speed of particle) is the rate of displacement which can be calculated as the movement of particle from one frame to the next frame, divided by the time interval. According to Qian and Bahnson, velocity analysis is suitable for making speed histograms due to a focus on the statistics of the dynamics [58]. The final analysis is morphology analysis, which is mainly concerned with shape evolution of particle in different time steps. Based on the morphology analysis of particles, different types of geometric information can be measured and calculated, such as measurement of size (surface area) and orientation invariant (sphericity, ellipticity) [59].

A consideration of the various methods for particle tracking in experimental works provided the impetus to devise a new method which can consider uniform computational data. Computational data is considered alongside quantitative analysis and mathematical models to solve scientific issues [11]. Usually, computer simulation and other types of computation from numerical analysis can be used to obtain the aims of computational data. This study considers computational data whereby particle movement is simulated using cell dynamic simulation.

The new particle detection method is utilised in the output of the Cell Dynamic Simulation program as computational data used to describe the morphology of diblock copolymer sphere under shear [37, 33]. The method of particle detection is novel, since the CDS output is numerical, and it would not be possible to do image processing in order to detect the particles while in tracking section. It should be noted that the fundamental idea of the new particle tracking is the same as shown in Figure 1.1, which is first detection and then tacking. Some previous techniques have been combined, such as nearest neighbour and linear assignment, and some new methods have been developed, as explained in the following chapter specifying not only the factors of the flow field, such as shear, but other properties such as particle number, simultaneously and concurrently allowing the tracking of multiple particles.

3.1 Method and Design

As mentioned earlier, in this study the cell dynamic simulation method was used to describe the spherical morphology of diblock copolymer under a shear flow. The new particle detection and tracking method will use the output of CDS calculation to achieve the aim of this study. The conceptual model of the system is shown in Figure 3.2.



Detection Framework
1 – Cell dynamic simulation outputs
2 – Uniformity of cell dynamic simulation outputs
3 – Detecting the particles for all time steps
Tracking Framework
4 - Finding the centre of mass of detected particles for all time steps, according to the shape of particles (single or mixed)
5 – Tracking the particles
Consider time step t_0 with detected particles
Particle M to track
Read its centre of mass: M_com
Consider time step $t_1 = t_0+1$ with detected particle
Read all the centre of masses in time step t_1
Find the nearest one to M_com
Name the particle as M
Replace time step t_1 with t_0
Repeat for all time steps

Figure 3.2: Conceptual model developed in this study for particle detecting and tracking.

The first step is the output of cell dynamic simulation of a block copolymer under shear, which provides details of equilibrium spherical morphology of a diblock copolymer. The second step refers to the uniformisation and segmentation of equilibrium morphology (*AB* diblock copolymer) by converting them to one and zero. In a lattice system, all domains forming particles will be distinguished by the value of one, and all grid points with the value of zero show the surrounding polymer. In fact, due to the difference in the volume fraction between components *A* and *B* in space, it is necessary to segment it in order to make the output usable for the next step. This segmentation allows recognition of polymer domains of polymer with a certain value. In order to undertake the segmentation, the value of all grid points/cells with negative value is changed to zero, while the value of grid points in the positive range is replaced by one.

The third step is an important part of the model and contains two sub-levels: nearest neighbour searching method for detecting the particles; and counting the number of detected particles. After detecting the particles based on the neighbour searching, calculating centres of mass for each particle is essential. In the fourth step all single and mixed particles are distinguished and their centres of mass are calculated. It should be noted that mixed particles can also have different shapes (oblique and horizontal). The final step of the investigation refers to the time evolution of particles and tracking particles. This is achieved by considering the centre of mass (COM) while tracking the trajectory of considered particles in different time steps.

3.2 Computational Algorithm

In order to detect particles correctly, a sequence of coordinates referring to the position of each particle is important, but this does not help the concept of detection. Therefore, it is necessary to focus on the computation aspects of these coordinates. The computation aspects refer to the quantitative measures from the coordinates, which involves neighbour searching and statistical study. Table 3.1 presents the computational optimisation algorithm for detecting and tracking particles. The whole of the pseudo-code of computational algorithm is presented in Appendix B.

```
Part I - Particle detection

1:  Step A – applying the detection method and Periodic Boundary conditions (PBCs)
2:    Find the first particle
3:    Apply the boundary conditions
4:    Start to search the nearest neighbours
5:    If neighbours are equal "1" then
6:    Change the current coordinates to the new coordinates
7:  Step B – counting the number of particles
8:    Count the number of particles
```

```
9:   labelling  particles

Part II - Calculating centre of mass

10:  **Step A** – initializing the values
11:   For particles which are in the left width boundary or
12:    Right width boundary, or up length boundary, or down length boundary
13:  **Step B** – finding the single particles
14:   Single particles (if the number of grid points for detected particle are less than 55)
        (55 grid points is an average size of spherical detected particle
        and can be changed based on the lattice system size.)
15:    Consider boundary conditions for them (left, right, up, down) and
16:    If particle is not on any boundaries.
17:  **Step C** – finding the mixed particles
18:   Mixed particles (if the number of grid points for detected particle is more than 55)
19:   Consider boundary conditions for them (left, right, up, down) and
20:   If particle is not on any boundaries.
21:  **Step D** – calculating and writing the centre of mass
22:   For all particles ( single and mixed )

Part III - Tracking the next particle

23:  **Step A** – Find the nearest particle and name the new particle as an initial particle
24:   Finding the nearest particle base on the finding the nearest centre of mass and statistical
study
25:  **Step B** - Considering periodic boundary conditions (PBCs)
26:   Periodic boundary conditions apply in five different situations.
27:    1 - PBC in width boundary
28:    2 - PBC in length boundary down -middle
29:    3 - PBC in length boundary up -middle
30:    4 - PBC in length and width boundary up
31:    5 - PBC in length and width boundary down
```

Table 3.1: Computational algorithm for detecting and tracking particles.

3.2.1 Particle Detection

Following the computational algorithm, the first part is detection of particles, which involves two steps. Step A contains periodic boundary conditions (PBCs) and computational methods of detecting particles. PBCs are a group of boundary conditions utilised to prevent of losing the particles on the edges of the grid box, and to simulate a big system by considering a small part of a system which will not be terminated when it will be close to the edges [6]. PBCs have been applied in four directions, with consideration of two grid points (neighbours) in each direction. For understanding the functionality of periodic boundary conditions, suppose it is a grid of the size of 128×128, without PBCs, and the particles are moving in the X axis. In this case each particle will be terminated on the last grid point (128) without having any reaction from the other side of the grid box. Therefore, it is necessary to apply periodic boundary conditions to the system to prevent such any issues. Figure 3.3 shows PBCs in 2D.

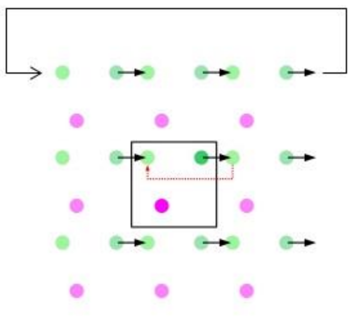


Figure 3.3: Periodic Boundary Conditions (PBCs) [60].

Since shear flow in periodic boundary conditions is very important, hence, equation 3.2 is used in cell dynamic method to satisfy boundary conditions in simulation. The reader is reminded that a shear flow is applied in the X direction. Therefore, shear expressed by:

$$v_x = \gamma y \quad v_y = v_z = 0. \tag{3.1}$$

Where $v = (v_x, v_y, v_z)$ indicates the flow. The X-axis is the shear flow direction, Y-axis refers to the velocity gradient and the Z-axis is the vorticity axis. The γ is a dimensionless shear and y indicates dimensionless coordinate.

$$\psi(x, y, z, t) = \psi(x + N_x TG + \gamma(t) N_x TG, y + N_y TG, z + N_z TG, t). \tag{3.2}$$

Here N_x, N_y, N_z are numbers of lattice points in X, Y, Z axes of coordinates and TG refers to the total size of lattice grid. The first term on the right part of equation indicates the direction of shear flow in X-axis $(x + N_x TG + \gamma(t) N_x TG)$.

As stated previously, a sequence of coordinates by itself is not helpful for detecting particles; therefore, it is important to implement computational methods. A quantitative measure is one of the computational methods [61] used in the new uniform computational program for detecting and considering time-evolution of particles. Neighbour search method has been used as a quantitative analysis/method for finding particles and reconstructing the time-lapse of the detected particles from the measured coordinates [62]. Nearest neighbours search was applied in eight directions, considering two neighbours in every check. The order of checking is: left, south, right, north, south/east, south/west, north/east, and north/west. In step B of particle detection, the number of detected particles will be counted, and each particle will be identified and labelled with unique number.

Figure 3.4 presents the sample of neighbours searching in the grid. Red dots show the searching points which start from central point (blue dot) in eight different directions. This part is based on the outputs of segmentation step which divides the domains of polymer into two parts. All particles will be distinguished by value of one and all grid points with value of zero show the surrounding polymer.

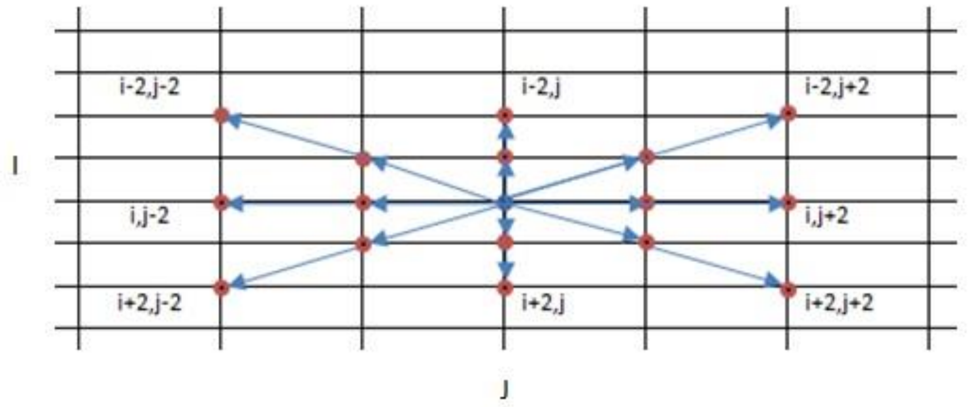


Figure 3.4: Example of neighbouring search.

3.2.2 Calculation of Centre of Mass

When tracking particles, the COM of each particle is used to describe its position. Therefore, finding the accurate COM of each particle is necessary. Current methods for finding the COM of particles were developed according to the output of the detection step. In order to accurately determine the position of each particle, particles are divided into two categories according to their size (number of detected grid points in each particle). Smaller particles (round shapes) are named as single particles and bigger particles (elongated) are mixed particles. Calculating COM of each particle depends on the shape of the detected particle.

It should be mentioned that calculating COM has two scenarios when the system is stabilised and when is not stabilised. In first situation, the particles will be fixed and there is no difference between the shapes of particles. In second situation, a more complex scenario, the numbers of particles are altering and particles have different types of shapes. Figure 3.5 shows spherical phase of diblock copolymer under a shear after system stabilisation (perfect system). The simulations were run on a cubic $128\times26\times128$ lattice for up to 1000,000 total time-steps, with 0.0003 shear rate to approach stable system.

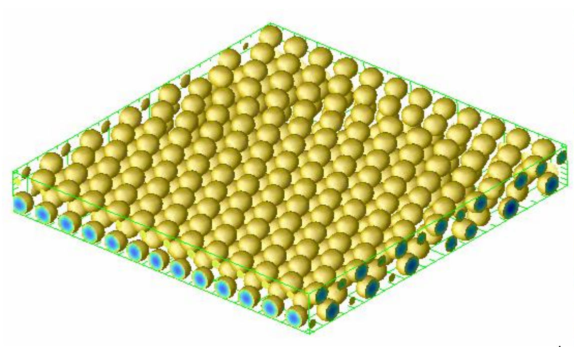
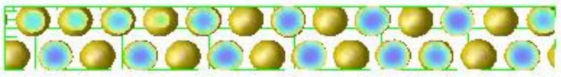
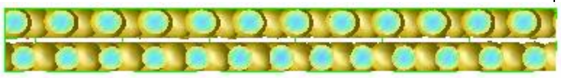
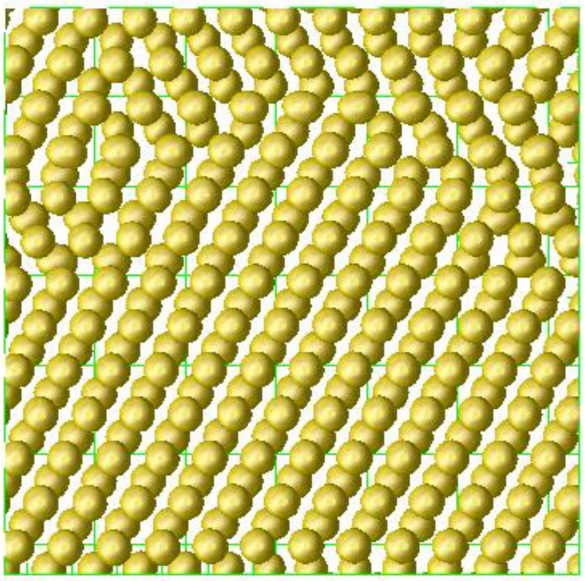
 (a)  (b)  (c)	
Bilayer spherical system – (a) diagonal view; (b) right side view in the x-direction; (c) back side view in the z-direction	Bilayer spherical system – vorticity plane top view

Figure 3.5: Perfect spherical phase of detected particles under shear flow.

Figure 3.6 illustrates an unstable system with different shapes of detected particles from top view of the system. The complete simulations were executed on a cubic $128 \times 26 \times 128$ lattice for up to 100,000 time-steps with 0.0003 shear flow rate.

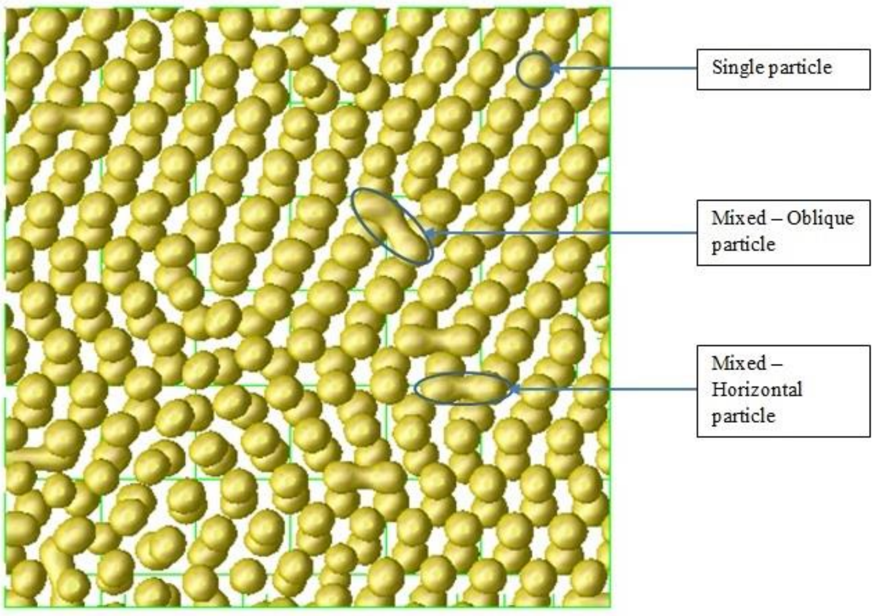


Figure 3.6: Bilayer spherical phase of detected particles.

A statistical study was undertaken on the output of the detection code, pertaining to the frequency of the number of grid points for the detected particles, which resulted in 55 grid points being chosen to distinguish between single and mixed particles. In total, 20 random particles were considered and found out that the number of grid points for detected single particles will not be more than 55 grid points. Therefore, 55 grid points was selected as a distinguish value. Based on the chosen limit, particles with less than 55 grid points were considered to be single and those with more than 55 grid points to be mixed particles. When the particles are homogeneous, the fundamental idea to find the COM of a particle is to add up the coordinates of all points in the X and Y directions separately and divide the sum by the whole number of detected grid points that belongs to the particles. It should be noted that the new particle detection and tracking method is based on the 2D results, thus there is no need to consider Z-axis.

$$X_{COM} = \frac{\sum_1^{NP} X_i}{NP}, \qquad Y_{COM} = \frac{\sum_1^{NP} Y_j}{NP}. \tag{3.3}$$

In addition, to increase the accuracy of tracking, COM of single and mixed particles were calculated in different ways. Single particles were considered to have one COM, while mixed particles have more than one COM. This is because mixed particles are likely to split into two smaller particles. Because of their bigger size and possible changes in their shapes, the movement of their COM might be bigger. Mixed particles can also have different shapes. For instance, they can be horizontal and oblique (in descending or ascending direction). These shapes are recognised by comparing the width and the length of each particle. Figure 3.7 presents an example of different shapes of mixed particle: a) oblique-ascending; b) oblique-descending; and c) horizontal.

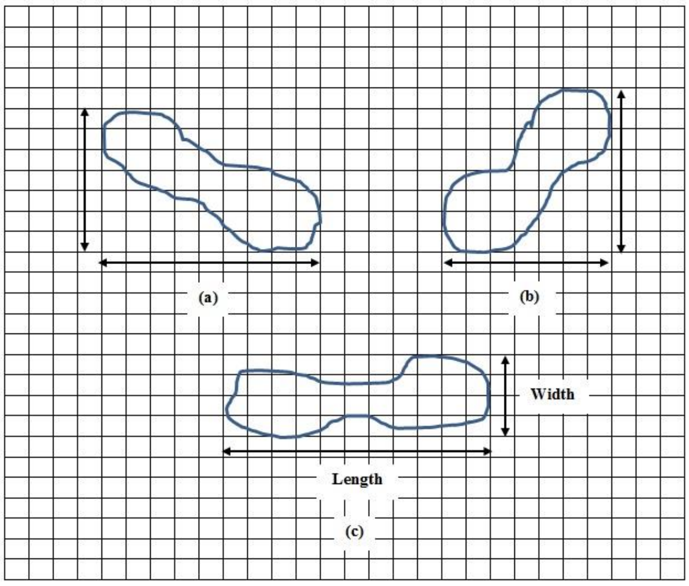


Figure 3.7: Different forms of mixed particle.

Based on the length and width of the mixed particle, if the width of mixed particle is less than the length, the shape of mixed particle will be horizontal; if the width is greater than the length, the mixed particle will be in oblique form. It should be noted that the length and width of mixed particles can be calculated by the following equation:

$$\begin{aligned} Length &= Greatest\_i - Least\_i \\ Width &= Greatest\_j - Least\_j \end{aligned} \tag{3.4}$$

Here Greatest_i and Least_i refer to the highest and lowest coordinate numbers of detected mixed particle in the X axis, while Greaatest_j and Least_j are the highest and lowest coordinate numbers of detected mixed particle in the Y axis, respectively. The following figures illustrate examples of models of the coordinates of the COMs of the mixed particle in oblique form.

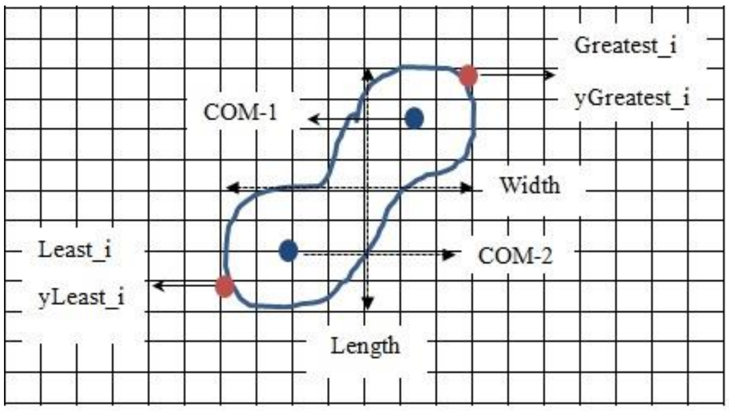


Figure 3.8: Coordinates of the COM of the mixed ascending particle.

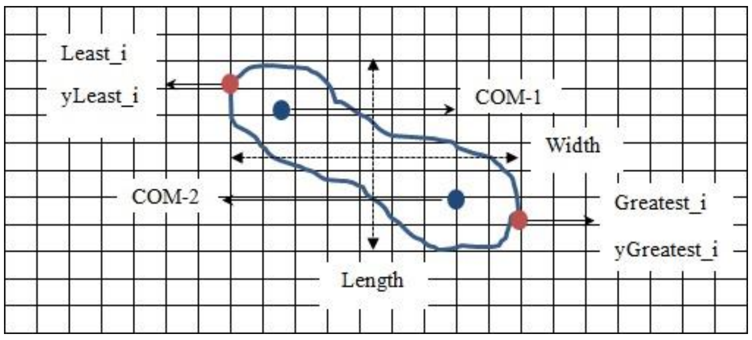


Figure 3.9: Coordinates of the COM of the mixed descending particle.

By comparing the coordinates of yLeast_i and yGreatest_i the ascending and descending form of mixed oblique particle can be recognised. In other words, if yLeast_i is bigger than yGreatest_i then the oblique particle will be in ascending shape, otherwise it will be in descending form.

After recognising different forms/shapes of the mixed particle, calculating the COMs of mixed particles are taken into account. As mentioned earlier, due to the division of mixed particle into two smaller particles in time, it is necessary to calculate two COMs for mixed particle. To approach this goal, calculations of two COMs are separately considered based on the distance of the lowest coordinate (least_i and least_j) of detected mixed particle to each COM with respect to the length and width of particle. Figure 3.10 shows an example of length comparison for mixed horizontal particle.

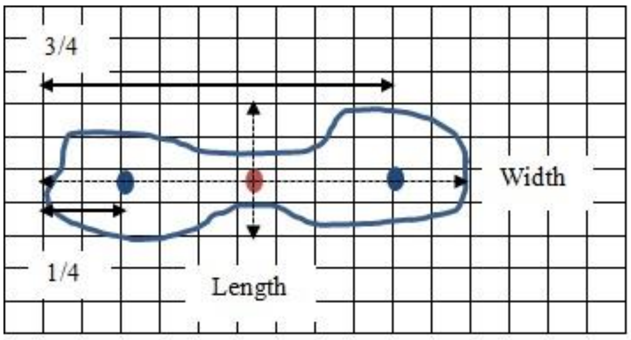


Figure 3.10: COM of mixed horizontal particle.

The following equations are used to calculate the COMs of mixed particles:

Mixed ascending particles:

$$\begin{aligned} X_{COM-1} &= (Length \times 3)/4 + Least\_i \\ Y_{COM-1} &= (Width \times 3)/4 + Least\_j \\ X_{COM-2} &= Length/4 + Least\_i \\ Y_{COM-2} &= Width/4 + Least\_j \end{aligned} \tag{3.5}$$

Mixed descending particles:

$$\begin{aligned} X_{COM-1} &= Length/4 + Least\_i \\ Y_{COM-1} &= Width/4 + Least\_j \\ X_{COM-2} &= (Length \times 3)/4 + Least\_i \\ Y_{COM-2} &= (Width \times 3)/4 + Least\_j \end{aligned} \tag{3.6}$$

Mixed horizontal particles:

$$\begin{aligned} X_{COM-1} &= Length/4 + Least\_i \\ X_{COM-2} &= (Length \times 3)/4 + Least\_i \\ Y_{COM} &= Width/2 + Least\_j \end{aligned} \tag{3.7}$$

For particles located on boundaries the coordinates are inconsistent. It means that grid points in the same particle will have more than 120 number of grid points distance from each other (with respect to grid size). In this case, by calculating the sum of grid points' coordinates, the COM can be calculated in the middle of the grid. In order to resolve this issue, PBCs were considered when a particle is on any of the boundaries. Periodic boundary conditions have been applied in eight situations. A particle can be on width boundaries, length boundaries or in all corners. Figure 3.11 presents the boundary conditions in different situations: a) PBCs in length boundaries (left, right); b) PBCs in width boundaries (top, bottom); c) PBCs in corners.

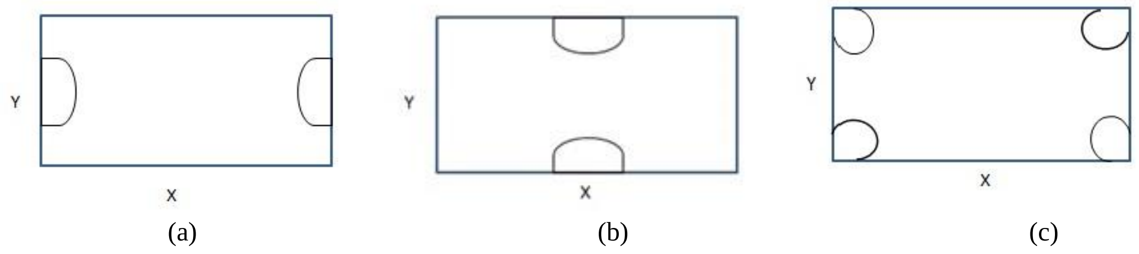


Figure 3.11: Examples of periodic boundary conditions in different positions.

3.2.3 Tracking Next Particle

A new target tracking method is an enhanced technique based on COM which is used to track the trajectory of particles. So far, all particles were detected in each time step (all grid points in each particle was assigned a unique number) and the COM of each particle was calculated. In this section the original idea is to use the COM and the expectation of particles movements in order to track the particles in each time step. These movements are effected by the direction and strength of a shear flow. Each particle cannot move more than a certain number of grid points forward, up and down on a lattice system. The particles according to the shear's direction also cannot move backward. The reader is reminded that the most important parameters affecting the morphology of diblock copolymers are the volume of the two polymers, temperature and external fields [18, 37]. In fact, the control of long-range order in structures is essential and important for any application in chemistry and material science. For example, the usage of diblock copolymers in any applications needs production of exceedingly ordered and free of defect structures. To achieve this goal, external fields such as electric field, surface fields or shear flow are playing important roles. Therefore, shear flow is applied to achieve a stable system with defect-free and highly ordered structures [18]. Although temperature is a very important parameter to tailor certain morphology, to accomplish a stable system requires considering an external field. According to section 2.3, when the shear flow is between 0.001 and 0.0001, the system obtains the spherical morphology with a hexagonal order (stable and perfect system), therefore all simulations results were executed with 0.0003 and 0.20 rates of shear flow and temperature respectively.

To identify the certain displacement of particle, frequency as a statistical method has been used to specify the movements of all particles in each time. In fact, frequency as a statistical method counts the total number of values which fall into specific samples or ranges. In order to calculate the frequency of particle movements, displacements of 10 random particles in 100 time-steps were taken into account and in total 1000 COMs of particles movements have been used. The following figures show how frequent the particles have travelled for certain displacement. The horizontal axis represents displacement magnitude and the vertical axis is the frequency of displacement happened. The square dots demonstrate frequency of certain displacements happened with respect to the displacement magnitude (in number of grid point terms) by random particles.

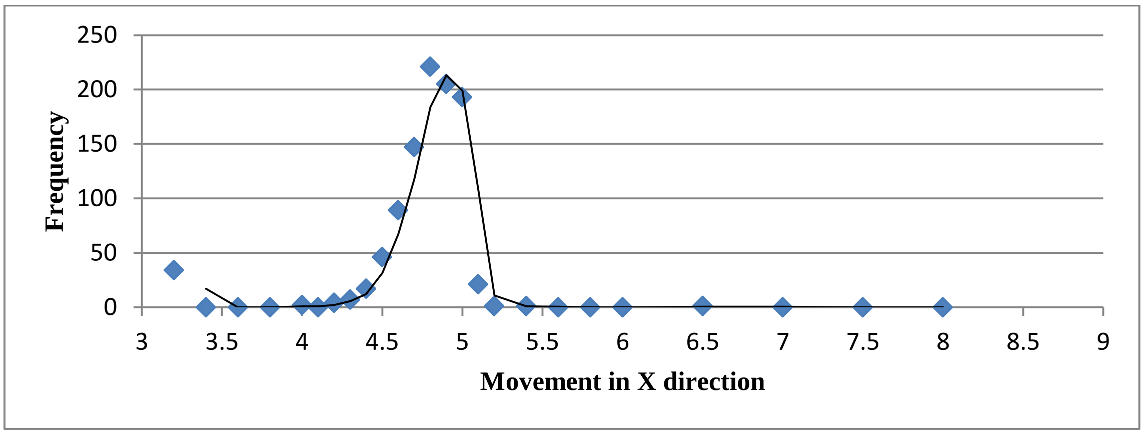


Figure 3.12: Frequency of data occurrence and movement of random particles in sequence time steps in X direction.

As shown in Figure 3.12 the most frequent number of grid points for a particle movement in X direction is around 5 grid points. Moreover, no movement has occurred in X direction exceeding 8 grid points. Based on this statistical result on particles' movement in X direction, 9 grid points is chosen as the limit of movements.

The same study was conducted for movements in the Y direction and 4 grid points were selected as the limit of travels in Y direction. According to Figure 3.9, the most numerous number of grid points for a particle movement in Y direction is between 0-0.2 and no movement occurred in the Y direction exceeding 4 grid points.

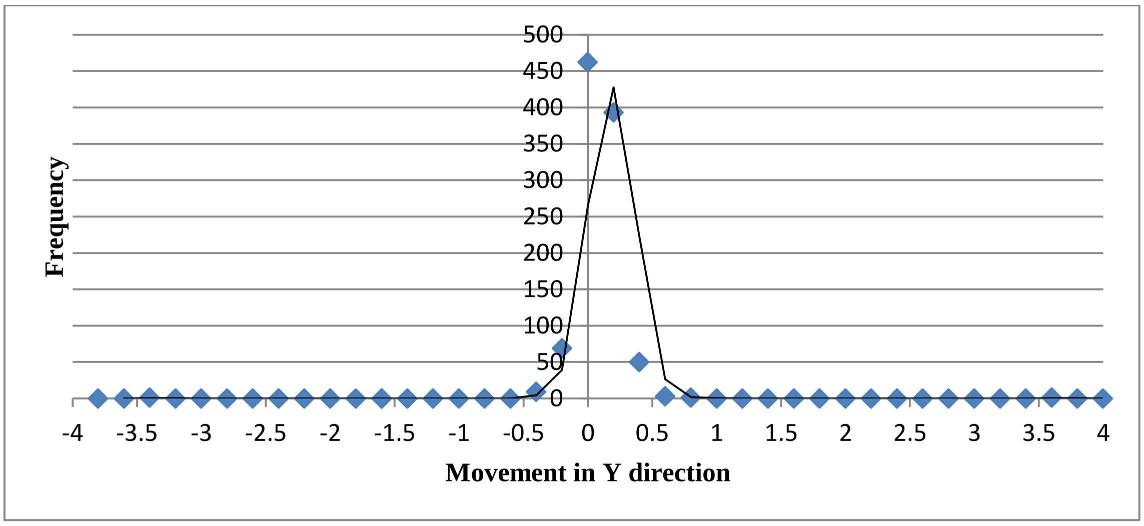


Figure 3.13: Frequency of data occurrence and movement of random particles in frequent time steps in Y direction.

In the current method, the initial time step is the *reference time step*, and the time step within which the considered particle is tracked is the *current time step*. Consequently, to find the considered particle in current time step, movement of COM from reference time step is considered, and periodic boundary conditions are also taken into account in all calculations. For instance, suppose particle A is in reference time step and the COM of particle A explains its reference position. In the current time step, particle A has moved forward in the X direction and might have variations in Y direction to up and down. This movement is estimated not to exceed 9 grid points in X direction and not more than 4 grid points up and down.

$$X\_R_A < X\_C_A < X\_R_A + 9$$
$$Y\_R_A - 4 < Y\_C_A < Y\_R_A + 4 \tag{3.8}$$

Here X_R_A explains the X coordinate of particle A in reference time step, X_C_A is its X coordinate in current time step. Y_R_A and Y_C_A refers to the Y coordinate of particle A in reference and current time step respectively.

Figure 3.14 illustrates example of tracking particle between reference and current time steps. the program searches inside the current time step for the particles with a COM from reference time step and 9 grid points forward as well as 4 grid points in Y direction up and down.

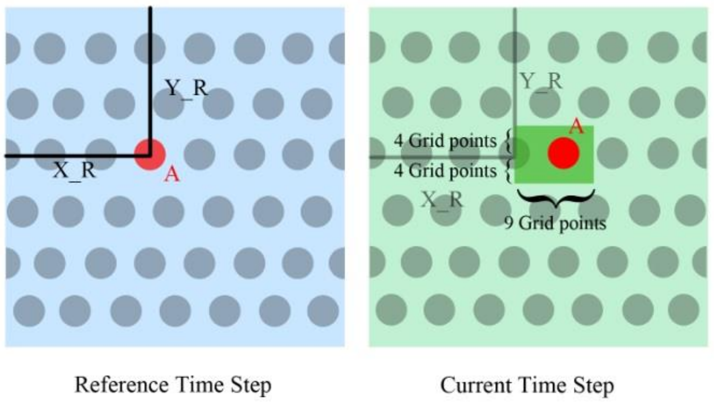


Figure 3.14: Illustration of movement of particle in time steps.

3.3 Benchmark Results

3.3.1 Segmentation Results

This part demonstrates the benchmark results of segmented outputs of CDS simulation with respect to a chosen value which differentiates between the two blocks (A, B), named distinguish value. Distinguish value is a value which specifies the most suitable criterion of ψ for dividing the particles into two groups of zero and one. In order to distinguish between two components different scales can be used. Choosing different numbers to separate two components from each other can allocate a bigger or smaller range to each polymer and change the size of the particles. Although this allocation can change the size of particles, this makes scant difference to their centre of mass (COM) coordinates. In fact, shrinking in size occurs in all directions it does not have any noticeable effect on COMs coordinates. Accordingly, the distinguishing value is chosen to be 0.0 with the intention of leaving enough space between the particles and making the outputs more accurate for analysis. Therefore, according to the range of values between two components (ψ) which was within [-0.9, 0.9], 0.0 is set as a distinguishing value for making the efficient division in this interval and this value will give equivalent weight to both components.

The following figure presents the segmented outputs between two components of a spherical system into zero and one, in which grid points with the value of one refer to the parts with originally positive values (particles). On the other hand, grid points with the value of zero are the grid points with negative original values (polymers around the particles). The total simulations were run on a 128×128 lattice for up to 100,000 time-steps with 0.0003 shear flow rate. It should be noted that Figure 3.15 shows the segmented output for the last time-step without any specific scale.

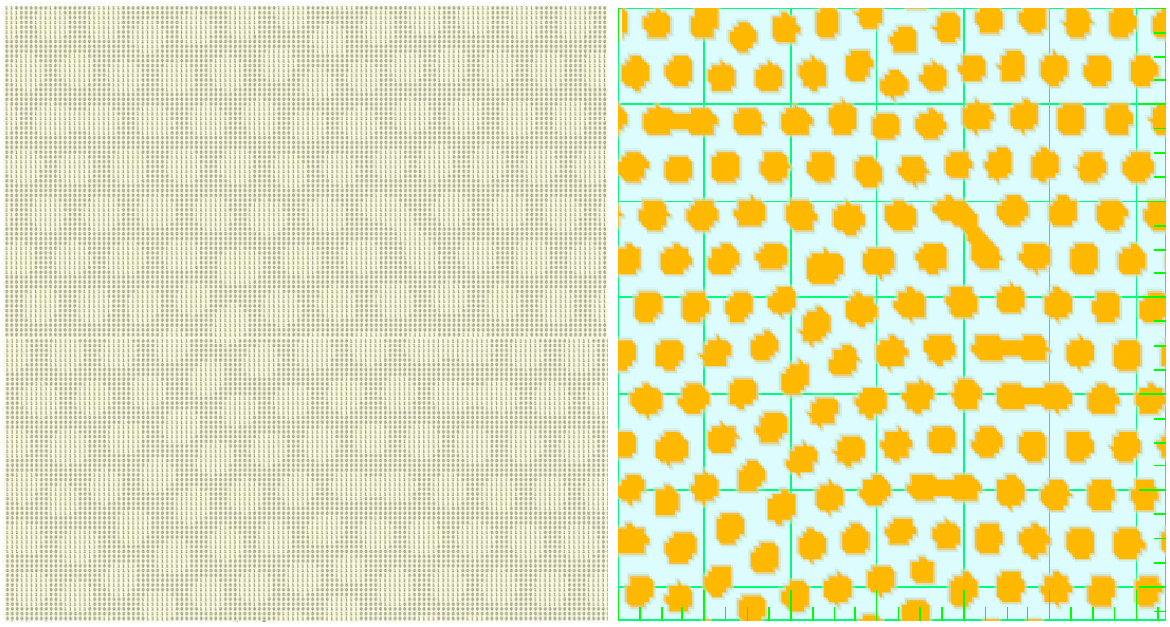


Figure 3.15: Segmented outputs of CDS in spherical system.

3.3.2 Benchmark Results for Detecting and Calculating COMs

Having distinguished between two components and obtaining the morphology of the system by assigning one and zero to related grid points, detecting particles becomes possible. Subsequently, the detection program explained in the previous section is applied to the segmented outputs. In total 154 particles were detected in the last time-step of simulations run on a 128×128 system size for up to 100,000 time-steps, each of which was assigned and labelled with a unique number and separated from the rest of the system.

It is also important to draw attention to the periodic boundary conditions effect. There are some particles on the boundaries of the grid. The program successfully detected these particles and labelled them in the two sides of the boundary identically.

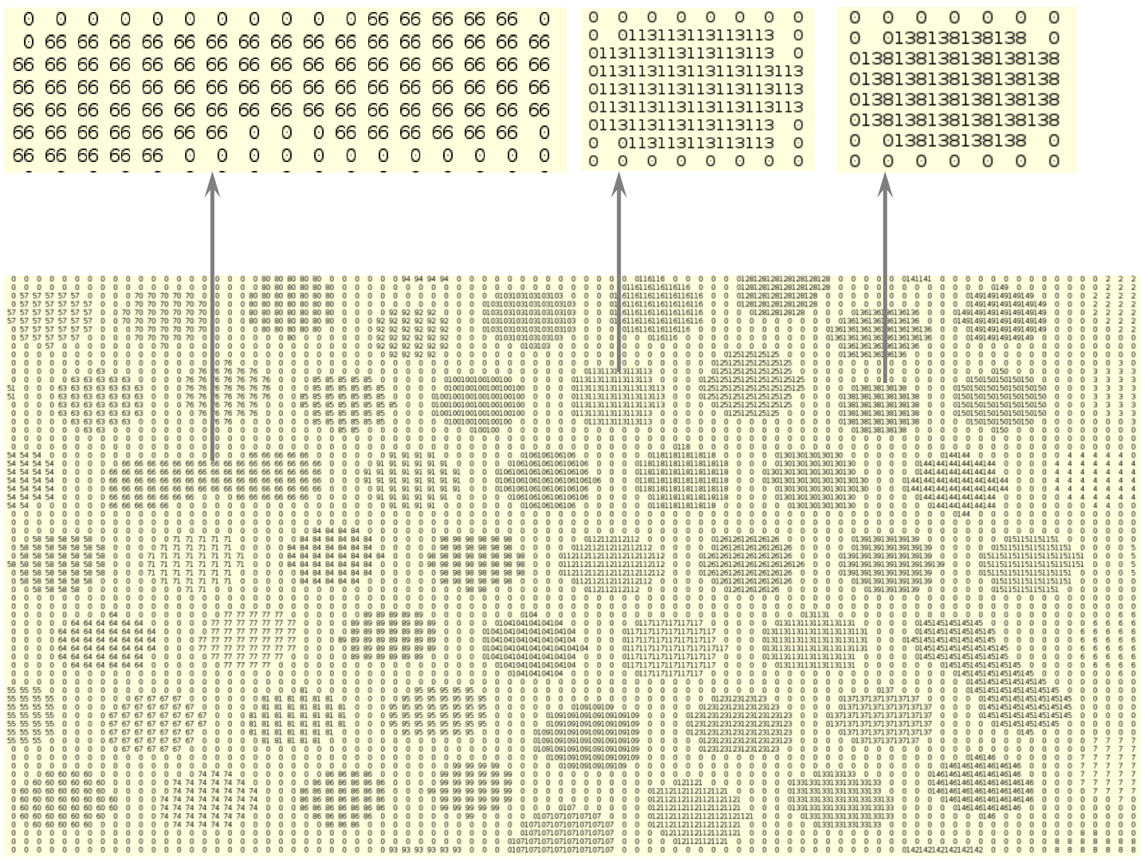


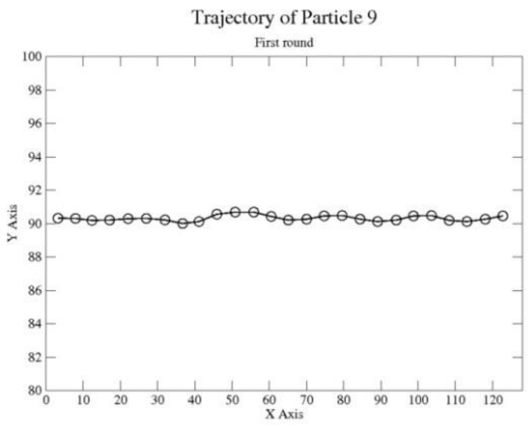
Figure 3.16: Second quarter part of detected particles.

After segmenting and identifying the domains into two categories calculating the COMs of detected particles becomes possible. As discussed previously, the COM of each particle is calculated based on its size. There are two types of particle according to their size and shape: single and mixed particles. The larger particles (mixed) can have three different directions: horizontal, oblique in ascending and descending way. To increase the accuracy of tracking, it is necessary to calculate two COMs for each of the mixed particles. For instance, particle 66 in Figure 3.16 is calculated as a mixed particle.

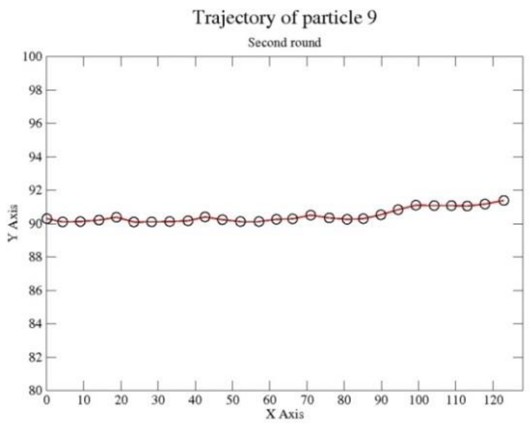
3.3.3 Results of Multiple Particle Tracking

This section presents the results of tracking the next position and dynamic behaviour of particle with respect to time evolution based on the new computational technique. In order to plot the track of each particle, the COM of the detected particle in each time step is used. As explained previously, the COM of each particle represents its position. The coordinates of particles are plotted in an X-Y coordinate system to illustrate the track of particles. In Figures 3.17 and 3.18 the numbers on X and Y axes represent the position in the grid with respect to the grid cells and do not have any specific unit. Each circle refers to the COM coordinate of the particle in one of the time steps.

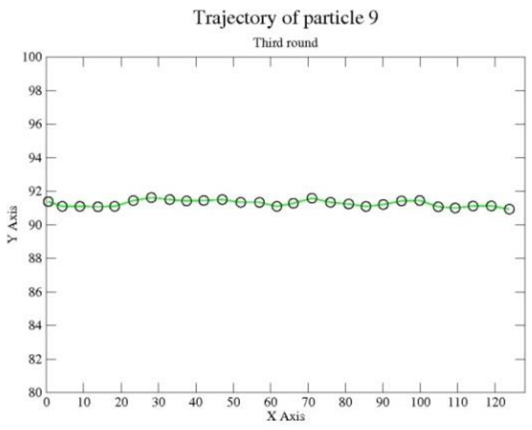
According to the periodic boundary conditions applied in the system, each particle travels in the length of the grid more than one time. The current simulation was executed on a 128×128 grid points system for 100,000 total time-step with 1000 time interval. Therefore, each particle moves approximately four times along the grid for a total of 100 time-steps. In each graph the movement of particle is tracked from one side of the simulation lattice box to another side. In the following figures, the whole journey of particle number 9 is illustrated and the movement of particle is tracked with different colours for each travel from the beginning of the lattice to the end.



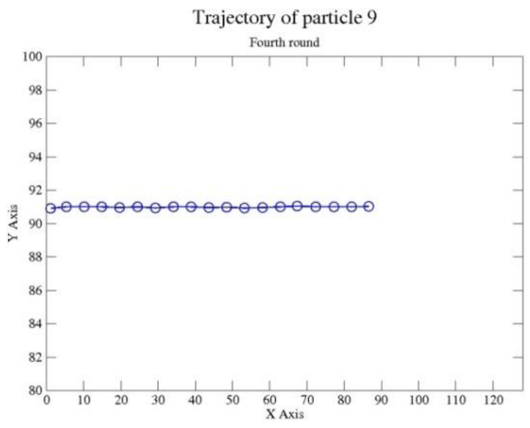
(a)



(b)



(c)



(d)

Figure 3.17: Trajectory of particle number 9 in different rounds.

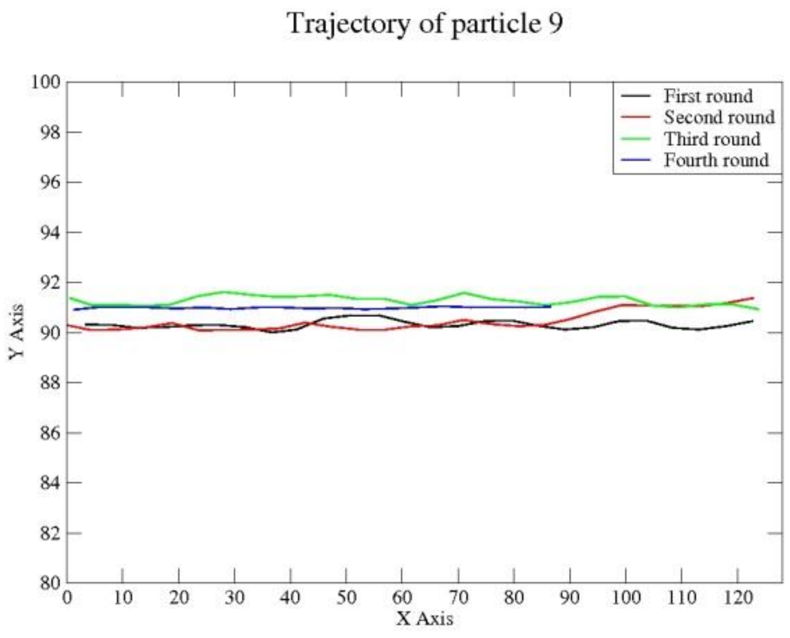


Figure 3.18: Entire trajectory of particle number 9.

From the above figures it can be seen clearly that the trajectory of particle 9 in each movement is getting straighter. The main reason is the stability of the system according to the time-steps. The trajectory line is linear and becomes a plateau in the last time-step of simulation. In fact, the ratio between total time-steps and system stabilisation is positive. Figure 3.19 shows the number of detected particles on 128×128 system size based on the different time-steps. The differences between numbers of detected particles in each time-step refer to the mixing or splitting of particles in different time-steps.

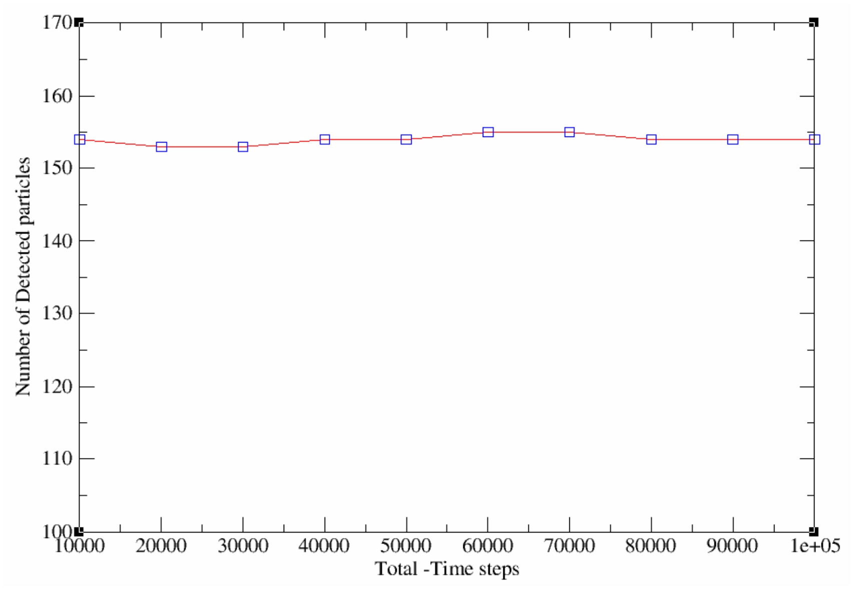


Figure 3.19: Number of detected particles.

Finally, it is also possible to illustrate the dynamic movement and behaviour of one or more particles concurrently based on the new method. The movement of each particle in the grid can be seen during the time. The value of grid points being the same as assigned number to detected particle and the colour of each particle is related to its number. Figure 3.20 presents the concurrent movements of three particles (70, 134, and 145) in 128×128 lattice system. The particles in red and orange are considered to be in the mixed/merged particles category, while the particles in green are single particles.

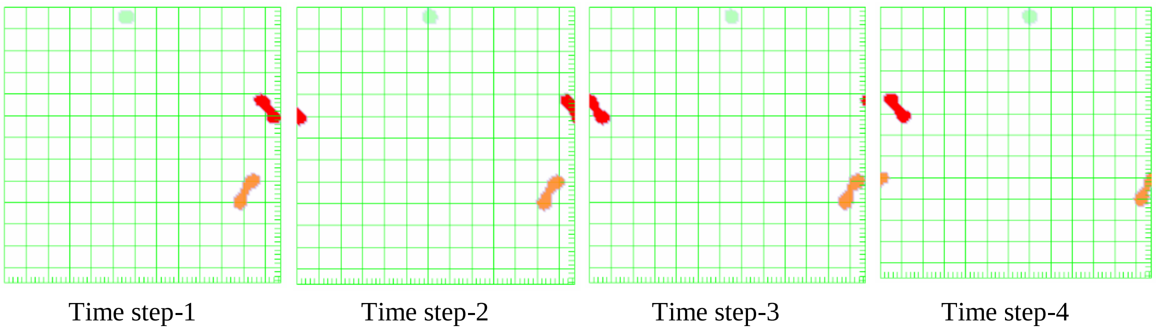


Figure 3.20: Concurrent dynamic movements of multiple particles.

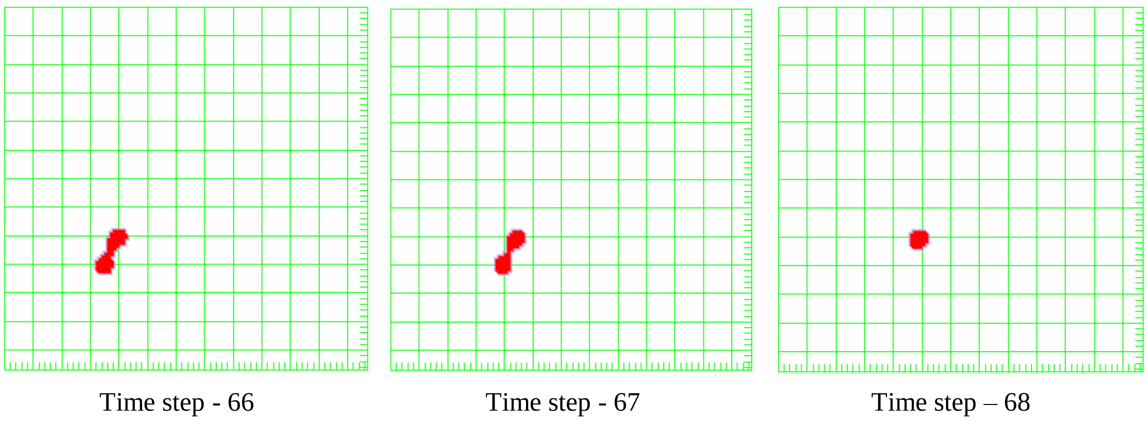
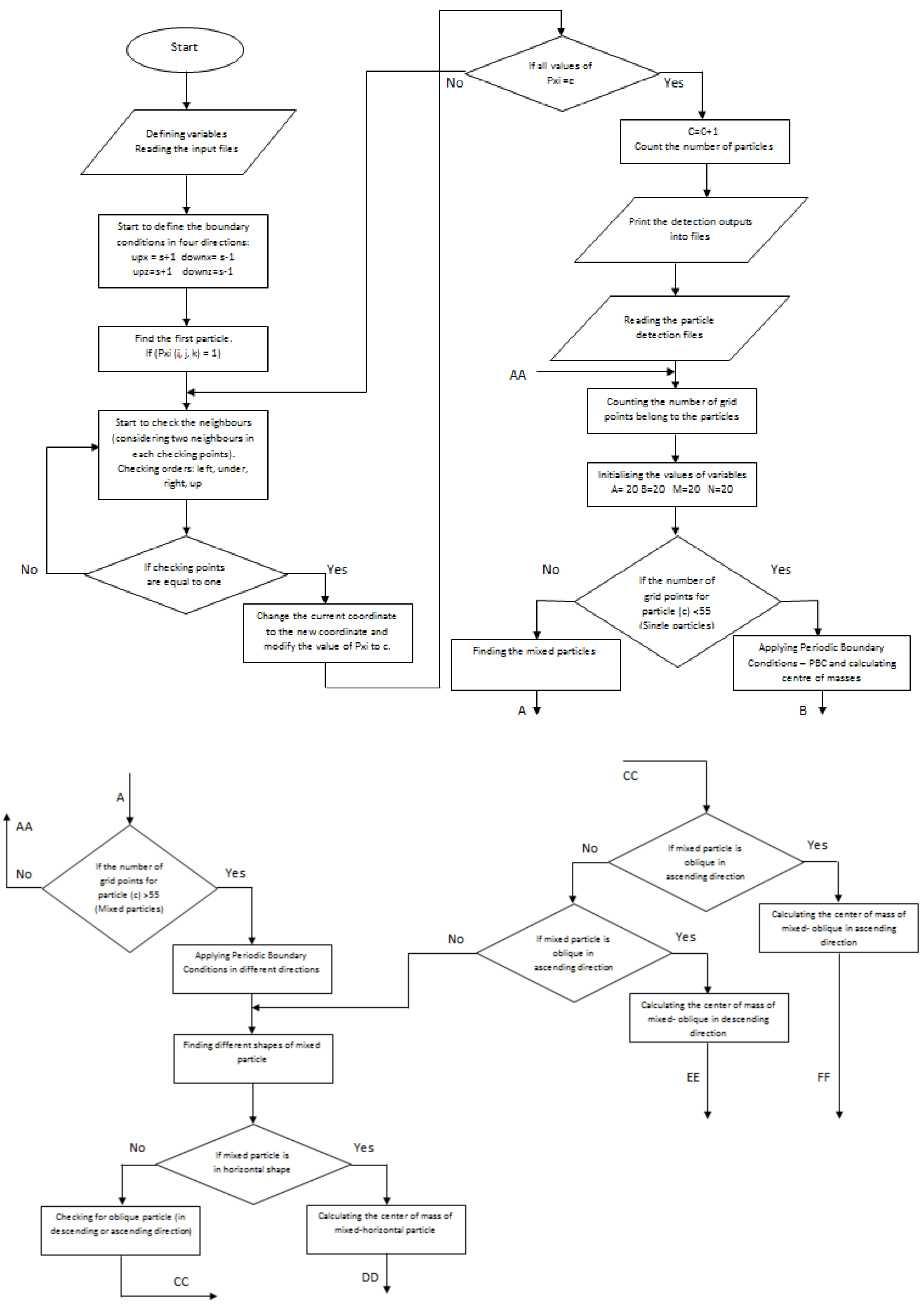


Figure 3.21: Division of mixed particle 134.

3.4 Flowchart of Computational Algorithm for Detecting and Tracking


Start
Defining variables
Reading the input files
Start to define the boundary conditions in four directions:
upx = s+1 downx= s-1
upz=s+1 downz=s-1
Find the first particle.
If (Pxi (i, j, k) = 1)
Start to check the neighbours (considering two neighbours in each checking points).
Checking orders: left, under, right, up
If checking points are equal to one
No
Yes
Change the current coordinate to the new coordinate and modify the value of Pxi to c.
If all values of Pxi =c
No
Yes
C=C+1
Count the number of particles
Print the detection outputs into files
Reading the particle detection files
AA
Counting the number of grid points belong to the particles
Initialising the values of variables
A= 20 B=20 M=20 N=20
If the number of grid points for particle (c) <55 (Single particles)
No
Yes
Finding the mixed particles
Applying Periodic Boundary Conditions – PBC and calculating centre of masses
A
B
A
AA
No
If the number of grid points for particle (c) >55 (Mixed particles)
Yes
Applying Periodic Boundary Conditions in different directions
Finding different shapes of mixed particle
If mixed particle is in horizontal shape
No
Yes
Checking for oblique particle (in descending or ascending direction)
Calculating the center of mass of mixed-horizontal particle
CC
DD
CC
If mixed particle is oblique in ascending direction
No
Yes
Calculating the center of mass of mixed- oblique in ascending direction
If mixed particle is oblique in ascending direction
No
Yes
Calculating the center of mass of mixed- oblique in descending direction
EE
FF

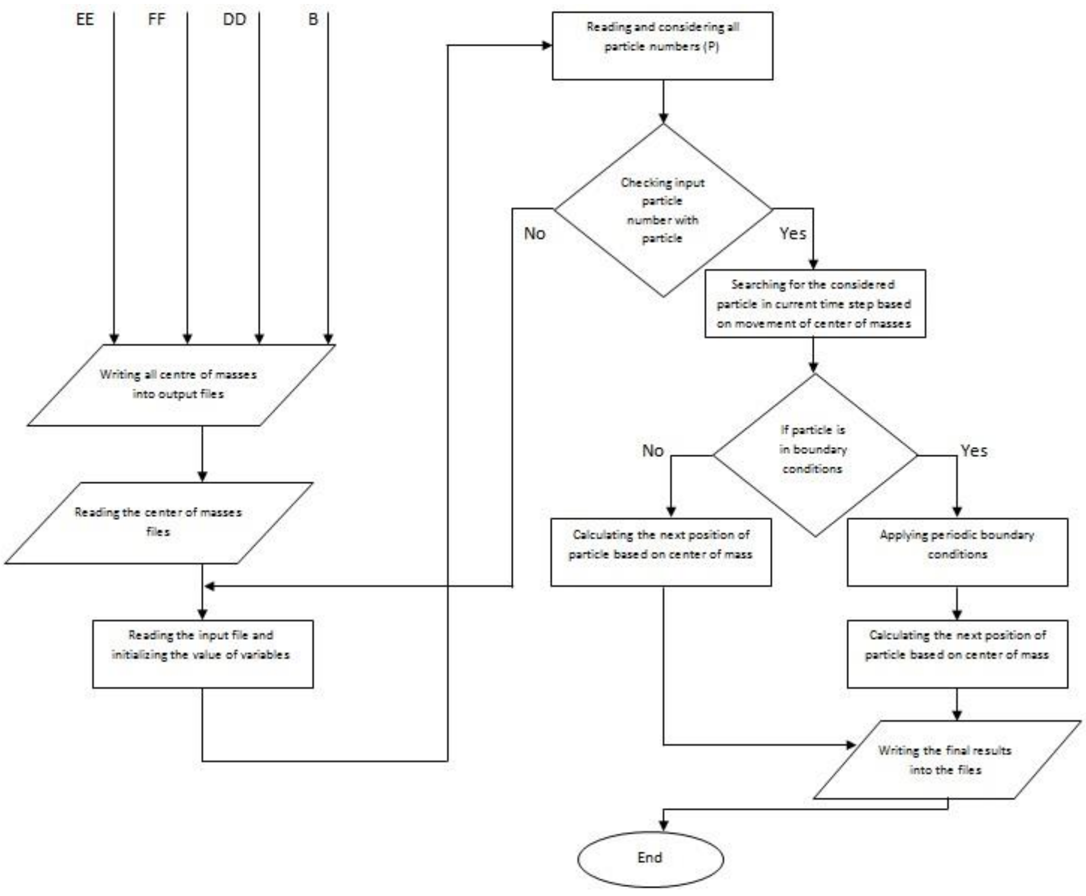

EE
FF
DD
B
Writing all centre of masses into output files
Reading the center of masses files
Reading the input file and initializing the value of variables
Reading and considering all particle numbers (P)
Checking input particle number with particle
No
Yes
Searching for the considered particle in current time step based on movement of center of masses
If particle is in boundary conditions
No
Yes
Calculating the next position of particle based on center of mass
Applying periodic boundary conditions
Calculating the next position of particle based on center of mass
Writing the final results into the files
End

Figure 3.22: Flowchart of the program.

3.5 Summary

A novel particle tracking technique[1] for a spherical phase diblock copolymer under a shear flow has been developed and presented. Two frameworks have been proposed to achieve the computational technique of tracking particles. In detection framework, neighbouring search technique is used for detecting particles and reconstructs the time-lapse of detected particles. In tracking framework, the centres of mass of particles have been calculated and particles were tracked based on their centres of mass and the movements in each time step. The proposed method was examined with various test cases and satisfactory results in terms of accuracy and concurrently tracking of particles were achieved, giving confidence in the technique developed in this study. The numbers of detected particles also changed in time until the whole system reached equilibrium (with no more changes after that specific time-step). Detected particles mixed and divided in time evolution.

[1] H. Soltani, D. Ly and W. Ahmed, "Computational Technique of Particle Tracking", *Fourth Annual Research Student Conference*, UCLAN, Preston, December 2014.

CHAPTER FOUR

4 Performance Analysis and Computational Throughput

When introducing new hardware architectures, it necessary to adapt programing languages that can support new hardware features. This fact raises a challenge and issue for scientific programmers to obtain the best throughput, performance and functionality. In fact, different scientific programs and applications which have existed for many years to solve specific problems must be updated to reflect the newest architectural systems. Consequently, based on the mixture of program size (number of code lines) and the cost of modification, it is important to consider new features such as scalability and portability and identifying bottleneck regions of code to answer the functionality challenges over the course of different platforms/architectures and improve the throughput and performance of code.

In this chapter we focus on computational throughput and performance analysis which involves four different perspectives: code optimisation, performance modelling, visual profiling and benchmarking.

4.1 Benchmarking

Different hardware architectures have different theoretical peak of performance and throughput based on floating point operations per second (FLOP/S); in reality it is difficult to obtain this peak of throughput. In effective (achievable) level, most current hardware architectures consider a single precision for calculation an instruction stream in SIMD execution units, which sometimes is less accurate compared to double precision [63]; less accuracy refers to the different rounding strategies for the addition, subtraction, multiplication and division operations between single and double floating point. In fact, the floating point calculations should be rounded in order to match/fit into a finite number of bits in memory, which is 32 bits for single precision and 64 bits for double precision.

On the other hand, use of double precision is not very straightforward due to the memory limitation and differences between the number of operations such as multiplication and addition. Therefore, there is extensive research on different metrics to decrease the gap between theoretical and effective level, to improve the arithmetic performance and to demonstrate the real performance obtained by program. To this end, benchmarks as pieces of code can help us to gather different data such as effective throughput and effective memory bandwidth.

There are two types of benchmarks: kernel or micro benchmark, which refers to low-level specifications and information of hardware; and application or macro benchmark, which indicates high-level hardware architecture information. The most well-known micro for the first type of benchmark is LINPACK [64], which can be used to specify a system's sustained FLOP/s rate or STREAM to assess the memory bandwidth [65]. However, it is complex and difficult to determine the performance and throughput metrics produced by kernel/micro benchmarks for multifaceted programs. Kernel benchmarks are mainly about general drifts and trends in hardware [66]. In fact, the communication and interaction between different hardware sections is more likely to be an outcome in lower throughput and performance than that produced by kernel benchmarks, and also the behaviour of some low-level components such as data cache are difficult to be measured by kernel benchmarks.

The latter type is more popular in engineering and scientific research, and it can be utilised to define information about the computational behaviour of specific programs. Well-known and popular examples of macro benchmarks include the NAS Parallel benchmark developed by the NASA Ames Research Centre [67]; and the ASC benchmark suite developed by the Los Alamos and Lawrence Livermore National Laboratories [68]. These benchmarks mainly effect a breakdown of the execution time, such that the performance and throughput bottlenecks of different applications can be recognised. This type of benchmark can be used with other types of performance analysis (visual profiling) to help identify performance bottlenecks in programs.

4.2 Visual Profiling

In many cases, considering the low-level and high-level benchmarks for identifying the effective performance and throughput of program is not easy and even not all programs can be split into enough low level to specify the bottleneck of program and to determine the effective performance. Therefore, it is necessary to utilise a visual profiler as a tool to monitor the whole program during time executions, enabling developers and programmers to evaluate high level performance analysis in terms of: (*i*) execution time [69, 70]; (*ii*) time spent for copying, synchronisation, reading and writing [71]; and (*iii*) memory usage [72, 73]. They must also study low level metrics such as kernel performance and L1 cache or shared memory consumption [74]. In addition, visual profiler provides this chance to profile and monitor the whole program in both cases (low and high levels) without any changes in the source code.

The visual profilers used in this study refer to the NVIDIA NSIGHT ECLIPSE which is part of CUDA toolkit and Intel Vtune Amplifier which is visual performance analysis. Nsight Eclipse visual profiler is a combined CPU and GPU integrated development environment (IDE) for monitoring and implementing CUDA programs, to help programmers on all levels of benchmark metrics (low and high) and to support developers on different steps of the program development pipeline [75]. Some of the main functions of Nsight Eclipse visual profiler are as follows [76]:

- Source code editor with support of CUDA C and C++
- Graphical user interface for debugging programs
- Visual profiler for source code
- Visual profiler for optimising program performance
- Program lifecycle management
- Compiler integration
- Occupancy profiler
- Memories profiler

Therefore, the method of profiling utilised in this research is to use source code directly. In fact, by instrumenting the whole source code on profiler, different situations can still be addressed, such as the performance of one specific kernel or the performance of a whole program in total execution time.

4.3 Code Optimisation

After breaking down the source code into different levels and identifying the bottleneck regions of a program, code optimisation can play significant role of performance and throughput analysis. Code optimisation can be divided into four main forms: (*i*) sub-code transformations to decrease the number of instructions (e.g. loop unrolling and tiling); (*ii*) rewriting code to be compatible with new hardware architectures (e.g. SIMD and vectorisation) [77]; (*iii*) using L1 cache or sheared memory to improve throughput and cache behaviour [78, 79]; and (*iv*) considering totally new algorithm with less computational complexity, more scalability and portability (this form of code optimisation is more demanded in parallel computing) [80]. It should be noted that in many cases analysing and measuring performance enhancements of code optimisation can be done against the original or baseline code which execute in the same specification and configuration system. In parallel computing, complex computation and arithmetic intensive parts move to accelerator section (e.g. GPU) and measuring performance improvements is performed based on the comparisons between parallel code using accelerator (GPU) and optimised or un-optimised CPU code with consideration of same floating point and same system scales [81, 82, 83].

Following parallel computing, ‘directive’ based programming can be used as a flag (pragma) to transfer the parts of complex computation code to an accelerator. Different application programs interfaces (APIs), such OpenMP [84, 85] and OpenACC [86, 87], support this ability. This approach helps developers to improve the throughput of application and to be compiled with different compilers (cross-compiled) for an accelerator. For instance, OpenACC and OpenMP programming standards are supported by different compilers from Intel, CAPS and PGI [84, 88]. Although this approach enhances the throughput of program, it should be noted that it still requires a code optimisation to achieve an efficient performance [89]. Another method/approach is ‘active libraries’, which allow programmers to handle the optimisation issue in a high level specific manner. In fact, this approach refers to leave the code optimisation and implementation to smart compiler and libraries. Although some examples (such as OP2 [90] and Liszt [91]) exist for this type of approach, to our knowledge the current generation of smart compiler and libraries are not very accurate, precise and expressive to handle a complex program into a binary optimised mode. In fact, the compiler does not have enough knowledge of the data dependencies, the problem of domain or how the code will be executed, thus it cannot make definite hypotheses and assumptions. Drawing on this background, in this thesis we consider different optimisation approaches based on different situations such as code transformation, vectorisation, SIMD intrinsic, memory access pattern, coalescing global memory access, kernel fusion and a new algorithm with lower complexity, divergent paths and synchronisation coordinates. Finally, we strive to obtain a reasonable comparison of CPU and GPU as much as possible by using all available resources in both architectures.

4.4 Performance Tuning

Performance tuning contains a group of techniques that can help to estimate the effective performance and throughput of program. In fact, performance modelling/tuning, by identifying performance bottlenecks of program and evaluating the influence of code optimisations in the development process, helps to predict the effective performance of application in new architecture, and decreases the gap between maximum arithmetic performance and achievable performance [92, 93, 94]. In this context, the main technique of performance tuning refers to the ‘analytical calculation and modelling’ where computational throughput, execution time and memory bandwidth are presented mathematically as equations. The main benefit of analytical calculation is fast evaluation and prediction of program performance and throughput. However, the parametrisation of equations is not straightforward and needs a good understanding of code behaviour and hardware architecture.

The following equation shows analytical calculation of effective memory bandwidth for a program based on GPU [95].

$$BW_{Effective} = \left(R_B + W_B\right) / \left(t \times 10^9\right). \tag{4.1}$$

Where $BW_{Effective}$ indicates the effective bandwidth in units of GB/s, R_B refers to the number of bytes read per kernel, W_B is the number of bytes write per kernel, and t refers to the execution time given in seconds. In fact, in many cases a big percentage of programs are memory bandwidth bound, therefore analytical calculation of effective memory bandwidth helps to understand the bottlenecks and to improve performance of programs.

The theoretical bandwidth can be identified by hardware specifications. For instance, the Tesla M2050 NVIDIA GPU with double data rate RAM (DDR), 1,546 MHz memory clock rate and 384 bit memory interface has 148 GB/s theoretical memory bandwidth [95].

$$BW_{Theoretical} = 1546 \times 10^6 \times (384/8) \times 2/10^9 .$$

In theoretical bandwidth calculation, memory clock rate convert to Hz, memory interface converts to byte and multiply by 2 because of dual interface rate, and finally the whole result is divided by 10^9 to convert the unit to GB/s.

Equation 4.2 demonstrates the calculation of effective data throughput (memory bandwidth) by knowing the process of accessing data in a program. Another essential analytical metric that directly relates to the program's algorithm and code optimisation is computational throughput. As with theoretical peak bandwidth, maximum theoretical computational throughput depends on hardware architectures. For instance, the maximum theoretical throughput for the same NVIDIA TESLA M2050 GPU device for single precision floating point throughput is 1030 GFLP/s, and for double precision floating point it is 515 GLOP/s. In the effective computational throughput context, the following equation can be used to calculate computational throughput [95, 17]:

$$CT_{Effective} = OP_N \times E_N / (t \times 10^9). \tag{4.2}$$

Where *OP* refers to the number of operations, *E* indicates number of data elements in a program and t is execution time in seconds. However, calculating effective computational throughput in complex programs is very difficult and laborious. Accordingly, it is more beneficial and helpful to utilise visual profiling and analyser to understand the effective computational throughput of program and to identify bottlenecks issues in computational throughput, which can be useful to optimise and improve performance.

Finally, analytical modelling of total execution time that can be used to evaluate and estimate execution time of program is:

$$T_{Total} = (T_{Computation} + T_{Communication} - T_{Overlap}) + T_{Synchronisation} + T_{Overhead}. \tag{4.3}$$

Here T_{Total} refers to the sum of elapsed time of arithmetic calculation and communication (transferring data), minus the total time that arithmetic computation and communication consumed concurrently, plus the last two terms which refer to the overheads of synchronisation and communication for data transformation, which cannot be overlapped. This type of analytical modelling can obtained high levels of precision on different types of program in numerous scientific fields and applications [96, 97].

4.5 Summary

This chapter presented performance analysis and computational throughput from different aspects. We adopted these techniques in this thesis, specifically code optimisation, to make a comparison between baseline source code and optimised source code on CPU, using directive-based programming (OpenMP) to consider parallel computing for optimised source code on CPU and to develop a new algorithm with less computational complexity, more scalability and portability for GPU. We used visual profiling to identify and evaluate different types of bottlenecks in computational and data throughput for improvement the performance of cell dynamic simulation program. Finally, we explored analytical calculation of performance modelling with the concept of theoretical peak and achievable peak for data and computational throughput and total elapsed time. Finally, because of the complexity, difficulty and size of scientific programs, these techniques are becoming usual and common for analytical computation and performance analysis [98].

CHAPTER FIVE

5 Parallel Computing and Programming Model

This chapter explores the concept of parallel computing and parallel architectures, and discusses different terminology related to parallel computing. Traditionally, computer architectures were based on serial executions and they did not support for feature parallelism. In serial processing a problem/program divides into different series of commands by which instructions are performed seriatim. Serial computing is mainly performed on a single processor, whereby only one instruction can be executed at a time. Clearly this type of execution is not suitable for complex and expensive computations, and it causes long time delay to solve problems. The following figure shows an example of how a problem can be executed in serial computation.

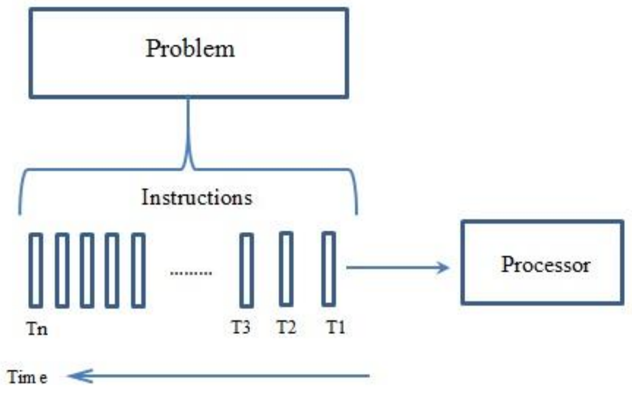


Figure 5.1: Example of serial execution.

Parallel computing is the obvious answer to solve expensive and complex problems with less time consumption. In parallel computing a problem is divided into separate and distinct sections, each of which is further split into a series of commands and instructions, which can be executed concurrently and simultaneously on multiple processors. In other words, parallel processing solves a problem by utilising multiple resources (processors) simultaneously. Resources are mainly divided into two categories: (*i*) a computer with multiple processors, and (*ii*) group of computers with multiple cores that are connected to each other on network. Figure 5.2 illustrates how a problem can be executed in parallel execution.

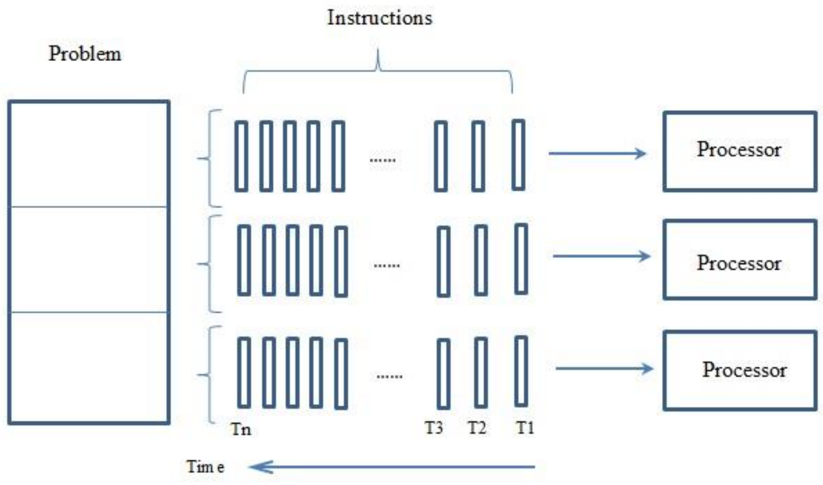


Figure 5.2: Example of parallel execution.

It should be noted that there is a difference between the concepts of concurrency and parallelism; the former refers to the properties of a system when multiple tasks can be progressed at the same time while the latter utilise the concurrency of a system to actually execute tasks simultaneously. The following figures present examples of concurrent and parallel execution.

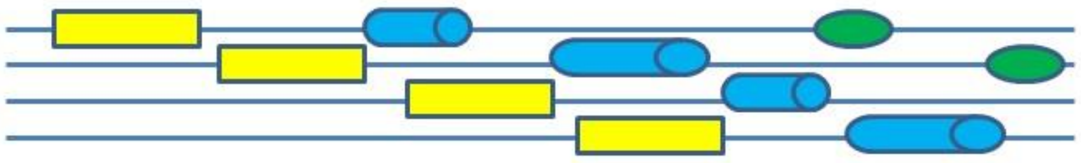


Figure 5.3: Concurrent execution.

In Figure 5.3 each colour indicates active tasks in a block being performed sequentially and in mutual swapping, while their simulation execution is in parallel. Figure 5.4 demonstrates concurrent and parallel executions, in which all tasks are executed at the same time in a block.

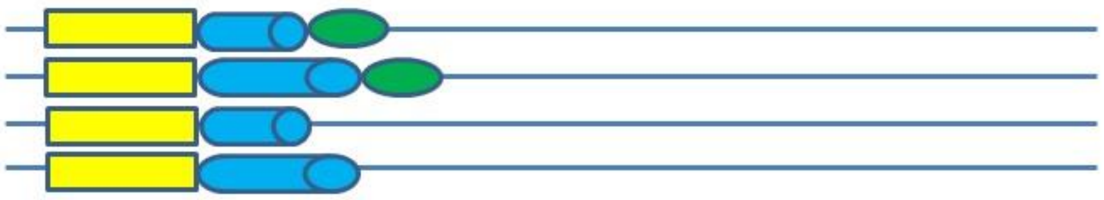


Figure 5.4: Concurrent and parallel execution.

In regard to the concurrent and parallelism concepts, applications or programs are classified into two classes: (*i*) concurrent program, and (*ii*) parallel program. Concurrent program has concurrency built in the problem definition. A good example of concurrent application refers to web server designed to be concurrent from beginning to take concurrent inputs. In parallel application, tasks are executed at the same time for faster processing or to handle a bigger amount of problems. In fact, in concurrent application, there is no answer to identify the problem without concurrency, because of the concept of a problem. Figure 5.5 shows the relationship between program, concurrent program and parallel program [99].

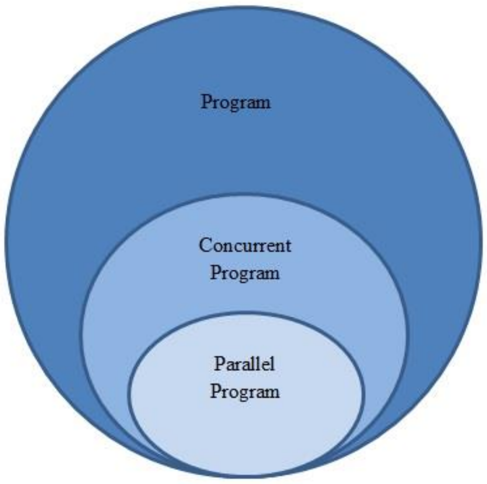


Figure 5.5: Relations between program, concurrent program and parallel program.

In 1960 Gene Amdahl introduced a formula to calculate the potential program speed-up by a portion of code which can be parallelised [100]. In other words, Amdahl’s formula (known as Amdahl’s Law) demonstrates that a fraction of a program that cannot be parallelised will impact directly on the whole speed-up obtainable from parallelisation, and the expected speedup of the parallel program over the serial program when using N processors is dictated by the proportion of a program that can be made parallel. The following equation states the maximum speed that can be calculated based on Amdahl’s Law [101]:

$$S(N) = \frac{1}{(1-p) + p/N}. \tag{5.1}$$

Here p refers to the fraction of code that can be parallelised, $(1-p)$ indicates the portion of a program that cannot be made parallel, S is the speed-up, and N is the number of processors.

Relative to Amdahl’s Law, Gustafson proposed a new law about the speed-up with N processors. Gustafson’s Law states the whole amount of the work can be executed in parallel while linearly different based on the number of processors [100, 101].

$$S(P) = \alpha + P(1 - \alpha). \tag{5.2}$$

Here P refers to the number of processors, S indicates the speed-up, and α is the serial part (not parallelised) of a program. In fact, by introducing new hardware architectures, and more resources becoming accessible, Amdahl’s Law, which was based on fixed size of problem (i.e. a portion of code can be executed in parallel, separate and independent of the total number of processors) is no longer realistic for the evaluation of parallel performance, while Gustafson’s Law offers a more accurate evaluation of parallel performance. Table 5.1 presents an example of the Gustafson’s law with 0.1 percentage of α , and various numbers of processors.

Number of processors	Non-parallelised portion (percentage)	Speed-up
10	0.10	9.1
100	0.10	90.1
120	0.10	108.1
150	0.10	135.1

Table 5.1: Example of Gustafson’s law speed-up.

5.1 Parallel Terminology

Parallel computing comprises many terminologies which are important to explore for a better understanding the context of parallel computing.

- **Central Processing Unit (CPU):** also known as the processor, core, and socket based on singular execution or multiple executions. CPUs are mainly divided into two categories: (*i*) scalar processor, and (*ii*) vector processor. A scalar CPU performs computation on each single data packet separately, one at a time. On the other hand, a vector or array processor performs computations on one dimensional array/vector concurrently and simultaneously enhances the performance of a system [102]. Most modern CPUs support vector processing.
- **Task:** a program which comprises group of instructions to be executed by a processor or multiple processors. In fact, a task indicates a logically distinct portion of computational job.
- **Shared Memory:** this divides into two aspects: (*i*) hardware aspect - physical memory that all cores/processors can access; and (*ii*) programing aspect - defines a model that all parallel tasks can read, write and access to the identical logical memory.
- **Symmetric Multi-Processor (SMP):** indicates shared memory parallel architecture by which multiple processors have equivalent access to all resources.
- **Distributed Memory:** network parallel architecture whereby all computers (nodes) on a network use communication to utilise resources on other devices.
- **Communication:** in parallel computing, communication generally means data exchange between instructions of parallel tasks, which can be in shared or in distributed memory architecture.
- **Synchronisation:** in parallel computing, synchronisation is the coordination of parallel computations in real time, by which a parallel task cannot be started until another task reaches the same level [103]. Usually, synchronisation increases the execution time of a program by adding a wall-clock.
- **Overhead:** the amount of elapsed time which no useful and beneficial work has processed. Parallel overhead involves many issues such as synchronisation, communications and data reading and writing.
- **Scalability:** the capability of parallel system to adopt new resources features, such as program algorithms and hardware features, specifically memory bandwidth (data throughput) and computational throughput.

- **Massively Parallel:** utilisation of a big number of processors to execute a group of computation tasks simultaneously and concurrently.
- **Embarrassingly Parallel:** also known as perfectly parallel, this is when a program is small and there is no need to distinguish groups of parallel tasks. In this type of situation, there is no communication between parallel tasks.
- **Granularity:** the ratio between computations to communications in parallel computing. Granularity is classified into two classes: fine grained and coarse grained. Fine grained refers to small volumes of computational task with less data volume (frequently) transferring between processors. On the other hand, coarse grained parallelism indicates the huge volume of computational task with infrequent communication between processors [104]. Fine grained parallelism improves the performance and speed of program, but also increases the overhead execution time. In order to achieve the best performance, the system should have a balance in granularity.

5.2 Flynn's Taxonomy for Parallel Computing

Flynn's classification has been one of the most utilised taxonomies for parallel and sequential computing since 1966. In Flynn's classification, multiple processor computer architectures are categorised based on the two parameters of instruction stream and data stream, each of which can only use a single or multiple set of instructions and data streams [105, 106]. According to the Flynn's taxonomy there are four different classifications of parallel and sequential computers: SISD, SIMD, MISD and MIMD. It should be noted that the term 'stream' indicates a sequence of data or instructions process by the CPU in one complete cycle. Figure 5.6 presents example of instruction and data stream. Therefore, the group of instructions processed by the CPU are called the instruction stream and a dataset is needed for processing instructions, called a data stream. Table 5.2 demonstrates different types of Flynn's taxonomy.

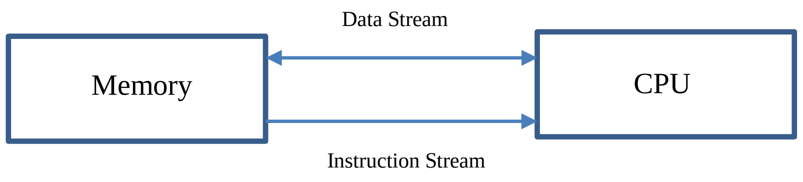


Figure 5.6: Instruction and data stream.

Flynn's taxonomy	Description
Single Instruction, Single Data (SISD)	This class of Flynn's taxonomy refers to the sequential computing by which only single instruction can be performed on one CPU (single processor) in one clock cycle, and just one data stream can be utilised as input data in one clock cycle [107, 108]. Scalar processor is categorised in the SISD classification of Flynn's taxonomy. SISD processing is shown in Figure 5.7.
Single Instruction, Multiple Data (SIMD)	SIMD is another type of parallel computer in Flynn's taxonomy by which multiple processors process a single instruction or the same operation with multiple data streams simultaneously, in parallel. In SIMD type, instructions have to be completed by assigned processers before other instructions can be started for execution [109]. Consequently, synchronisations exist for the execution of operations in SIMD. Vector/array processors and GPUs are classified in this type of Flynn's taxonomy. Figure 5.8 illustrates SIMD computation model.
Multiple Instruction, Single Data (MISD)	In MISD model, each processing unit processes different instructions in the same data stream. This type of Flynn's taxonomy is not very common compared to the other types of parallel techniques.
Multiple Instruction, Multiple Data (MIMD)	In this technique, each processing unit processes different instruction streams on different data streams. Instruction executions in this type of Flynn's taxonomy can be synchronous or asynchronous. It should be noted that MIMD technique comprises an SIMD class of parallel computing.

Table 5.2: Flynn's taxonomy types.

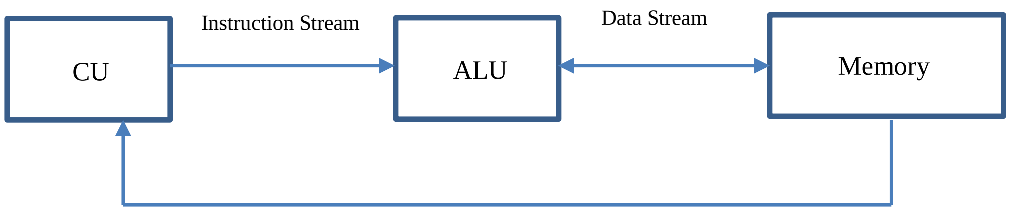


Figure 5.7: SISD model, where CU refers to Control Unit and ALU is Arithmetic Logic Unit.

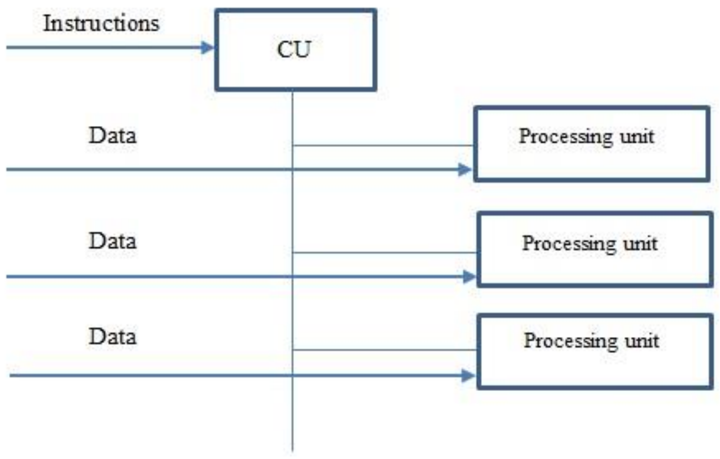


Figure 5.8: SIMD Computation model.

5.3 Parallel Programming Model

New hardware architectures are highly parallel and they are not limited to any specific type of parallelism or parallel memory architecture. This section explains different types of parallelism and then details different parallel programming models utilised in this study based on the different types of parallelism. In general, there are four main types of parallelism: (*i*) bit level parallelism; (*ii*) instruction level parallelism; (*iii*) task level parallelism; and (*iv*) data level parallelism.

1. *Bit level parallelism:* this is one of the earliest forms of parallel computing, based on increasing processor word size. A word in computer science is a unit of data that can be used by a processor to perform a single operation, and the total number of bits in a word refers to the word size or word width. In bit level parallelism, by increasing the number of word size, the number of instructions the processor should process decreases which helps to process instructions that are bigger than word width [110]. For instance, if we have an eight-bit processor and sixteen bits of instruction, then the processor needs to perform two times to finish a single operation. Therefore, it has a direct impact on the performance of system. Modern computer architectures are typically x86-64, which refers to 64-bit processor.

2. *Instruction level parallelism (ILP):* this refers to how many instructions in a computer can be processed concurrently and simultaneously. While new hardware architectures handle ILP in different types, older processors can also support this level of parallelism in a different format called speculative execution. In this type of execution, operations or instructions are not following the same order of executions in a program, and as soon as the instructions are available they can be executed, whether or not they are needed. However, this type of execution in older processors helps to improve the performance

of systems compare to sequential execution. It should be noted that developing a program to consider maximum level of ILP is a complex and difficult job that needs good comprehension of the program and target hardware architecture. Other important factors of instruction level parallelism indicate data dependency between operations/instructions and how to map instructions to target hardware architecture (scalability and portability) [111]. Moreover, different techniques (such as loop unrolling and pipelining) of instruction level parallelism for enhancing performance and throughput currently are supported by task and data levels parallelism [112].

3. *Task level parallelism:* different computations can be processed on the same or different groups of data in task level parallelism, in contrasts to data level parallelism in which same computations can be performed on the same or different groups of data [107]. In this level of parallelism tasks are divided into smaller portions called sub-tasks, each of which is assigned to a processor or thread for simultaneous execution. Cluster hardware computer architectures and multi-core computers offer task level parallelism.

4. *Data level parallelism:* this is another type of parallel computing in which the same types of instruction are performed on the same or different vectors of data. SIMD of Flynn's taxonomy is classified into this parallelism.

In addition to different levels of parallelism and programming models, Figure 5.9 presents the concept of parallelism mapped to software and hardware aspects.

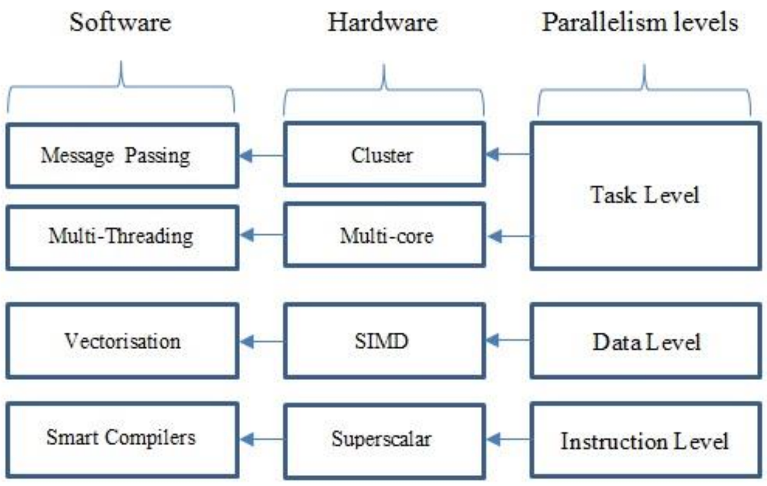


Figure 5.9: Relations between parallelism, hardware and software.

It can be seen in Figure 5.9 that cluster and multi-core computer offer task level parallelism, which supports multiple threads or cores separate and simultaneous task execution. OpenMP as a multi-threading application program interface is used in this research. SIMD refers to data level parallelism in which each instruction is executed on the same or different vector of data, and superscalar processors deliver Instruction Level Parallelism (ILP). However, any of these parallelism levels are able to utilise the underlying parallelism level (hybrid model). For instance, any node in a cluster can have multiple cores, each of which can execute multiple threads, and each thread can process SIMD instructions. This study investigates SIMD and multi-threading via data level and task level parallelism more deeply.

5.3.1 SIMD/Vectorisation

The model of SIMD execution in new hardware architecture theoretically is comparable to the vector processors in the 1980s, which could perform the same computations on one dimensional vector concurrently and simultaneously. In SIMD execution model the number of different data that can be processed by each instruction (SIMD width) varies from architecture to architecture, and usually is less than vector processors, but on the other hand, modern hardware architectures have higher Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX). For instance, GPUs normally support 1024 or 2048 bit SIMD. Instruction set refers to the computer architecture which involves data types, memory architecture, addressing modes, mathematic operators and Boolean operators. Streaming SIMD Extensions (SSE) indicates extended instructions (70 new instructions) in the x86 computer architecture that can be improved the performance of system (new processor architectures support SSE4). Advanced Vector Extensions (AVX) is extensions to the x86 instruction set computer architecture which mainly refers to the extension of SIMD width register capacity [113]. For example, SSE 128-bit can store four 32-bit single precision floating points or two 64-bit double precision values of x86 computer architectures. AVX SIMD is increased to 256-bit, and AVX2 is augmented to 512 bit.

Traditional procedural programming languages such as C and FORTRAN are scalar-based languages that were not designed to use SIMD execution model. In order to utilise SIMD execution structure, the execution model should provide either auto-vectorisation compliers which convert/transfer scalar program into a vectorised program or intrinsic/built-in functions in scalar program for specifying vectorisations explicitly. Most programmers prefer to use auto-vectorisation compilers to automatically transfer scalar code into vectorised program by fruitfully unrolling the loop to match the size of computer architecture’s SIMD width. However, in case of complex programs auto-vectorisation cannot address efficient vectorisation for obtaining maximum performance [114]. In fact, the auto-vectorisation compiler has no knowledge about the dependency of operations, how the program will be executed, or the domain problem [114]. On the other hand, finding vectorisation explicitly with intrinsic functions programming provides the best way to use SIMD execution model to obtain high performance. Intrinsic functions method assigns computer hardware instructions directly and specifies vectorisation explicitly, helping achieve high parallelism. However, this technique needs a good sympathetic SIMD execution model and computer hardware capability.

Another way to utilise SIMD execution units refers to the combination of mentioned methods which leads to the Single Program Multiple Data (SPMD) programming. SPMD is a high-level programming model in which different tasks are divided and executed on multiple processors simultaneously with different data streams [115]. In SPMD programming the whole program is developed from the perspective of separate and independent tasks to execute parallel and concurrently, which also facilitates easier auto-vectorisation. The most frequently used SMPD refers to high performance computing (HPC) for cluster computers, and multiple cores computer [116], but also shows a significant achievement for GPU NVIDIA's CUDA programming model [117].

5.3.2 Multi-Core – Multithreading

Traditionally, computer scientists believed that performance comes from hardware, thus adding more clock speed and frequency on a processor provides better performance. This idea led computer scientists to design hardware based on performance optimisation architectures for many years. However, this idea failed due to increased power and cooling needs in architectures with increased clock speeds, consequently they designed new hardware architecture based on power optimisation rather than performance optimisation by adding number of cores [118]. Modern computer hardware is designed based on the power optimisation by having multiple numbers of cores with low clock speeds, and effective support for concurrent and parallel execution on all multiple cores. New computer architectures also provide the concept of hyper-threading or simultaneous multi-threading (SMT), which allows multiple threads to execute a program or sub-program (task) separately and independently by utilising the resources of a distinct core. For instance, NVIDIA Quadro k5000 supports 192 threads per core, and usually CPUs with x86 architecture support 2 SMT. The most common model of parallel execution for multi-threaded programs refers to the fork-join model. In this model, fork indicates a master thread which makes a number of parallel threads for parallel execution in parallel regions, and when the number of threads finishes the tasks in the parallel sections, they will be synchronised (join) and just leave a master thread [84, 88]. It should be noted that the number of threads and number of parallel sections are arbitrary. The most famous application interface program that used this model was OpenMP. Figure 5.10 presents the fork-join model.

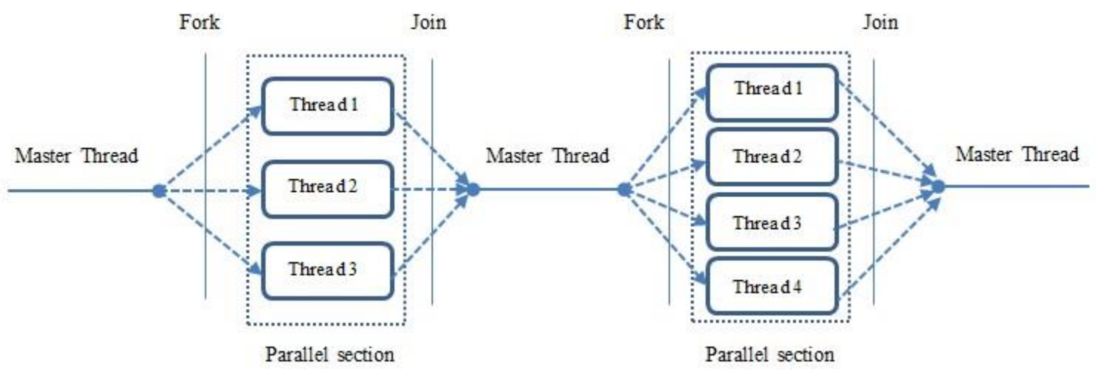
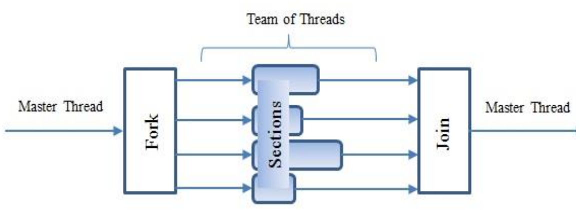


Figure 5.10: Fork - Join model.

In OpenMP, there are different techniques to execute parallel region and assigning tasks to different threads. These techniques are called work sharing constructs, and they involve: (*i*) do/for work sharing construct, this type of execution shares iterations of a loop between numbers of threads; (*ii*) sections work sharing construct with individual and separate sections, each of which is processed by a thread; and (*iii*) single work sharing construct which executes a program by single thread in the team, which is beneficial when mixing with other types of work sharing constructs. The following figures show models of do/for construct and sections construct.



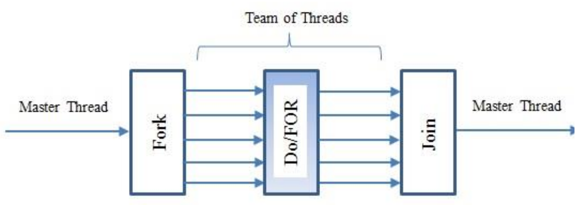


Figure 5.11: Sections work sharing construct.

Figure 5.12: Do/ For work sharing construct.

The SPMD programming model can also be implemented in multi-threading, by which all tasks are separate and distinct at compiler time, and the compiler is able to allocate tasks between threads/cores in addition to (or instead of) SIMD instruction units without impacting accuracy and efficiency. In NVIDA CUDA GPU programming model also threads are congregated together into blocks which provide some guarantees about memory access pattern and synchronisation within-block. However, it is not possible to ensure that all threads will be performed and executed by the same core. In fact, threads within the same block can share and access data but they cannot synchronise it with other threads within different blocks.

5.4 Parallel Computer Memory Architecture

Another important concept in parallel computer refers to parallel memory architectures. Parallel computer memory architectures are based on two memory organisations: physical memory architecture and programmer perspective of the memory. Physical memory architecture indicates the local physical shared memory (e.g. multiprocessors), distributed physical memory such as multicomputer, and a combination of two models, called hybrid architectures. The programmer perspective of the memory considers accessing memory in a shared address space and in distributed address spaces [103]. Consequently, there are three types of parallel computer memory architectures: (*i*) shared memory architecture, (*ii*) distributed memory architecture, and (*iii*) hybrid distributed shared memory architecture.

5.4.1 Shared Memory Architecture

In this type of architecture multiple processors or cores share the physical memory address space (also known as global memory) and data can be transferred across cores based on the global memory by accessing, reading, and writing shared variables. In this architecture, processors can be executed separately and individually, but all cores access and share the same global memory resources, thus a change in a memory address by one processor will be noticeable to other processors. The shared memory organisation can also be classified based on the way of accessing memory data, namely uniform memory access (UMA) and non-uniform access (NUMA). In UMA class, all processors are identical and have equal access time to memory. Symmetric Multiprocessor (SMP) computers, in which multiple processors are connected together and linked to the shared memory via a central bus [119], are the most famous machines using UMA architecture of shared memory. In SMPs there is no private memory, but each core/processor has its own cache hierarchy, known as cache coherent. The cache coherency helps if one processor changes or updates an address in shared memory, which causes other processors to distinguish that specific change or update. Each SMP computer has a multicore processor and different numbers of cores. Figure 5.13 illustrates the UMA of shared memory model.

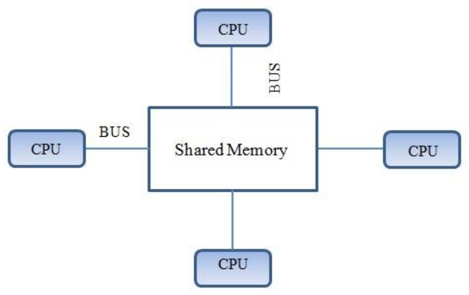


Figure 5.13: Shared Memory - UMA model.

NUMA model refers to type of shared model wherein the memory access time is based on the memory position comparative to the processor. In this model, a processor access to the own memory faster than other local memory in other processor or shared memory between cores (access time to memories is not equal among processors). In other words, when physically connecting two or more SMP computers together, there is a NUMA model and in general access to memories over link is slower than UMA model [120]. Figure 5.14 presents the NUMA model of shared memory, and Table 5.3 demonstrates the main advantages and disadvantages of shared memory architecture.

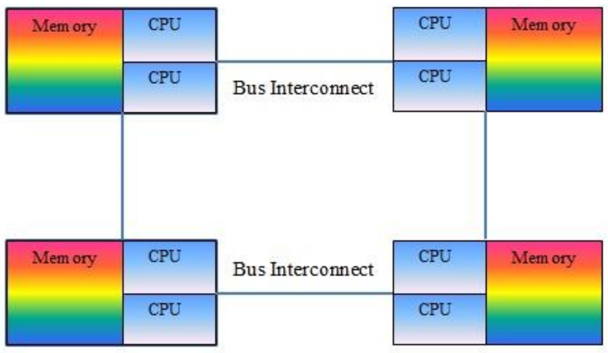


Figure 5.14: Shared Memory - NUMA model.

Advantages	• Global access to memory and address space is easy. • More user-friendly (for programmers). • Relatively faster to access memory and share data.
Disadvantages	• Scalability issue between processors and memory; by connecting more cores, more collisions occur. • More programmer accountability and responsibility for synchronisation, and having accurate data sharing in global memory.

Table 5.3: Advantages and disadvantages of shared memory architecture.

5.4.2 Distributed Memory Architecture

This model includes varying numbers of computers (nodes) which are connected to each other through network. All data sharing and data transferring between nodes are based on the network connections. A node refers to an independent computer on the network that comprises processor and local memory. Data can be saved in the local memory of one computer/node on the network or different nodes. It should be noticed that local memory for each node is private, and only the local node can access its own memory, therefore the concept of global address space between processors does not exist, and changes or updates of local memory do not impact on the other nodes' memories (i.e. there is no cache coherency). When a node requires data from another node's local memory, it should send a request to that specific node via a network. This process can be performed by a message passing programming model, which is based on communication between computers. The most common interface for this model refers to Message Passing Interface (MPI). Figure 5.15 shows distributed memory model, and Table 5.4 demonstrates its main advantages and disadvantages.

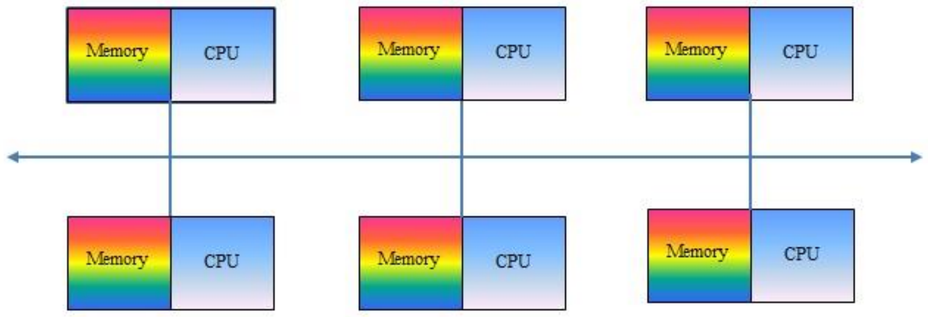


Figure 5.15: Distributed memory architecture.

Advantages	• Each node on the network can access self-data rapidly, without any overhead. • There is no scalability issue between memory and processors.
Disadvantages	• From the programmer's perspective, the developer is accountable for the whole process of communication between nodes and data transferring. • Non-uniform memory access times take longer to access and transfer data (also depends on the network bandwidth).

Table 5.4: Advantages and disadvantages of distributed memory architecture.

5.4.3 Hybrid Distributed – Shared Memory Architecture

This model is a combination of shared memory architecture and distributed memory architecture, sharing the criteria of both models. In this architecture, shared memory section can involve a SMP computer or a graphic processing unit (GPU) machine. On the other hand, distributed memory refers to the networking of different number of SMP computers or GPUs, each of which has a private memory. Consequently, network communications are needed for data transferring (message passing). It should be noted that 'shared' in this regard refers to the sharing of address space, not of the single main (centralised) memory [121]. This type of architecture is highly demanded for high-performance computing, and it is prominent in the next generation of parallel computing models. The main advantages of the hybrid model concern scalability improvement, while its disadvantages are the difficulty and complexity of programming. Figure 5.16 presents the hybrid distributed – shared memory model.

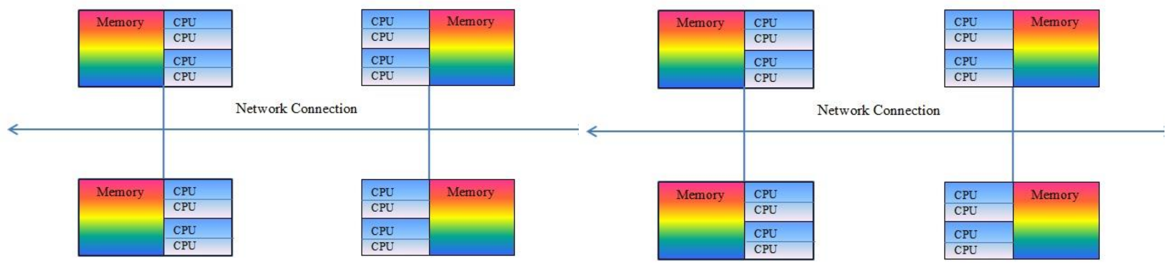


Figure 5.16: Hybrid architecture.

5.5 Summary

New hardware architectures support different types of parallelism relative to programming and memory models. Several scientific programs (such as Lattice Boltzmann, Molecular Dynamics and Pipelined Wavefront) used different parallelism models, but utilising them accurately and effectively for complex programs (e.g. cell dynamics simulation) in lower levels remains problematic due to compound data dependency and a lack of memory access. This chapter presented the concept of concurrency and parallelism to understand the differences between them and how identifying concurrency can help to define the algorithm strategy of parallel program. Task and data level parallelism were explained to justify their use in this research, along with the fork-join model in OpenMP for multi-threading implementation.

CHAPTER SIX

6 GPU Many-Core Accelerator

Utilising multi-core GPU accelerator technology in high performance computing has greatly increased over the last few years [122]. Since the late 1980s companies such as Microsoft started to create graphical operating systems based on 2D display accelerators that delivered hardware-assisted bitmap operations to help in the display and usability of graphical operating systems [123]. In 1992, Silicon Graphics released the programming interface OpenGL Library for writing 3D graphics applications. Later, other companies such as NVIDIA and ATI started releasing graphics accelerators that were good enough to attract different scientific users. In the early 2000s, GPUs were designed to produce a colour for each pixel on the screen by utilizing programmable arithmetic units known as pixel shaders, which use an (X, Y) coordinate on the screen combined with some other additional information, such as texture coordinates and input colours to compute a final colour. Because additional information is totally controlled by the programmer and the arithmetic being performed on the input colours, it is possible to compute any data rather than input colours [124], thereby enabling GPU processing for non-graphical purposes which is called general purpose computing on graphic processing unit (GPGPU). By supporting fully and effectively programmable pipelines, GPGPU allows high performance computing developers to benefit from GPUs' high parallelism in different scientific programs and engineering applications [125]. This chapter mainly demonstrates the programming model and architecture of the GPU used in this thesis.

6.1 GPU Architecture Evolution

The demand for real time, high quality graphics in computer systems has been the motivation and inspiration of graphic processors. The evolution of GPU started from a fixed function pipelines to micro coded processors, from micro coded processors to programmable processors, and from programmable processors to scalable parallel processors [126]. Scalable parallel processors have a large number of GPU transistors providing high parallelism and performance. To understand the architecture evolution of GPU, it is important to consider the concept of GPU in terms of the graphics pipeline which refers to several inputs (vertices of triangles) which can be executed different vertex operations and instructions, such as lighting and spatial transformations to generate scene. Scene indicates the creation of 2D image that can include textures transformed into pixels to produce the final image [127]. This process is known as graphics pipeline which can be programmed by different application program interfaces. Figure

6.1 illustrates NVIDIA GeForce 6 series architecture that involves different levels of graphics pipeline process [128].

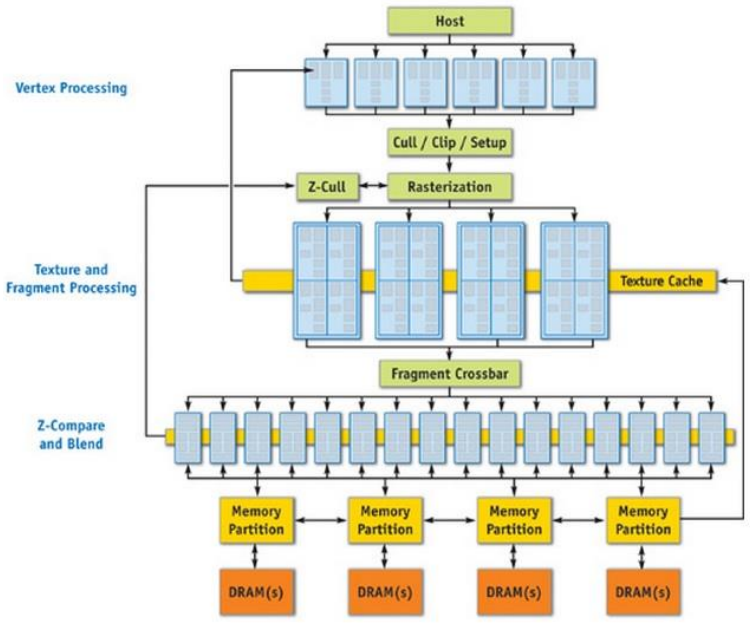


Figure 6.1: NVIDIA GeForce 6 series architecture [128].

The salient feature of this model is the inequity of weight in the process between vertex and texture processors (load balancing issue). For instance, primitive shapes utilise small amounts of the vertex processors' throughput, but a huge amount of the texture and fragment processors, which causes inefficient usage of resources and wastage of computing power. To solve this problem, in 2006 NVIDIA GeForce introduced 8 series GPU, which includes a unified shader model that provides set of cores that can be processed and which support any level of the graphics pipeline [129]. To this end, unified architecture provides graphics programming and enables programming for non-graphical purposes. The most famous APIs which took the benefits of unified shader architecture/model are: (*i*) OpenCL [130], (*ii*) DirectX 11, and (*iii*) NVIDIA CUDA GPU. Figure 6.2 presents CPU and GPU architectures [131, 17].

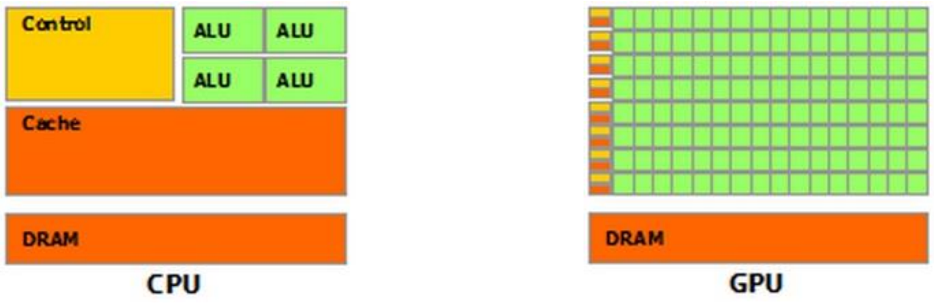


Figure 6.2: CPU and GPU architectures.

According to Figure 6.2, it is clear that GPUs with more processing threads/cores is more suitable for high performance computing and appropriate to execute computations with huge dataset.

Another important factor in the evolution of GPU is the development of microarchitectures with different compute compatibility. In general, compute compatibility is currently classified into four main classes, each of which has sub-versions [131]:

1. *Tesla microarchitecture with compute compatibility 1.x:* this was the first microarchitecture introduced by NVidia GPUs, which supported only fundamental and basic calculations. In this class of compatibility, there was no cache between cores and memory, no support of 3D grid of thread blocks, and no support of dynamic parallelism. Double precision floating point number was added in sub-version 1.3 of this architecture.

2. *Fermi microarchitecture with compute compatibility 2.x:* the fermi model was the foremost leap in computational GPU microarchitecture and improved many important areas of compatibility compared to former architectures. In this microarchitecture, two classes of cache memory are developed (between processing cores and memory), thus double precision performance improved. It also enhanced atomic operations (read, modify, write), supported high level programming languages, enabled error correcting code facility and 64 bit unified addressing, and increased computational resources such as registers and number of cores [132].

3. *Kepler microarchitecture with compute compatibility 3.x:* in mid-2010, NVIDIA presented a new microarchitecture for GPU called Kepler that involved three important new features: increased number of streaming multiprocessors (SMXs or SMs), Hyper-Q and dynamic parallelism [133]. Hype-Q increased the performance of the GPU by allowing several CPU cores to simultaneously use a single GPU core. Dynamic parallelism enhances GPU performance by allowing it to schedule and execute jobs without including CPU resources. In fact, NVIDIA in Kepler microarchitecture is more targeted toward programmability and energy saving [134].

4. *Maxwell microarchitecture with compute compatibility 5.x:* Maxwell micromodel is the newest development of NVIDIA GPU. This architecture generally is same as the Kepler model, with more improvements in resources (number of registers per multiprocessors and number of resident blocks per processors).

In this study a commodity NVidia Quadro K5000 with Kepler microarchitecture (compute compatibility 3.0) is used. The Quadro K5000 GPU is based on the first version/generation GK104GL of Kepler microarchitecture which does not support Hyper-Q and dynamic parallelism. The CPU to GPU copy (memcopy) bandwidth and GPU to CPU copy (memcopy) bandwidth were measured as 5868.3 MB/s and 6532.2 MB/s, by utilising a benchmarking program provided in the CUDA SDK. Figures 6.3 and 6.4 show the model of two key features of Kepler microarchitecture compared to Fermi microarchitecture [135].

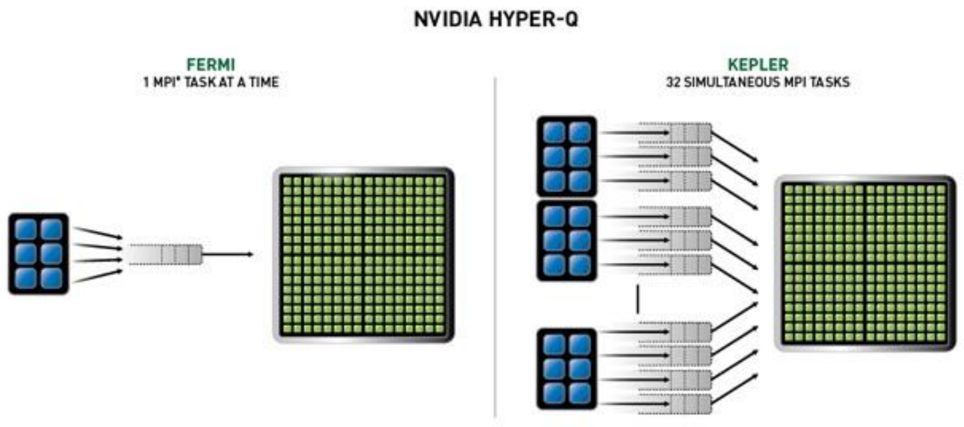


Figure 6.3: Hyper-Q model of Kepler microarchitecture [135].

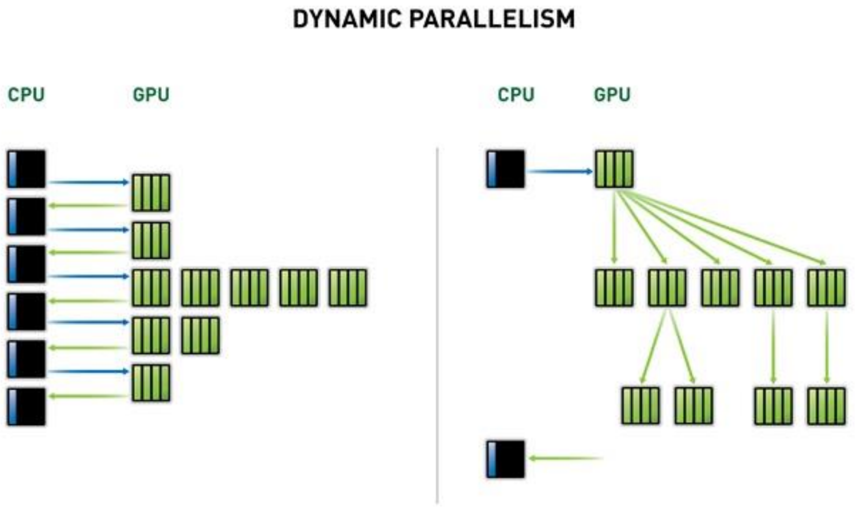


Figure 6.4: Dynamic parallelism model of Kepler microarchitecture [135].

6.2 GPU Memory Architecture

GPU architecture has different types of memory to support the requirements and needs of programming. In general, GPU memories are classified into two groups, on and off chip memories. On chip memory involves registers and shared memory/L1 cache while off chip memory comprises local memory, constant memory, texture memory, and global memory. Obviously it is fundamentally important to achieve high performance in applications using the available memories [136]. Figure 6.5 illustrates the different types of memory in GUP and relationship between them and CPU.

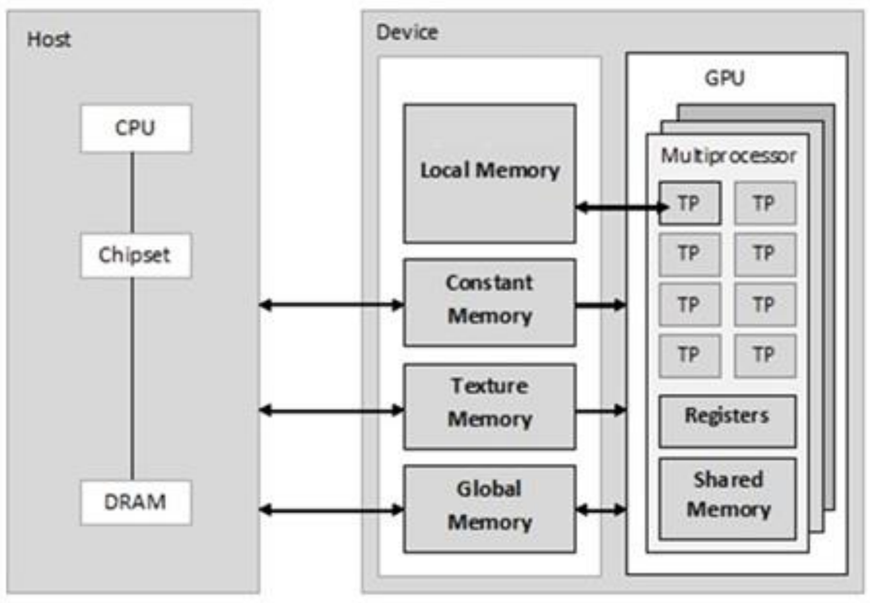


Figure 6.5: GPU memory model.

Kepler generation NVIDIA GPUs comprise six different types of memory that can be utilised in programming. It should be noted that texture and constant memories are read-only memories for GPU, but CPU can read and write on global, texture and constant memories.

1. *Global memory (DRAM):* also known as device RAM, it is the biggest memory on the GPU, residing off it. The main features of this type memory are its comparatively slow in latency (400 – 600 clock cycles) [131], it saves data in global memory accessible to all threads, and both GPU and CPU can read and write in global memory. The other important point of this memory is the L2 cache, which provides a buffer to global memory to increase access time [124, 131].

2. *Shared memory/L1 cache:* it is on chip memory with higher bandwidth and lower latency compared to global memory. Each streaming multiprocessor comprises a 64KB shared memory/L1 cache accessible to all streaming processors or cores within a

processor. In the Fermi architecture of GPU, L1 cache was a consistent automatic cache for global memory, but in Kepler architecture L1 cache is utilized for local memory access such as registers on GPU. Shared memory capacity can be divided with L1 cache, 48KB shared memory / 16KB L1 cache or vice versa. Kepler model has added a new feature to this division, 32KB shared memory and 32KB L1 cache, which can be useful when L1 cache requires more than 16KB but less than 48KB capacity of shared memory per core [137].

3. *Constant memory:* generally, it is used for saving data which will not modify and change over the execution of a kernel. This memory is read only for GPU, but CPU can read and write on constant memory. Utilizing constant memory in program rather than global memory reduces the latency and decreases the need for memory bandwidth.

4. *Texture memory:* texture memory is the other type of cache to DRAM that is read only, the same as constant memory for GPU. Texture memory is mainly used by the texture processing for assigning 2D scene onto a 3D surface, and for rendering 3D images. Utilizing texture memory decreases memory traffic and improves the performance.

5. *Registers:* it is another on chip and fast memory in which GPU can read and write. Usually the register uses zero clock cycles per instruction, but sometimes more clock cycles can happen because of race conditions and read/write dependencies [138].

6. *Local memory:* local memory is another off chip memory which can be used as an abstraction of global memory to avoid high latency. Automatic variables with big structures or arrays normally are assigned in local memory to improve the performance of application.

6.3 Compute Unified Device Architecture (CUDA) Programming

CUDA as a programming architecture contributes to GPGPU technology by supporting heterogeneous data parallel computing and solving time-consumption and expensive computations issues. CUDA architecture hardware consists of a group of streaming multiprocessors (SMs), each of which comprises a set of streaming processors (SPs), also known as cores, register memory, shared memory, read-only texture and constant cache memory [132]. In CUDA programming model, the computing system is divided into two sections: the host section, which is CPU; and the device section, which is the GPU. The host section calculates low volume or non-parallelised data, while the device section computes a large amount of data parallelism. Device codes on GPU are executed in single instruction multiple data (SIMD) model, while each stream multiprocessor accepts a set of single instruction multiple thread (SIMT) to assign each thread block on one stream processor or core.

6.3.1 CUDA Thread Hierarchy Model

Threads in CUDA model are allocated into a hierarchy level, thread blocks and grid of blocks. When CUDA kernel function is called, the execution process automatically moves from host-CPU to a device-GPU, and based on the hardware specification of GPU architecture sufficient resources such as number of threads in blocks and the size of grid (number of blocks in grids) are allocated to execute tasks in parallel. In this model, entire threads in a block access the same block index (blockIdx) in a grid, each of which thread has its own thread index (threadIdx) in a block. Both blockIdx and threadIdx are built-in variables in a kernel function. Based on these variables the coordinates of the thread can be identified. GridDim and blockDim are also important pre-initialised variables; the former indicates the total number of blocks in a grid or its dimensions, while the latter is the number of threads in a block or dimensions of a block. Based on the compatibility and architecture of GPU, grid of blocks (gridDim) can be two dimensions or three dimensions. For instance, GPUs with Tesla microarchitecture and 1.x compatibility can have two dimensional grid of blocks, but Fermi and Kepler microarchitectures support three dimensional. Also, blocks of threads according to the needs of programme can be one, two or three dimensional arrays of threads. Therefore, to determine the index of threads in 3D grid and make sure all data in 3D array is covered by a unique and identical thread, the following equations/patterns based on global index values can be used [139]:

$$\begin{aligned} x &= blockIdx.x * blockDim.x + threadIdx.x, \\ y &= blockIdx.y * blockDim.y + threadIdx.y, \\ z &= blockIdx.z * blockDim.z + threadIdx.z. \end{aligned} \tag{6.1}$$

In CUDA hierarchy model each thread block assigns to streaming multiprocessors and the execution of each thread on core/SP is totally independent; when all the threads have finished their execution, the following grid of the thread block will be terminated [140]. In fact, each CUDA core performs integer instruction or a floating point per clock for a thread. Figure 6.6 shows an example of CUDA grid hierarchy model [141].

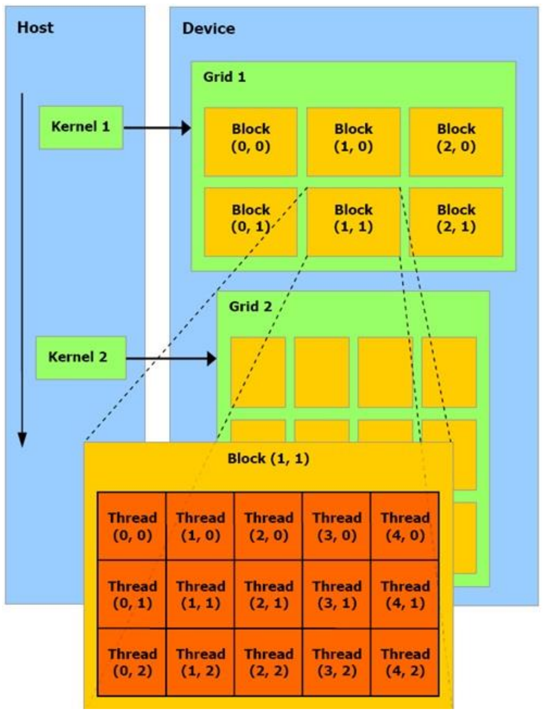


Figure 6.6: Example of two dimensional of CUDA hierarchy model [141].

Threads in CUDA hierarchy model can access data from different types of memory available on GPU, but also each thread has own registers memory that cannot be shared with other threads. A group of threads in a same block can access shared memory which is fast but limited in capacity. And blocks in a grid can share data through global memory. Figure 6.7 presents GPU memory chart based on CUDA thread hierarchy model.

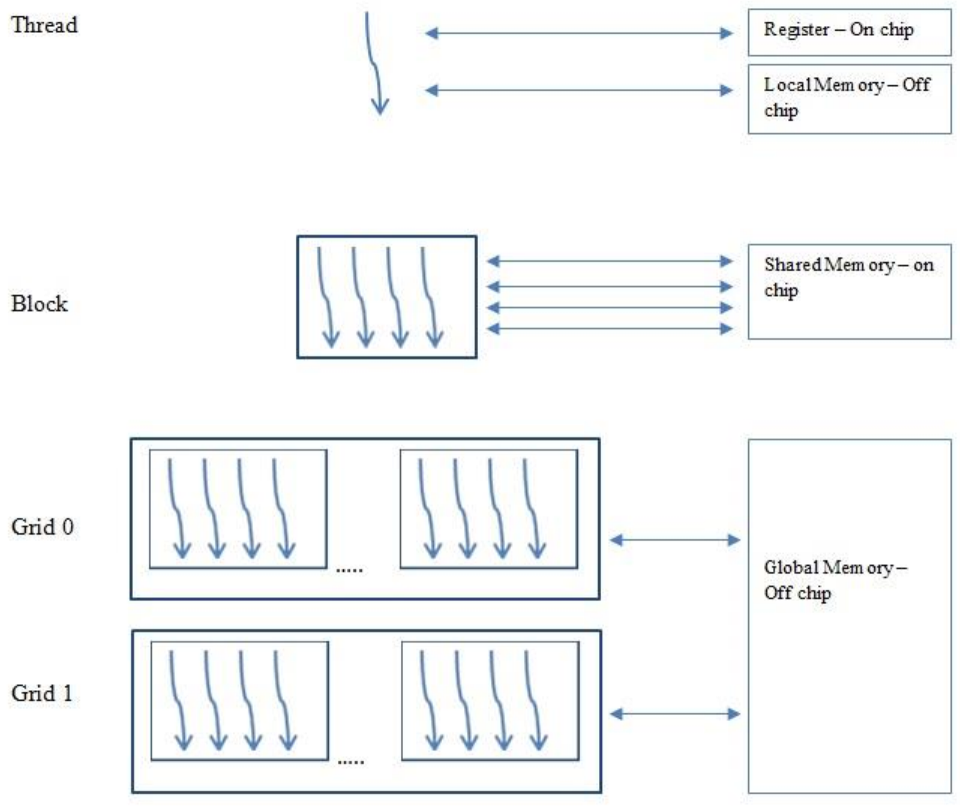


Figure 6.7: GPU memory hierarchy: Threads share local memory and each thread has own register. Threads within block share shared memory. Grids may access global memory.

Based on the nature of a program, 1D, 2D, or 3D thread hierarchy model can be used. Usually it is more useful and convenient to utilise the same dimensions for grid and block for processing data elements, however it is possible to have a grid with higher dimensionality than its block and vice versa. As mentioned earlier, threads are gathered into blocks, and a block based on the computability of GPU can hold a maximum of 512 or 1024 threads. For instance, there is a product matrix of size 76×62 (76 in x direction and 62 in y direction). In order to map threads into this 2D product matrix, it can use 5×4 grid of thread blocks with 16×16 blocks to cover the whole of the product matrix elements. Figure 6.8 illustrates the block utilised to process a product matrix of size 76×62 [139].

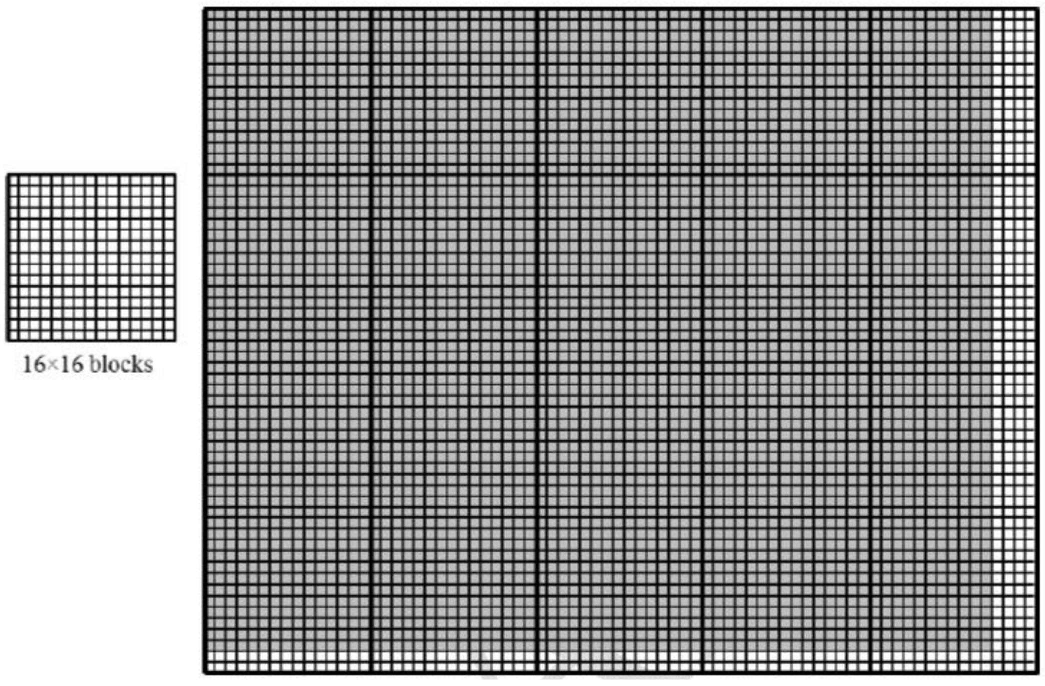


Figure 6.8: A 5×4 grid of 16×16 blocks to compute a 76×62 system size [139].

In Figure 6.8, the heavy lines distinguish the block boundaries, and shaded zone presents computed/processed area by threads. According to the kernel configuration parameters, in total there are 256 threads per block and 20 blocks per grid. Therefore, 256×20 =5120 threads exist to compute 76×62 = 4712 of product matrix. This number of threads is more than the number of system size, consequently threads will assign themselves to the matrix and the rest of threads that are outside the system will terminate. As can be seen clearly, there are four extra threads in the x direction and two extra threads in the y direction.

Another important benefit of CUDA programming model based on Fermi and Kepler GPU architecture is warping, which is useful to optimise the performance of CUDA program. In CUDA model, a block is divided into 32 threads called a warp and the execution of the thread block is based on the warp execution. Warp plays an important role when parallel execution (instruction processed by the threads in a warp) is waiting for the outcome of former calculation with long latency; in this case another wrap (32 threads units) is automatically chosen for execution which is no more waiting for results [140]. This process avoids long latency time by choosing another warp which is called latency tolerance or latency hiding. In fact, by providing enough warps, GPU can easily find a warp to execute at any time. In addition, it should be noticed that latency hiding does not make any overhead time (idle time) into total execution time of program, and provides better performance in application.

6.3.2 CUDA Synchronisation

As mentioned in chapter 5, synchronisation in parallel computing means the coordination of different parallel tasks in real time. In CUDA programming model, threads within a block can be coordinated or synchronised in the kernel by using synchronisation function syncthreads(). In this situation, all threads within a block will be waited at the barrier location until all other threads within a block arrive to barrier location. In fact, using synchronisation function ensures that all threads in a block have finished a part of their task before they can start the next part of the computation task. The other synchronisation function in CUDA programming refers to the cudaDeviceSynchronisation() which can be used to make sure all kernels have finished and completed their job. Mostly, syncthreads() synchronisation function is used when shared memory is implemented [142]. Figure 6.9 shows an example of threads synchronisation.

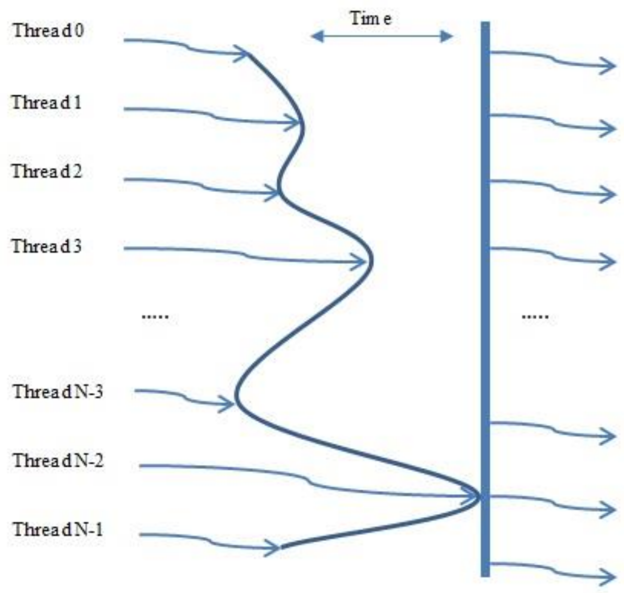


Figure 6.9: An example of threads synchronisation.

Another important factor in synchronisation is overhead time (not useful time). In CUDA run time model, to prevent overhead time threads within a block should execute in close time with each other. CUDA run time model does this by mapping enough resources to entire threads in a block as a unit [139]. In this model, until the run time system has not employed enough resources required for entire threads within a block, a block as a unit will not start to compute the task. In fact, CUDA run time system assigns the same resources for all threads in a block and avoids long overhead time during synchronisation, and makes the time execution of threads closer to each other. This factor of CUDA run time provides transparent scalability between blocks of kernels. Transparent scalability refers to the execution of blocks in any order which is related to each other without waiting for each other [139], or in other words threads in different blocks do not need to have barrier synchronisation. Figure 6.10 presents transparent synchronisation between thread blocks.

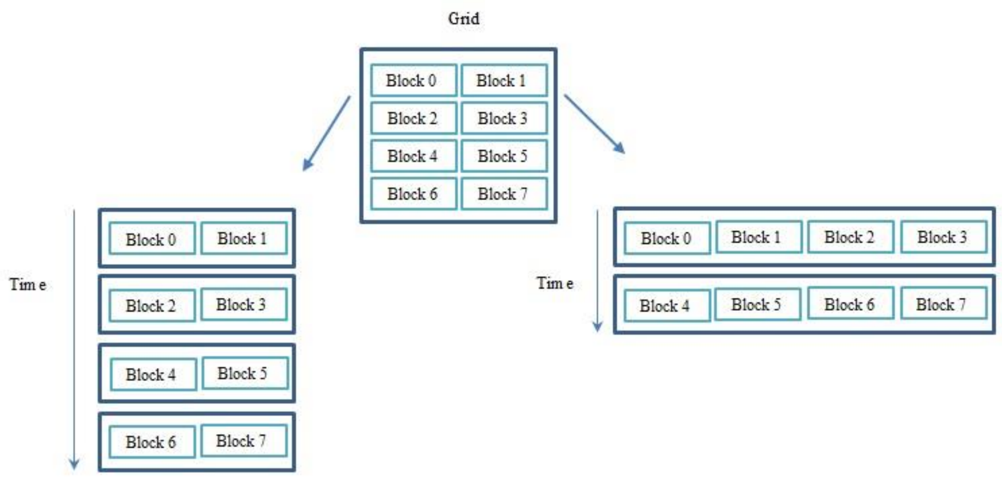


Figure 6.10: Transparent scalability example for CUDA run-time system.

As shown in Figure 6.10, the execution process based on the needs and requirements can be scalable. For instance, when few execution resources needed, a kernel with less blocks at the same time can be used, similar to the left side grid of Figure 6.10 (two blocks simultaneously). On the other hand, when large execution resources are required, a kernel with more blocks can be utilised, similar to the right side grid of Figure 6.10 (four blocks simultaneously). To this end, transparent scalability provides the capability to execute a program with different speeds, and to develop different range of applications according to the usability, power, and performance requirements.

6.3.3 Efficient Implementation of GPU Code

Based on the highly parallel nature of GPUs architecture and different types of memory on GPU, there are different strategies to develop efficient GPU code, and to achieve high performance application. One of the well-known method/strategy for implementing efficient code has been recommended by Cohen & Molemaker [143], who stated that efficient results necessitate implementation of as much of the code as possible on the GPU. Although this method can be usable and valid, problematic issues are faced when trying to transfer the existing CPU code to GPU. Transferring complex code is very difficult and even in the best situation only a small portion of code can be transferred. Alternatively, Frigaard [144] introduced a technique to accelerate and optimise existing CPU code by finding computationally expensive and time consuming parts of program and transfer them to the accelerator (GPU). Both strategies recommended first altering the algorithm of baseline code into a more efficient algorithm by removing iteration paths (loops) and synchronisation points, then optimising the performance of code. In general, performance optimisation comprises three strategies [131, 135]: (1) maximise parallel execution to approach high utilisation; (2) optimisation of memory usage to achieve high memory throughput; and (3) optimisation of instruction usage to achieve high instruction throughput, as described below.

1. *Maximise parallel execution:* this refers to the design of application that can be utilised as much parallelism as possible and efficiently assigns this parallelism to different sections of program to prevent idle time (overhead time). Maximise utilisation involves three levels:
    a. Application level: this is a high level utilisation of CUDA programming model that refers to maximise parallel execution between the host/CPU, the device/GPU, and the PCI bus connector between host and device. This level of utilisation can be made by assigning little or non-parallelised workload to the host and large amount of data parallelism workload to the device.
    b. Device level: this indicates to a lower level utilisation of programming model. To achieve maximise utilisation at this level, it should maximise parallel execution between the multiprocessors of a device by keeping them busy most of the time.
    c. Multiprocessors level: as with the previous level this is a lower level that should be considered to maximise parallel execution between different functional units in a multiprocessor. Different functional units in multiprocessors refer to the threads level parallelism (SIMT model) which is indicated to the number of warps within multiprocessor. Indeed, this part has direct link to latency tolerance or latency hiding. As discussed in section 6.3.2, the total number of clock cycles of warp takes to be ready and to perform next portion of code is called latency. Therefore, maximum utilisation between different functional units in a multiprocessor will be achieved when latency is totally hidden.

2. *Optimisation of memory usage to achieve high memory throughput:* in general maximising memory throughput refers to minimising data transfers between the host and the device with low bandwidth, maximise usage of on chip memories, and optimally memory access.

3. *Optimisation of instruction usage to achieve high instruction throughput:* to achieve full instruction throughput should be considered the following key points:

a. Reduce the utilisation of arithmetic instruction with low throughput, such as using mathematical intrinsic functions, and single precision floating point instead of double precision floating point.
b. Minimising the number of instructions such as number of synchronisation points. It should be noticed that here throughput refers to the number of operations per clock cycle per multiprocessor. For instance, a warp with 32 threads has one instruction to correspond to 32 operations. Thus, the instruction throughput for N operations per clock cycle is equal to N/32 instructions per clock cycle. In addition, to calculate the whole throughput for the GPU device should multiply total number of multiprocessor to the throughput for each multiprocessor.

By applying these techniques Simek et al. [145] achieved 8x-9x speedup for modelling and simulating the atmospheric equations, and Bell & Garland presented a speedup of 1.5x compared with baseline GPU code for matrix multiplications [146].

6.4 Benchmark Platforms

The study in this research makes use of different hardware architectures for CPU and GPU based on the latest hardware available at the time of study. In fact, because of the fast development of hardware devices, incompatibility between hardware platforms and programming languages model, and limitation of access to high performance computers (large scale computers), only two different hardware architectures are considered.

In both hardware platforms, the maximum rate of data transfer for memory bandwidth is mentioned in gigabytes per second (GB/s), power is calculated based on the thermal design power (TDP) in Watts, and performance is reported based on the maximum GFLOP/s in single precision for CPU and GPU. It should be noted that the maximum rates of performance and memory bandwidth are theoretical peaks that cannot be achieved in reality. In addition, the total number of compute units (cores) and processing elements with support of Simultaneous Multi-Threading (SMT) or hyper-threading is reported. The following tables present the hardware specification of the CPU and GPU used in this research.

CPU	Intel Xeon Processor E5-2420
Number of cores	6
Number of threads (processing unit) per core	2
Total number of threads/processing units	12
Peak GFLOP/s	91.2
Max. memory bandwidth (GB/s)	32
Instruction set	x86 - 64 - bit
Instruction set extensions	AVX
Processor base frequency	1.9 GHz
Processor Max. Turbo frequency	2.4 GHz
Thermal design power (TDP) Watts	95 W
Memory Types	DDR3
Total capacity of Memory (RAM)	12 GB
Error Correcting Code (ECC) Memory	Yes

Table 6.1: Hardware specifications of the CPU and the Memory.

GPU	NVIDA QUADRO K5000
GPU capability	3.0
GPU micro-architecture	Kepler GK104GL
Number of streaming multiprocessors (SMs)	8
Number of processing units per SM	192
Total number of processing unit	1536
Peak GFLOP/s – Single precision	2150
Max. memory bandwidth (GB/s)	173
Memory bus width (bits)	256
Memory clock rate (MHz)	2700
GPU memory (GB)	4
GPU base frequency (GHz)	0.71
Warp size	32
Max. number of warps per multiprocessor	64
Max. number of blocks per multiprocessor	16
Max. number of threads per block	1024
Max. number of threads per multiprocessor	2048
Max. dimension size of a thread block (x, y, z)	(1024, 1024, 64)
Thermal design power (TDP) Watts	195 W

Table 6.2: Hardware specifications of NVIDIA GPU.

6.5 Summary

This chapter investigated GPU as many-core accelerator by considering different aspects of memory architecture and CUDA programming model. In CUDA programming model, the kernel execution specifies the dimensions of a grid and thread blocks. Once the kernel is called by a host, a grid will be launched and the thread blocks within a grid will be mapped to streaming multiprocessors (SMs) based on the transparent scalability of CUDA. In addition, thread blocks execution is further divided into warp execution which helps to avoid long latency by providing high occupancy for each streaming multiprocessor. The main strategies of optimising performance of CUDA application are presented by focusing on optimisation of memory usage, instruction usage and maximising parallel execution. Finally, GPUs as many-core architectures are currently investigated and reviewed as one ideal choice based on the evidenced by their prominence and importance in high ranking TOP500 supercomputers for different scientific research fields.

7 Cell Dynamic Simulation on CPU and GPU

As discussed in chapter 2, cell dynamic simulation as a coarse-grained discretisation method can be used to investigate mesoscopic structure formation and dynamic behaviour of diblock copolymers [18, 43]. Although the CDS method compared to the other techniques such as self-consistent field theory (SCFT) [147, 148] and theoretically informed coarse-grained (TICG) simulation [149, 150] is more scalable and reasonably fast, however CDS is still a computationally expensive scheme for traditional single processor computers. Hence, the main drawback for cell dynamic simulation is that its computations are time-consuming and expensive due to two fundamental constraints: the time steps and experimental scale size. These limitations have direct effects on the simulation results. To overcome these problems and make a connection between simulation results and experiments, a new parallel computational model is needed that can be executed on a multi-core device. This chapter presents the implementation of efficient CDS method on multi-core CPU and many-core GPU, demonstrates the results based on the proposed parallel algorithm on CPU and GPU and evaluates the results in terms of execution time and speed.

7.1 Optimisation of CPU Baseline Cell Dynamic Simulation

CDS time evolution of an order parameter is performed on a cellular system based on two mechanisms: (*i*) short and long range interaction between particles; and (*ii*) cell connectivity for diffusive dynamics due to order parameter differences in neighbouring cells [33]. Cell dynamic simulation specifies the group of neighbouring points by dividing the whole domain into cells and calculating them by isotropised discrete Laplacian excluding for the centre cell $\langle\langle X\rangle\rangle - X$ with respect to the time step. Therefore, according to chapter 2 the calculation of CDS comprises of five main steps: (*i*) calculations of periodic boundary conditions (PBCs); (*ii*) calculations of first isotropised discrete Laplacian; (*iii*) calculations of map function and free energy functional; (*iv*) calculations of second isotropised discrete Laplacian of free energy functional; and (*v*) calculation of time evolution of the order parameters $\psi(t+1, r)$. Figure 7.1 illustrates an example of Laplacian nearest neighbours, with modification [18].

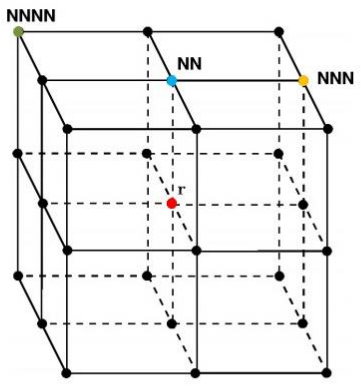


Figure 7.1: An example of Laplacian, where (●) NN presents nearest neighbours, (●) NNN next nearest neighbours, and (●) NNNN next-next nearest neighbours [18].

As mentioned in chapter 5, the method of transferring an algorithm from scalar development into a vector process which can be executed a single instruction on multiple data simultaneously is called vectorisation. The optimisation of CDS sequential algorithm for x86 computer architectures based on the vectorisation and AVX, SSE4 instruction set of SIMD has been investigated in this section.

The main optimisation challenges for the CDS scalar based implementation are: (*i*) difficulty in vectorisation due to the dependency (data dependency, control dependency or loop-nest dependency) in the CDS baseline code, which prevents vectorisations; and (*ii*) memory layout and access pattern (non-contiguous memory access), which cause the usage of expensive gather and scatter operations. To overcome the challenges mentioned, the following points need to be considered:

1. *Vectorisation:* in the CDS method, time evolution of order parameter and map function calculations can be auto-vectorised based on the Intel's C compiler with some additional support. Auto-vectorisation is achieved by unrolling a procedural's innermost loop a number of times to fit with the SIMD width of hardware to optimise the program. In the CDS baseline code, auto-vectorisation is performed with external intervention by considering compiler directives (#pragma ivdep and #pragma vector), to satisfy the potential control dependency in array updates. Each of the calculation values is unique and there is no overlapping in the same region of memory, thus the compiler directive can ignore data dependency in array updates. Due to the data dependency exists in the other CDS calculation (such as periodic boundary conditions), implicit vectorisation cannot be performed. Consequently, it is necessary to consider explicit vectorisation and parallelisation to solve the data dependency.

2. *Access pattern and memory layout:* the other issue of the CDS optimisation refers to non-contiguous memory access due to the indirect addressing (non-unit stride) in loops.

Non-contiguous memory access by using multiple numbers of instructions to load data increases the number of scalar gather and scatter operations, which obliquely transpose between Array of Structure (AoS) and Structure of Array (SoA) layouts, and decrease SIMD efficiencies (vector performance). The reader is reminded that the gather and scatter operations in vector processors refer to the loading vector indexed and storing vector indexed in non-contiguous way. Therefore, to reduce the overhead of gathers and scatters and to improve performance, two alternative ways of hand-vectorised are implemented: (*i*) considering SoA arrangement to reduce indirect addressing; and (*ii*) considering data alignment based on the AVX instruction set. Changing array structure to SoA helps to have arrangement of the unit-stride memory access which gives more effective vectorisation, and have a high locality of reference (specifically sequential locality). Locality of reference indicates memory locations which are regularly used and accessed. Having a good locality reference improves the performance by decreasing the number of data element access in memory. In addition, data alignment by aligning the data at a memory address with the same size or multiple of the word width (unit of data which can be handled by an instruction set) reduces the overhead of memory access, improves the performance of the system and makes the vectorisation compiler easier. In the CDS optimisation, 32 byte boundaries of data alignment based on AVX instruction set (256 bit) is used. Figure 7.2 shows a comparison of direct/stride and indirect access code for a simple loop in C language.

```
For ( int i = 0; i < 200; i+=3)
  {
   b[i] += a[i] * d[i];
  }
```

a) Stride=3

```
For ( int i = 0; i < 200; i+=3)
  {
   b[i] += a[i] * d[index [i]];
  }
```

b) Non-unit stride

Figure 7.2: Comparison of stride and indirect addressing of d by using index array.

7.1.1 Experimental Setup and Performance Results

One of the main techniques to obtain a high performance on computer architecture is to use a single precision instead of double precision floating-point if possible. Since the SIMD instruction units are 2x bigger/wider for single than for double precision, it is possible to achieve normally 2x better performance. This study considers utilisation of single precision floating-point to get maximum performance and to have a fair and reasonable comparison between CPU and GPU. In addition, to ensure that the C implementation of baseline code is strong enough for further optimisations, the comparison (based on the execution time) between C and FORTRAN90 implementations was done without considering any optimisations.

Table 7.1 presents the system specification and configuration used for executing the experiments in this study.

Specification	**Intel Xeon Processor E5-2420**
Double Precision GFLOP/s	91.2
Single Precision GFLOP/s	182.4
L1 / L2 / L3 Cache (KB)	32 / 256 / 15360
Clock (GHz)	1.9
Configuration	
Operating System	Linux - OpenSUSE 12.3
Linux Kernel Version	3.7.10.-1.1
Hyper-Threading Supported	Yes
Compiler Version	Intel (ICC - IFORT) 15.0.3
Compiler Flags	-O3 -ipo -fp-modelprecise -no-prec-div

Table 7.1: Specification and configuration of CPU for CDS optimisation and non-optimisation results.

The theoretical peak performance or floating-point throughput for CPUs can be calculated by the following equation:

$$Performance_{Theoretical-peak} = CPU_{Clock-speed} \times CPU_{Cores} \times CPU_{Instruction-per-cycle} \quad (7.1)$$

Where $CPU_{Clock-speed}$ indicates the multiprocessor frequency, CPU_{Cores} is total number of cores, and $CPU_{instruction-per-cycle}$ refers to the width of instruction set (SSE4-128 bit or AVX-256 bit) and the number of operations per instruction. The reader is reminded that the Intel Xeon processor is based on the Sandy-Bridge architecture which can execute two operations (addition, multiplication) per cycle, and each AVX SIMD instruction set extension can contain eight single-precision or four double-precision floating points [151].

Furthermore, to present the scalability and performance of the optimised implementation, the results of experiments for non-optimised and optimised code based on the SSE4 (128 bit) and AVX (256 bit) SIMD instruction set are demonstrated. The performance difference between AVX and SSE4 is due to the wider SIMD width registers, a new Vector Extension (VEX) which performs addition operations with greater ease and support of three operands which decrease register pressure by not changing the main source operands (no-destructive source operands) [151].

Table 7.2 illustrates the loop cost of each calculation of CDS optimisation based on 128 bit and 256 bit SIMD, and speedup versus scalar implementation on the same computer. Loop cost indicates the number of clock cycles taken to execute an instruction of one loop iteration. The amount of loop cost can be used to predict the likelihood of a performance improvement in the speed and consequently in vectorised loop. Table 7.3 shows a breakdown of the number of operations in the loop for each CDS calculation based on 128 and 256 bit SIMD. In case of 256 bit SIMD, the numbers of medium and heavy weights vector instructions are reduced due to the data alignment with a 32 byte boundary. It should be noted that according to the Intel compiler vectorisation report, vector operations do not have the same cost in terms of cycles; therefore, they are divided into three different categories from the lowest cost (light) to the highest cost (heavy) in terms of clock cycles.

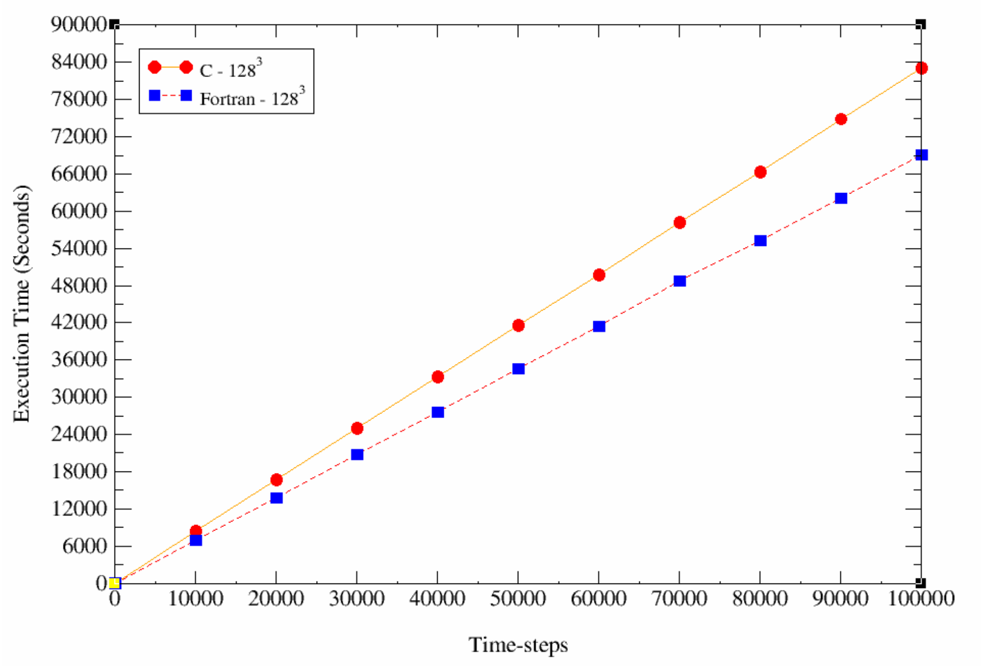
CDS	**CPU**			
Calculation	**128-bit SIMD**	**Speedup**	**256-bit SIMD**	**Speedup**
PBCs	200.25	2.44x	122.36	3.61x
First Laplacian	410.25	2.54x	121.74	3.85x
Map Function	39.74	2.92x	10.87	4.36x
Second Laplacian	362.500	1.92x	94.75	3.68x
Time Evolution	56.00	2.03x	30.25	2.78x

Table 7.2: Number of clock cycles for each CDS calculation based on the 128 and 256 bit SIMD instruction and speedup over a scalar implementation.

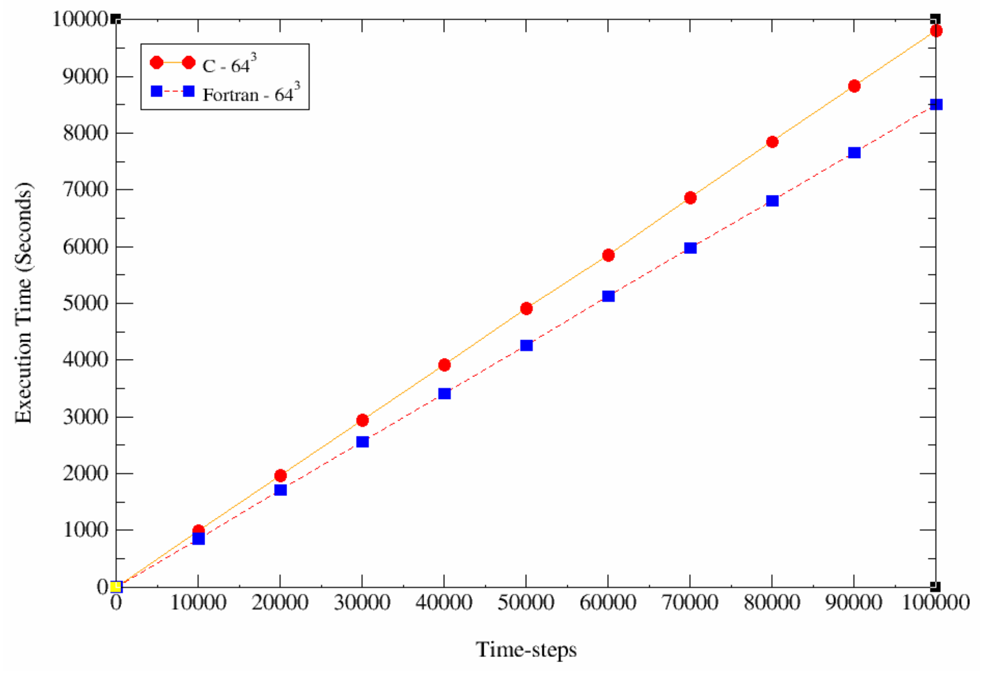
CDS	**128-bit**			**256-bit**		
Calculation	**Light**	**Medium**	**Heavy**	**Light**	**Medium**	**Heavy**
PBCs	102	8	7	110	4	3
First Laplacian	157	2	5	158	3	3
Map Function	21	1	3	24	0	1
Second Laplacian	118	2	2	121	0	1
Time Evolution	39	2	2	41	1	1

Table 7.3: Number of operations for each CDS calculation.

To ensure the C implementation of baseline code is strong enough for optimisation and multi-threading execution, the comparison between C and FORTRAN implementations are taken into account without considering any implicit and explicit optimisations. Figure 7.3 presents the execution times for the baseline (original) CDS code for both C and FORTRAN languages without any optimisation in different time-steps and different domain size. As expected, FORTRAN scalar implementation is faster then C scalar implementation due to the nature of FORTRAN which is static, the size of data will be identified at compile time, and array model. However, investigating the design of FORTRAN is out of the context of this thesis. It should be noted that the comparison between non-optimised baseline codes are performed on the same hardware architecture, compiler and floating point (single precision).



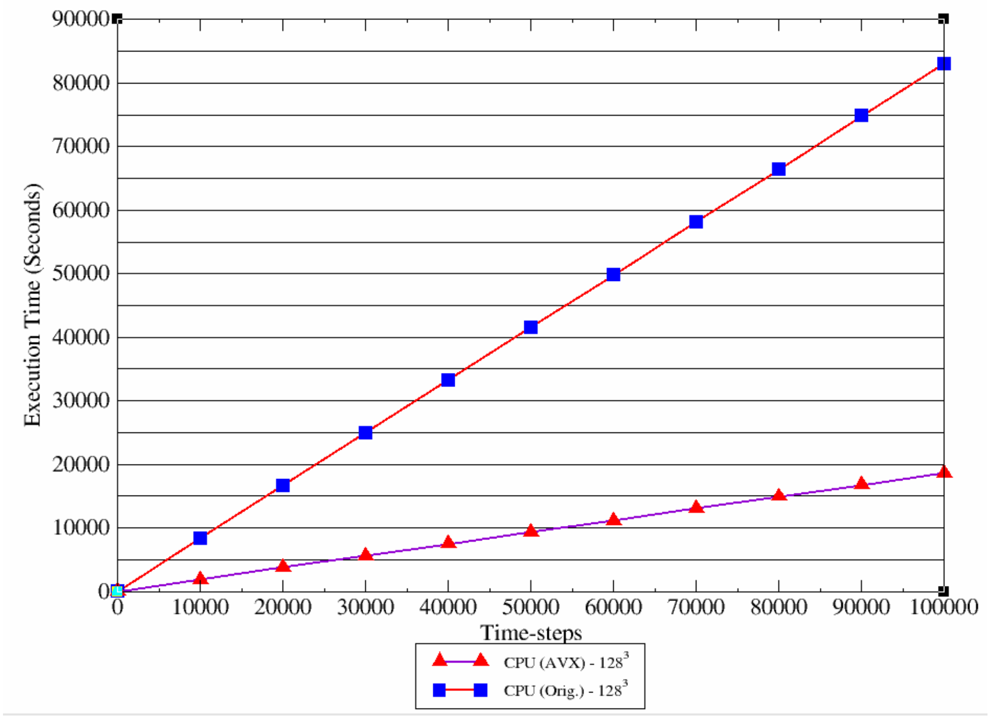
(a) $128 \times 128 \times 128$



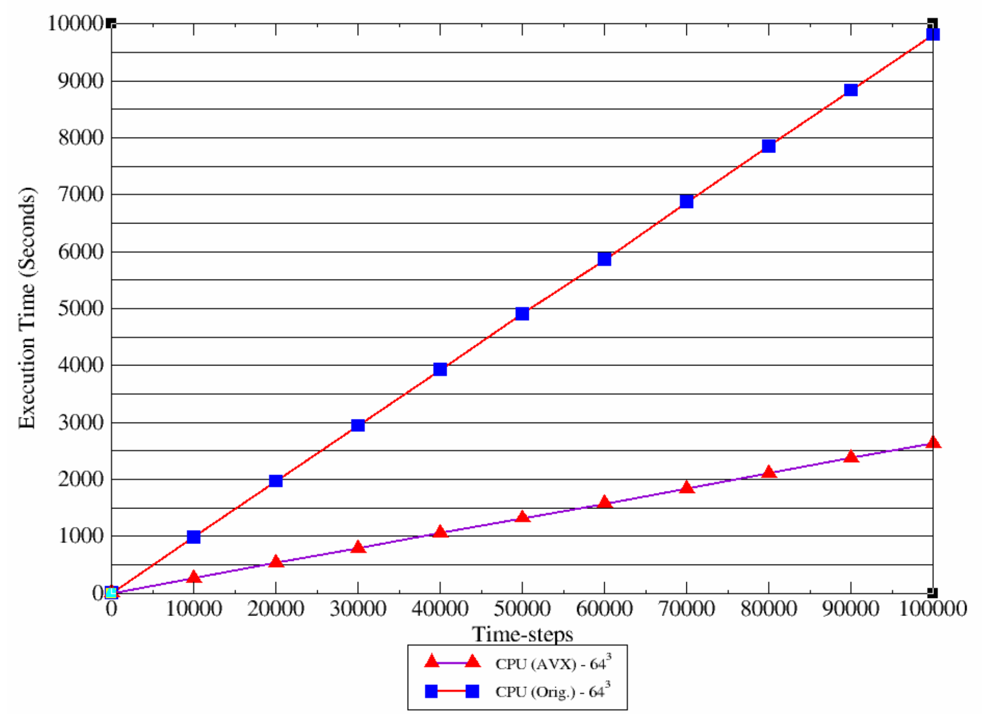
(b) $64 \times 64 \times 64$

Figure 7.3: Execution times for the CDS non-optimised baseline code based on the C and FORTRAN with different system sizes $128 \times 128 \times 128$ (a) $64 \times 64 \times 64$ (b).

Figure 7.4 compares the execution times of CDS optimised implementation with non-optimised CDS. For the different time-steps shown the performance of CDS is almost constant and consequently the computational cost per CDS calculation remains the same over time-steps.



(a) 128×128×128



(b) 64×64×64

Figure 7.4: Execution times for the CDS optimised and non-optimised baseline implementations in different time-steps

It can be seen in Figure 7.4 that SIMD optimisation of the CDS calculations increases the performance substantially. For the AVX instruction set implementation, speedups of 3.74x and

4.44x are achieved for the system sizes of $64 \times 64 \times 64$ and $128 \times 128 \times 128$ respectively. The difference between the speedups of two different system sizes can be attributed to the original non-optimised implementation and it is not related to the scalability of the CDS optimisation. The performance ratio between the two system sizes is the same. Table 7.4 presents the execution times based on the AVX SIMD acceleration for two domain sizes within each time-step.

Time-steps	**Elapsed time (Seconds)** $64 \times 64 \times 64$	**Elapsed time (Seconds)** $128 \times 128 \times 128$
10000	260.87	1866.86
20000	529.72	3771.11
30000	786.27	5594.96
40000	1058.48	7457.51
50000	1313.80	9320.15
60000	1570.65	11164.71
70000	1836.41	13090.16
80000	2099.63	14916.98
90000	2370.48	16750.24
100000	2623.14	18610.78

Table 7.4: Execution times in different time-steps based on the AVX instruction set.

Figure 7.5 illustrates the execution times for the SSE4.2 and AVX instruction implementations of the CDS for two different system sizes. Compared to the SSE4.2 (128-bit), the AVX (256-bit) implementation achieves an additional speedup of 1.3x.

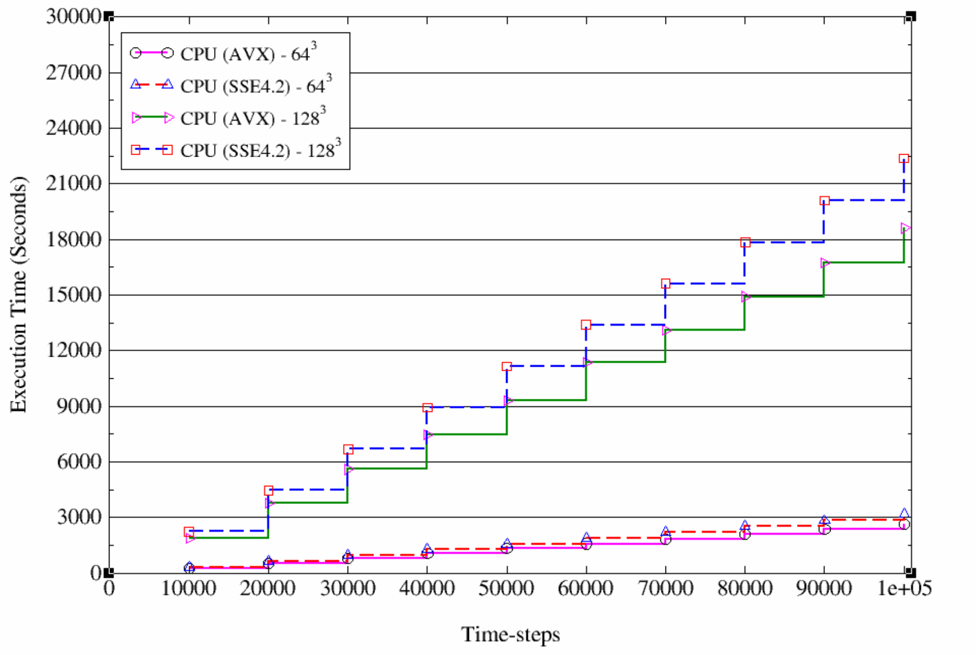


Figure 7.5: Executions times for SSE4.2 and AVX implementations.

By comparing the original (not optimised), SSE4.2 and AVX implementations of the CDS for the system sizes $128\times128\times128$ and $64\times64\times64$, the SSE4.2 implementation achieves 3.71x and 3.10x speedups, respectively; AVX implementation achieves 4.45x and 3.74x for the same system sizes.

7.2 Cell Dynamic Simulation Method on Multi-Core CPU

The second step in the performance enhancement refers to the exploitation of methods for parallelisation on the task level. According to the parallel architectures on the task level there are two well-known APIs for shared memory and distributed memory models: the OpenMP and MPI. The shared memory architecture OpenMP can be considered in terms of two types of shared memory machines: symmetric multiprocessor (SMP) and non-uniform memory access (NUMA). As mentioned in chapter 5, in SMP machine there is no special processor and the operating system treats all the processing units equally. There is also no special memory and all memories are equally accessible by different processors/cores. In NUMA, different memory regions have different access time; the processor has fast access to own memory but slower access to other memories. In the distributed memory architecture every processor has access to own memory and communication (massage passing) needs for accessing and sharing data with each other on the network. These two models of task level parallelism based on OpenMP and MPI have been extensively utilised for parallelising different algorithms in scientific research such as molecular dynamic (MD) [152]. However, this study investigates the implementation of the CDS based on the OpenMP multi-threaded shared memory computers.

OpenMP is the explicit programming model that can be used to accomplish parallelism based on threads/cores. Memory hierarchy model in shared address space machine is classified into shared memory (heap) and private memory (stack). Shared memory or heap can be shared and accessed between all the threads. Private or stack refers to private memory of each thread and cannot be shared with other threads. In shared address computer the shared data structure plays an essential role for parallelisation and optimisation of application by realising synchronisation and communication between threads and controlling data granularity for memory and communication contentions.

Although threads communicate by sharing variables in the heap memory address space, this data sharing may not always be safe, such as when one thread tries to write on a variable that other threads try to read from causing the results to change each time. This data conflict situation is called race conditions, when results fluctuate due to different thread arrangements. By organising and controlling access to shared variables, synchronisation can help to prevent race conditions and other data conflicts. In OpenMP there are two types of synchronisation constructs: barrier and mutual exclusion. Barrier synchronisation impacts all the threads in the team by holding them at a barrier point until the other threads reach that point. Mutual exclusion specifies a section of code that can be executed only by one thread at a time. It should be noted that synchronisation is expensive and frequent synchronisation breaks performance down.

The other challenge in multi-thread parallelisation which undermines the performance (speedup) is the load imbalance (the allocation of unequal amounts of workload to threads), which increases threads' idle time. For instance, when whole numbers of threads reach a synchronisation point the thread with the most workload (i.e. the slowest thread) will control the overall performance.

7.2.1 Parallel Algorithm of CDS for Multi-Threaded Systems

The first step to design a parallel algorithm refers to the decomposition of system into separate parts that can be spread into different parallel tasks and executed simultaneously. There are numerous methods for splitting the system between parallel tasks/threads. The most common of which are data and spatial decompositions. Data decomposition method divides the data related with a problem or computational work and assigns portions of data into different processing units [153, 154]. The spatial decomposition method divides the whole domain into different sub-domains, each of which is organised in a hierarchy data structure, which illustrates the spatial relationship between domains [155, 156]. In fact, data structure plays a very important role in spatial decomposition scheme. Based on the nature of the problem, spatial decomposition method can utilise different types of data structure, comprising of kd-trees [157]; octrees*;* and regular grids [158, 159].

Octrees are axis-aligned tree-based hierarchies dividing the domain. In the octrees data structure each sub-domain has eight children with three axis-aligned splitting dimensions. The k-dimensional tree (k-d tree) is a generalisation scheme of octree, where k indicates the number of dimensions. In the k-d tree each sub-domain has two children without considering the number of dimensions of the system. Both octrees and k-d tree refer to the non-uniform subdivision of the domain and are useful in non-homogenous systems. Regular grids refer to the uniform system which involves number of equal size of cells or parts. The regular grids data structure of spatial decomposition method overlays the whole system with a uniform grid.

A well-known example for spatial decomposition refers to the MD simulation in the distributed memory model, which is applied spatial decomposition method to partition the whole MD system into 3D sub-domains and associate each sub-domain with different processors on the distributed memory architecture [160]. In this model, processors communicate with one another through the number of massage passing calls based on the MPI on the network. Both data and spatial decomposition methods have advantages and disadvantages.

Data decomposition is better in terms of load balancing but not so good in scalability, while spatial decomposition method is better in scalability but weak in load balancing.

It should be noted that based on the model of the shared memory there is no global communication and communication cost between the threads is negligible, therefore communication cost is not considered in both cases. However, to overcome the issues mentioned and to obtain a better performance the appropriate mixture of both methods can be deployed. Therefore, a hybrid decomposition algorithm based on the work-sharing constructs of OpenMP and regular grids data structure according to the CDS simulation method on the shared memory machine was developed. The original idea for the spatial decomposition as a first step of hybrid algorithm for the CDS was based on the processor data structure, which splits the grid into different sub-grids, each of which it assigns different processing units. Although this scheme may be the answer for decomposing the whole system, the results cause a high number of load imbalances and race conditions. Consequently, due to the nature of the CDS method (i.e. cell based homogenous system), the data structure of regular grids is used for spatial decomposition. Hence, the spatial partitioning in the new algorithm considers the whole system as a grid divided into three-dimensional sub-grids. Each 3D sub-grid defines as an array of linked cell model with the same size, which has a relationship with the neighbouring sub-grids. This hierarchy arrangement delivers faster access to memory locations. Figure 7.6 shows the spatial decomposition method based on the cell-linked uniform grid.

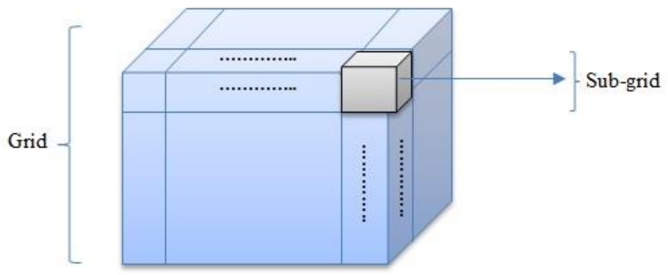


Figure 7.6: Cell linked spatial decomposition scheme.

After partitioning the whole system into different sub-grids/cells, data decomposition as a second step of hybrid decomposition algorithm starts to play a role. In data decomposition, block based decomposition is used to partition a group of cells into different blocks and then map each block to a core/thread in the shared memory machine. The main reason for block based data decomposition is to prevent load imbalance. It should be noted that the size of the block or number of cells per block can be determined according to the size of the system N_x, N_y, N_z and the blocks are scheduled statically (at compiler-time) to assign to threads. In the hybrid method the uniform cell linked data structure was considered to be a shared data structure that can be divided into different blocks of data and split among different threads (rather than mapping each part of the CDS method to a single thread in the multi-core machine). Thus, the main differentiation of this method is the scheme of distributing sub-systems among the cores/threads, and structure of the shared data. The following figure presents the flow chart of the OpenMP CDS algorithm based on the hybrid method.

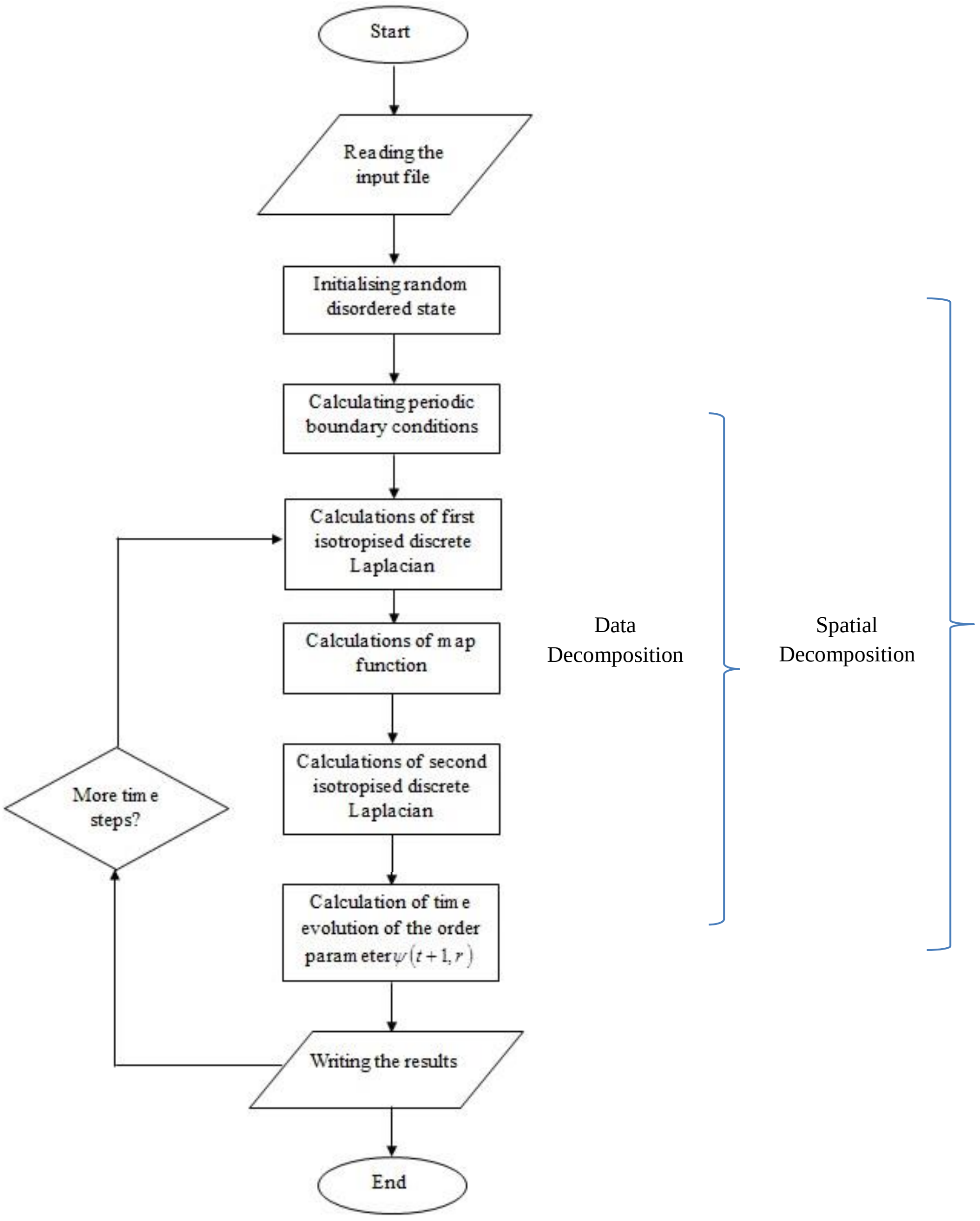

Start
Reading the input file
Initialising random disordered state
Calculating periodic boundary conditions
Calculations of first isotropised discrete Laplacian
Calculations of map function
Calculations of second isotropised discrete Laplacian
Calculation of time evolution of the order parameter $\psi(t+1,r)$
Writing the results
End
More time steps?
Data Decomposition
Spatial Decomposition

Figure 7.7: Schematic of the OpenMP cell dynamic algorithm.

As discussed earlier, one of the main challenges in multi-threaded programming and shared memory model is race conditions, specifically when dealing with shared arrays. In OpenMP implementation, the initial random disordered state is implemented as a critical section to prevent race conditions. Due to the nature of the CDS method the whole simulation starts from an initial random disordered state (ψ) and is discretised on a lattice. Therefore, it is necessary to prevent race conditions and data conflict. It should be noted that other than the critical section the implied barriers are considered at the end of each sub-system. The other sub-systems are decomposed more based on the data decomposition method by considering loop work-sharing constructs of OpenMP.

7.2.2 Simulation Results and Performance Tuning

This section presents the simulation results and performance tuning for the CDS multi-thread development. The experimental results were executed on the same machine specifications which mentioned in section 7.1.1, and the following table illustrates the simulation parameters which have been used for the simulation results of the CDS method.

Simulation parameter	**Parameter value**
Number of Cores	6
Hyper-Threading	2
System Size	64^3, 128^3
Instruction Set	AVX
Total Execution-time	100,000

Table 7.5: Simulation parameters in multi-threads implementation.

To present the performance of the proposed hybrid algorithm, the simulation is executed with a different number of cores and the execution or computing time of each core is demonstrated. The execution time is based on the wall-clock time which involves the CPU and system time. The CPU time indicates spending time for the computations and the system time refers to the time that was spent for file input/output, transferring or waiting. In addition, two other performance analyses are considered: parallel speedup and efficiency. Parallel speedup refers to the execution time taken to process the computation on *P* processors against on single processor. The parallel speedup can be defined by the following equation [161]:

$$S(N,P)=\frac{T(N,P=1)}{T(N,P)}. \tag{7.2}$$

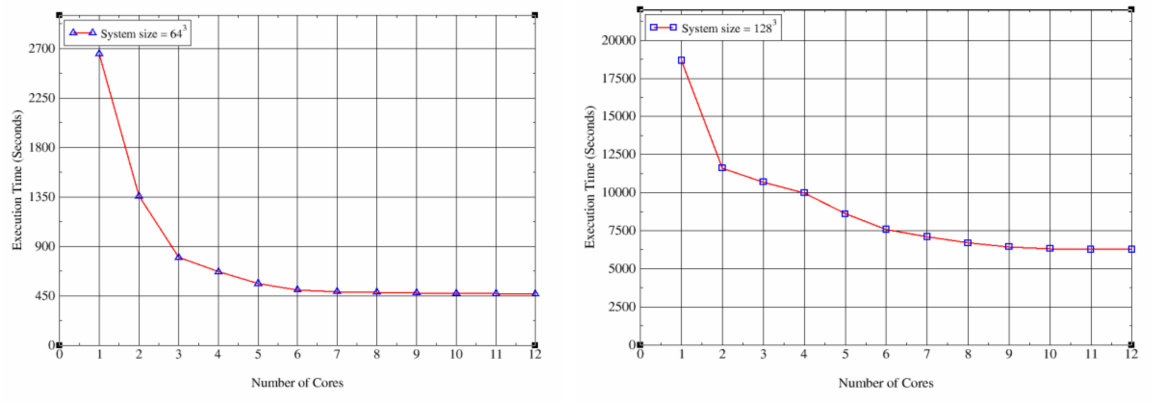
Where *N* refers to the total volume of computational work and *P* indicates to the number of processors/ threads. *T (N, P=1)* and *T (N, P)* are execution time for 1 and *P* processors respectively.

To achieve a more accurate comparison between the parallel and serial execution the efficiency of parallel execution should be taken into account. The parallel efficiency considers the efficiency of fixed volume of computational work executing on *P* cores. This relates to how efficiently processors are used in parallel execution. The efficiency of parallel execution can be determined by [161]:

$$E(N,P)=\frac{S(N,P)}{p}. \tag{7.3}$$

Here *p* is the number of processors, *N* indicates a computational work, and *S (N, P)* refers to the parallel speedup with respect to the number of processors.

To understand the impact of the block size of data decomposition in the hybrid algorithm, three different scenarios were considered: (*i*) data partitioning based on the block size = 1000; (*ii*) data partitioning based on the block size = 100; and (*iii*) data partitioning based on the function of the number of threads. Note that the other numbers of block sizes were considered, but the results turned out to be unacceptable and poor in terms of speedup, execution time and load balancing, therefore they are not stated here. The following figure presents the execution times as a function of the number of threads for two different system sizes based on the first scenario of data decomposition in the hybrid algorithm.



(a) $64\times64\times64$ (b) $128\times128\times128$

Figure 7.8: Multi-threaded execution times for two different system sizes based on the first scenario.

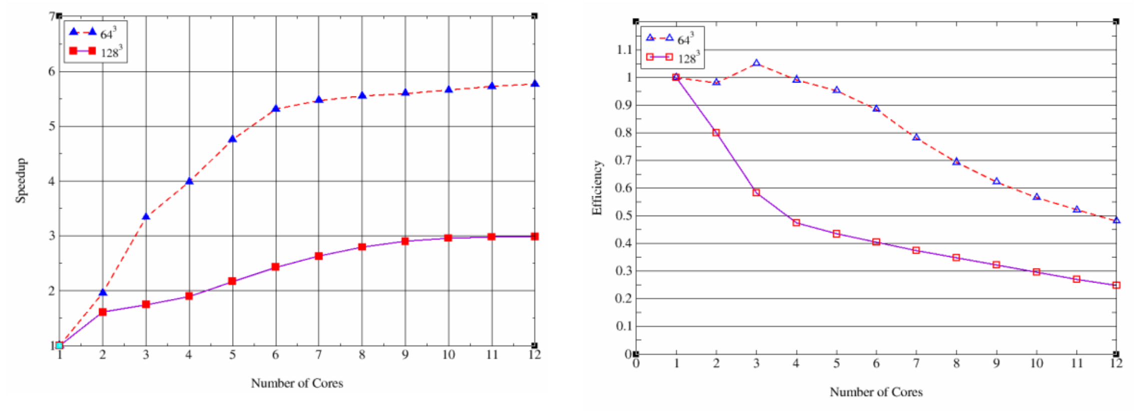


Figure 7.9: Speedup (left) and efficiency (right) for $64\times64\times64$ and $128\times128\times128$ system sizes based on the first scenario.

Figure 7.9 illustrates the speedup and efficiency of multi-threading for two system sizes based on the first scenario, which considers the block size =1000. It can be seen that the speedup in $64\times64\times64$ system varies from 1.96 to 5.77, with the number of threads increasing from 1 to 12, while the efficiency degrades from 1 to 0.48 (100.0% to 48.1%) except in the third core, which exhibits super-linear speedup. One of the possible reasons for super-linear speedup in this phenomenon refers to the cache effect resulting from various memory hierarchies of a computer. In parallel computing, in addition to the numbers of processors/cores changing, the sizes of accrued caches from different processors/cores also change. In fact, by having larger accrued cache size, more computational work can be stored in the cache, thereby decreasing the memory access time. This leads to extra speedup for the specific processor/core with larger accumulated cache [162, 163]. In other words, super-linear speedup occurs in the third core because of more efficient resources (such as RAM, cache and registers) in low-level computations available. In the larger $128\times128\times128$ system, the speedup differs from 1.60 to 2.98 with respect to the number of threads, while the efficiency drops sharply from 1 to 0.24 (100.0% to 24.9%). By considering the speedup between the two system sizes and with the help of the Intel Vtune visual performance analysis [164], the inherent parallelism and the overhead of parallel libraries are high in scenario one. Inherent parallelism refers to the load imbalance and parallel libraries overhead indicates the scheduling overhead. Scheduling overhead refers to the thread scheduling overhead when the workload is not adequate between threads, which consequently increases idle or waiting time. It should be noted that in addition to thread scheduling overhead, synchronisation also increases the idle time.

Vtune performance analysis is used to monitor and gather all the statistical data and analysis of the system involving the benchmark program and Linux kernel. The method for collecting statistical data and analysis is based on the sampling technique. Sampling technique or statistical sampling refers to the execution of the program in an environment where it is broken up into some group of frequency (e.g. 100 times per second) and the position of the program counter will be saved before the program is started again. When execution of program is finished these positions/locations are decoded into the source code and then statistical data will be analysed to find out the hotspots of the program [164].

Figure 7.10 shows a percentage of the wall-clock time when the specific number of threads were executing simultaneously in the OpenMp region of isotropised discrete Laplacian in the CDS, based on the Vtune tool for $128\times128\times128$ system size.

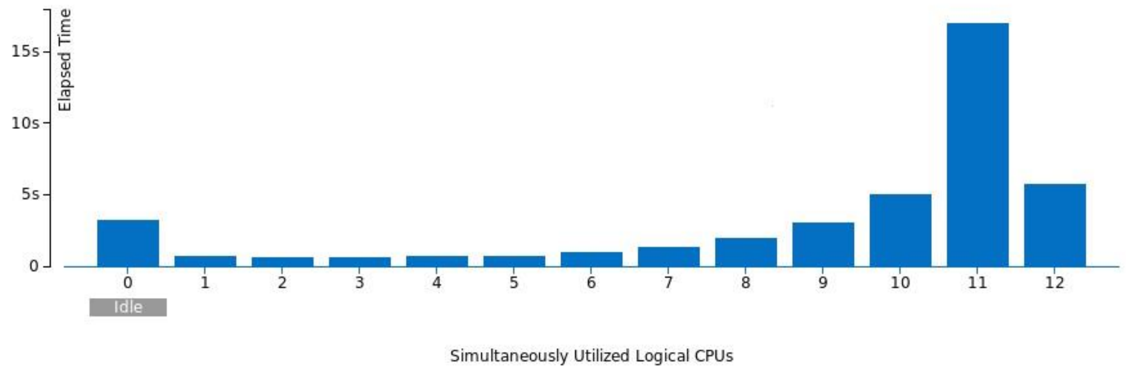
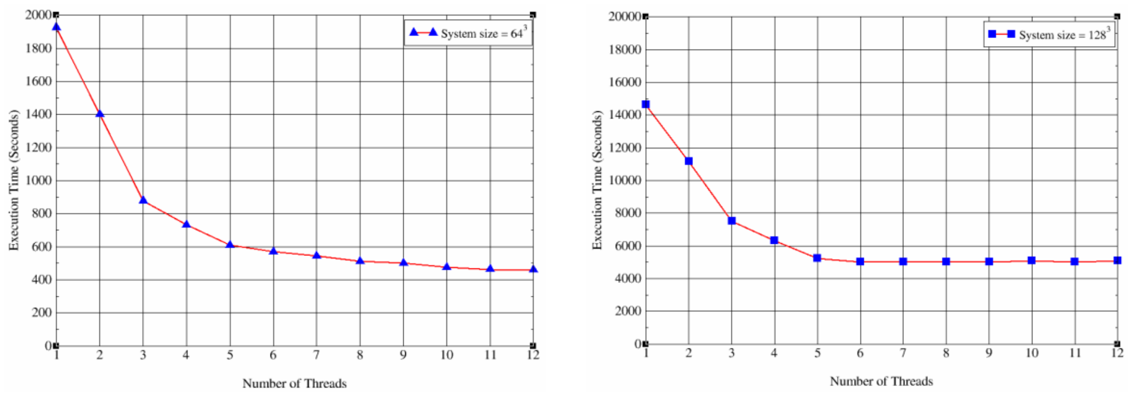


Figure 7.10: Histogram of OpenMP threads usage for the calculation of first discrete Laplacian based on the first scenario.

It can be seen that there is a high amount of load imbalance between the threads; therefore there is a need to consider other scenarios for partitioning data blocks. Figure 7.11 presents the execution time as a function of the number of threads based on the second scenario of data decomposition in the hybrid algorithm.



(a) $64 \times 64 \times 64$ (b) $128 \times 128 \times 128$

Figure 7.11: Multi-threaded execution times for two different system sizes based on the second scenario.

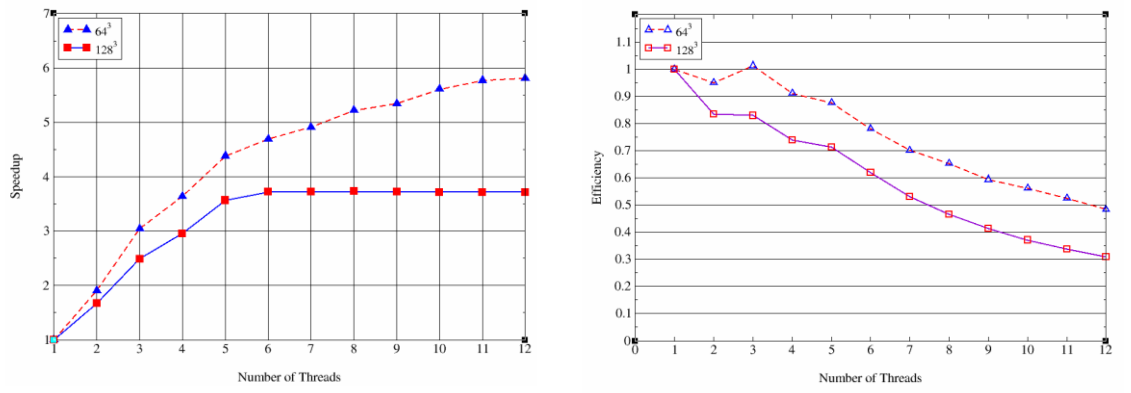


Figure 7.12: Speedup (left) and efficiency (right) for $64 \times 64 \times 64$ and $128 \times 128 \times 128$ system sizes based on the second scenario.

Figures 11 and 12 present the execution, speedup and efficiency of multi-threading for $64\times64\times64$ and $128\times128\times128$ domain sizes based on the block size =100. It can be seen from the figures that the speedup in $64\times64\times64$ system varies from 1.90 to 5.81, with the number of threads increasing from 1 to 12, while the efficiency is reducing from 1 to 0.48 (100.0% to 48.4%) except in the third core, which is same as in the first scenario (super-linear speedup). In $128\times128\times128$ domain size, the speedup differs from 1.67 to 3.71 with respect to the number of threads, while the efficiency is decreasing from 1 to 0.3 (100.0% to 30.9%). By comparing the speedup and efficiency between the two scenarios, it can be noted that in $64\times64\times64$ domain size there is a very small difference in the speedup and efficiency, while in the larger $128\times128\times128$ system size there is a big difference in terms of the speedup and efficiency between two scenarios. Consequently, the second scenario is more sufficient and adequate for a large system. The following figure displays a percentage of the wall-clock time when the specific number of threads were executing simultaneously in the OpenMp region of first isotropised discrete Laplacian based on the second scenario for $128\times128\times128$ domain size.

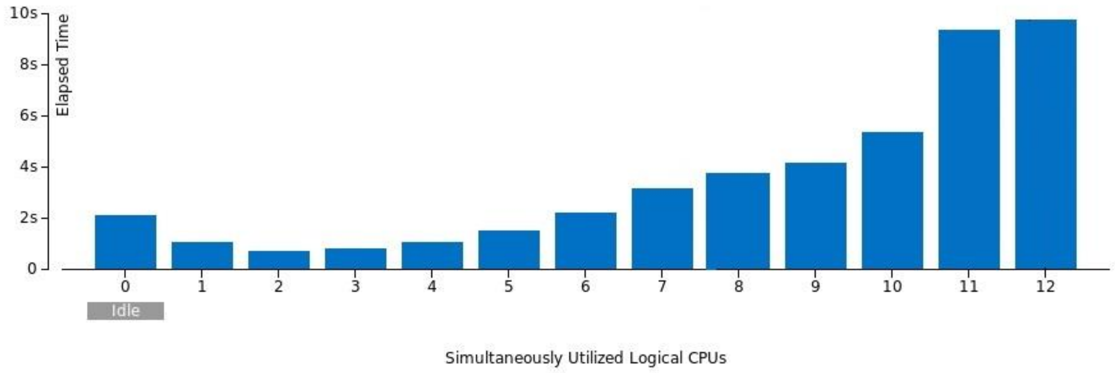
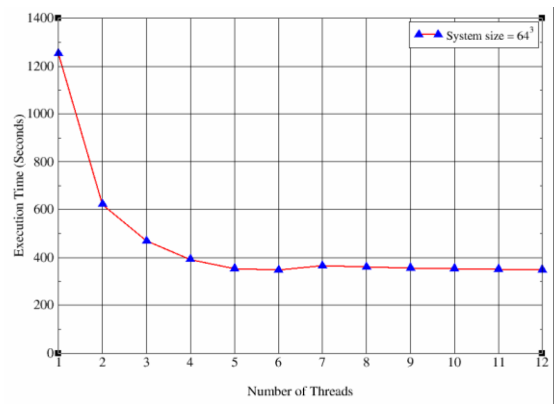
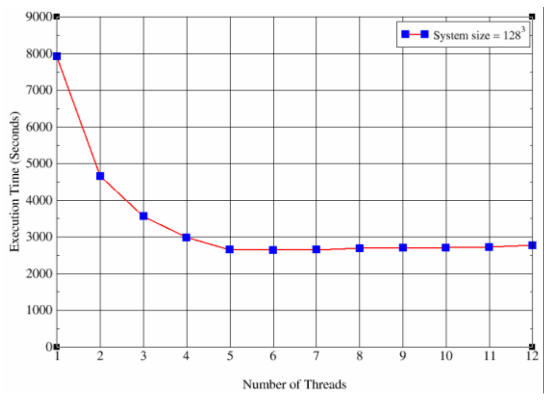


Figure 7.13: Histogram of OpenMP threads usage for the calculation of first discrete Laplacian based on the second scenario.

From Figure 7.13 it can be seen that the load imbalance is slightly improved compared to the histogram of OpenMp for the first scenario, but the volume of inherent parallelism remains insufficient. In the last step, the third scenario based on the function of the number of threads is considered. Figure 7.14 illustrates the execution time for two system sizes based on the third scenario.



(a) $64 \times 64 \times 64$



(b) $128 \times 128 \times 128$

Figure 7.14: Multi-threaded execution times for two different system sizes based on the third scenario.

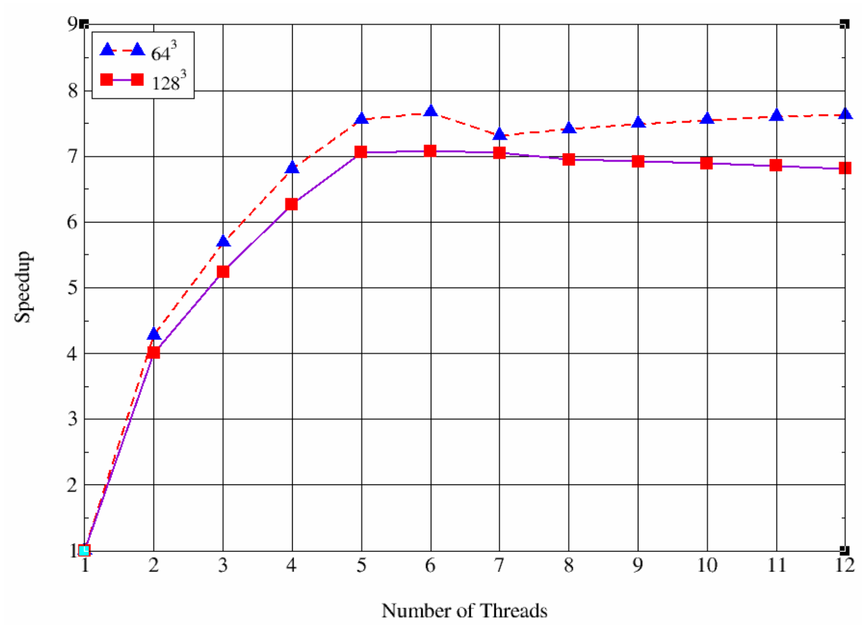


Figure 7.15: Speedup $S = S(n, P)$ for the third scenario.

Figure 7.15 presents the algorithm speedup with respect to the function of the number of threads. It can be seen the speedups from 4.27 to 7.26 and 4.00 to 6.81 with the number of cores cumulative from 1 to 12 for $64 \times 64 \times 64$ and $128 \times 128 \times 128$ system sizes respectively. As expected, the speedup and efficiency in the third scenario are better than the other cases, specifically when simulating a large CDS system with data decomposition based on the fraction of the system size to the function of the number of threads. The following figure presents a percentage of the wall-clock time the specific numbers of threads were executing concurrently for the first isotropised discrete Laplacian calculation based on the third scenario for $128 \times 128 \times 128$ system. Figure 7.15 clearly shows the improvement of load imbalance between threads.

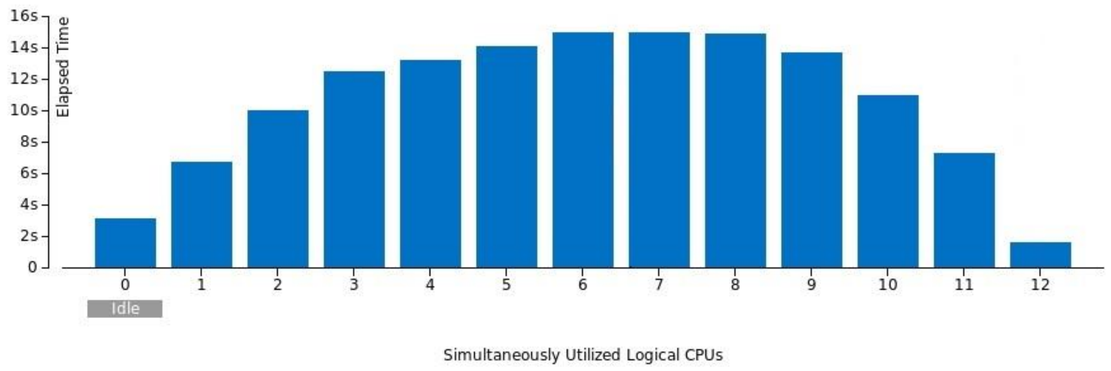


Figure 7.16: Histogram of OpenMP threads usage for the calculation of first discrete Laplacian.

Although in the third scenario the overall speedup and the load imbalance are improved, it can be seen from Figure 7.15 that the speedup in both system sizes, specifically in the large system, starts to decrease after the sixth thread. The main reason for this phenomenon is Intel's Hyper-Threading (HT) and the effects of HT technology on a system performance. In general, HT technology indicates the enhancement of parallelisation of computations by considering a single physical processor as two logical processors. The resources of physical processor such as cache and control units are divided, and the architectural state, which includes control registers, memory management unit, counter registers and addresses registers, is duplicated for two logical processing units [165, 166]. By duplicating architectural state and sharing resources, HT technology allows a single physical processor to processes instruction streams in parallel from different threads improving the performance [166].

In principle, there are different levels of parallelism (section 5.3) that can be utilised in the modern processor to increase the performance. These parallelism levels can utilise the underlying parallelism level (hybrid model) to obtain a better performance. Regarding the hybrid levels of parallelism the HT technology provides exploitation of the hybrid model based on the instruction and task levels parallelism. Although HT by exploiting hybrid model allows a processor to dynamically allocate resources to threads and permits multiple threads to be executed simultaneously on an SMT processor, threads must share the main physical processor resources between each other. Therefore, this concurrent sharing of resources causes a potential bottleneck and degrades the performance.

Different studies have investigated the impact of HT technology on performance and whether HT is beneficial or not. These studies and Intel company have exposed that the HT technology can be enhanced the performance of application by 10-30%, depending on the characteristics of the program [167, 168]. They also suggest that the workload plays an essential role in the performance of multi-thread program. Therefore, the main reasons for the degradation in the performance when HT is enabling in high amount of workload are: (*i*) greater number of threads is directly related to increased synchronisation cost (increasing number of threads means more spending time between threads for synchronisation); (*ii*) considering two logical threads in single physical processors shares the processor resources, which has a direct impact on the performance, especially when the volume of workloads is high; (*iii*) supplementary memory contention; and (*iv*) competition between logical threads for access to the caches cause additional cache-miss situations. However, to answer whether HT is beneficial or not depends on the nature of the program and the algorithm of implementation. In the third scenario of the CDS multi-threading program, it can be seen clearly that HT technology is not very useful in $64\times64\times64$ system size and is considerably inferior when increasing the domain size to $128\times128\times128$.

To comprehend the impact of HT on shared memory computer the third scenario is considered with and without HT in two system sizes. Figure 7.17 displays the speedup with respect to the number of physical cores with HT enabled and disabled for $64\times64\times64$ and $128\times128\times128$ domain sizes.

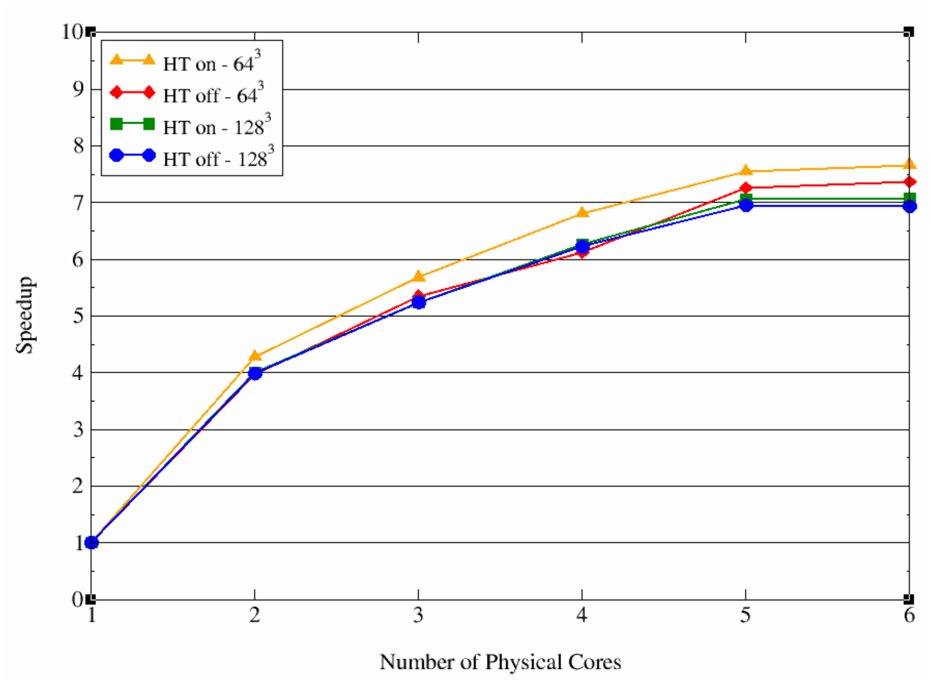


Figure 7.17: Parallel scaling results with Hyper-Threading enabled and disabled.

The results displayed in Figure 7.17 show that when HT is enabled in $64\times64\times64$ system size the performance improves by around 5%. On the other hand, in $128\times128\times128$ domain size there is no considerable difference when HT is enabled. In fact, the degree of difference when HT is enabled compared to when it is disabled in the bigger system size is less than 2%. Therefore, as mentioned earlier, HT is not very beneficial in $64\times64\times64$ domain size and is considerably inferior when increasing the system size to $128\times128\times128$. The following figures present the execution times and the speedup for the original CDS and multi-threads implementation based on the first scenario for 128×128×128 system size, with a total of 100,000 time-steps.

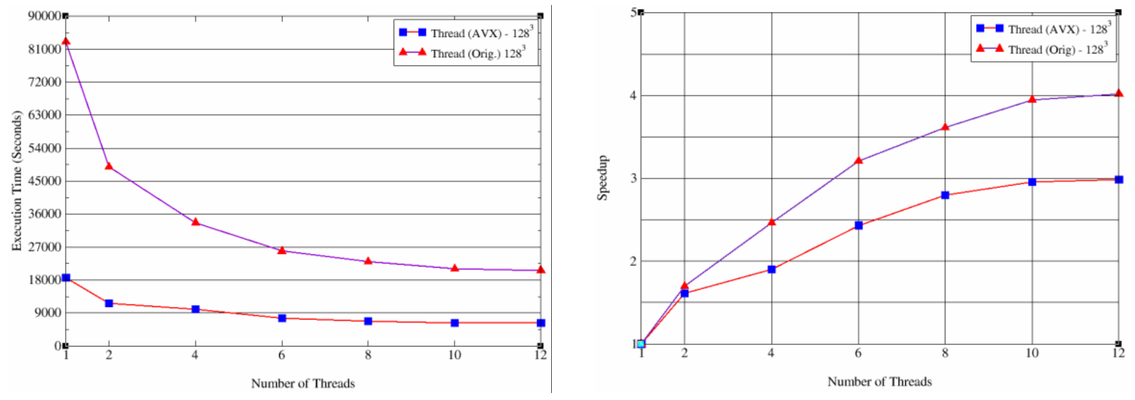


Figure 7.18: Execution times (left) and speedup (right) for the original and AVX multi-threaded.

From the above figures it can be seen that for the original implementation the speedup improves by a factor of 4x from 1 to 12 threads, while for AVX multi-threads implementation grows by 3x from 1 to 12 processing units. As a result the original code scaling to some extent is better; the main reason for this is the much worse total execution time of original implementation. Figure 7.19 shows the speedup for the original and AVX implementations as a function of the number of physical cores based on the third scenario. To have a fair and reasonable comparison between original and AVX codes, HT for both situations is considered as disabled. The original implementation gains around 2.4% speedup on six physical cores without HT.

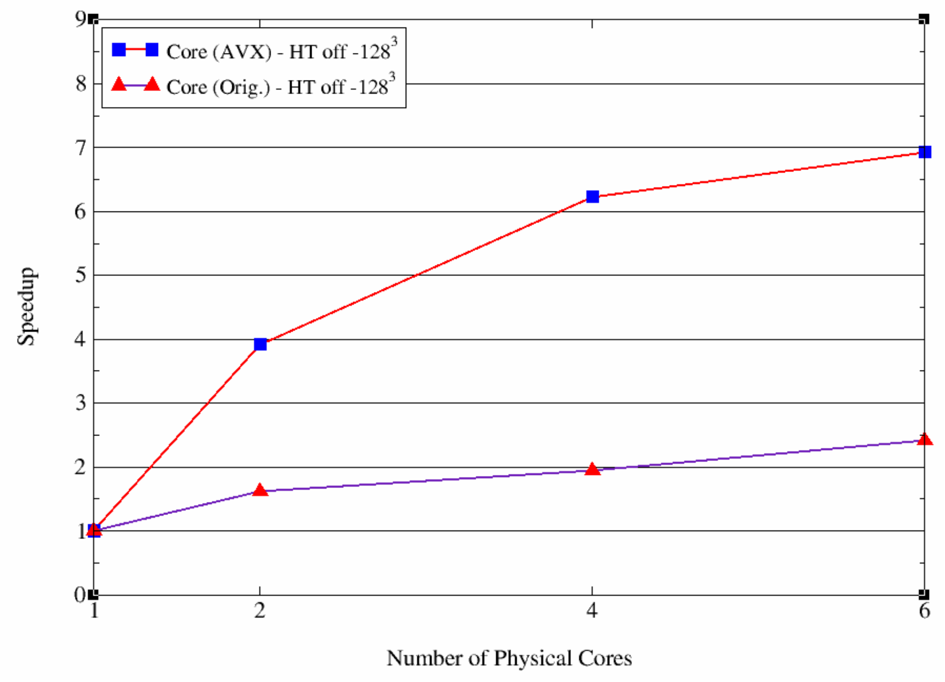
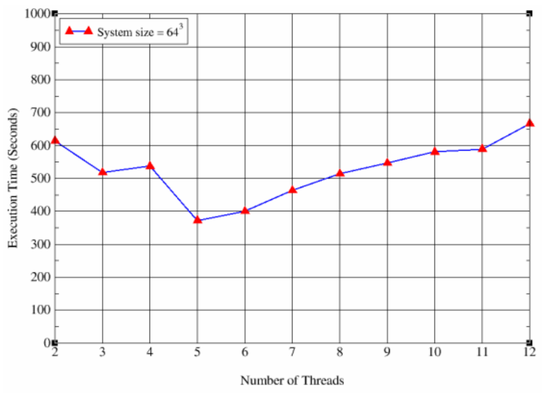
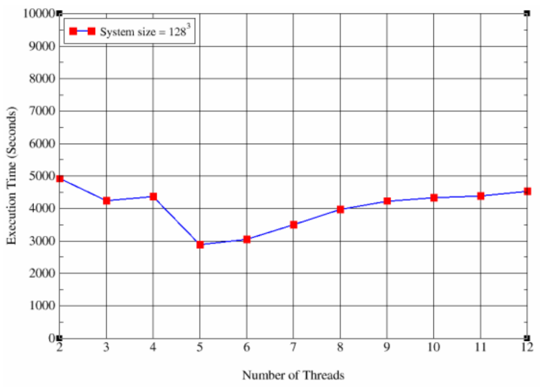


Figure 7.19: Speedup for the original and AVX implementations based on the third scenario.

In addition, to evaluate the hybrid algorithm and to show the accuracy, the CDS method was also developed based on the functional partitioning in spatial decomposition scheme. In general, functional partitioning focuses on the computational works which need to be processed rather than data operated by the computational works. In functional decomposition the computational work or problem is partitioned into different parts/tasks, each of which will be assigned to one processing unit. A good example of functional partitioning is climate modelling. In climate modelling the whole problem is decomposed into four components (atmosphere, ocean, land, and hydrology), each of which can be considered as a separate task to assign to processing unit [169]. In the spatial functional decomposition scheme the spatial decomposition is same as previously discussed (Figure 7.7), but in the functional partitioning, rather than considering a block of data for partitioning, each calculation of the CDS is considered as a separate task mapped to the processing unit. Note that each direction (x, y, z) of periodic boundary condition is also divided into a separate task. The following figure displays the execution times as a function of the number of threads for $64\times64\times64$ and $128\times128\times128$ system sizes based on the spatial functional decomposition method.



(a) $64\times64\times64$



(b) $128\times128\times128$

Figure 7.20: Multi-threaded execution times for two different system sizes based on the functional decomposition.

The scalability in functional partitioning scheme is poor as illustrated above. With the help of the Intel Vtune performance analysis it is found that in functional partitioning the parallel overhead between threads is high and false sharing occurs. False sharing refers to the situation wherein different array elements try to share the same cache-line. It should be noted that if the code is weak scaled perfectly, the number of threads will not change the execution time. Figure 7.21 illustrates the parallel speedup of hybrid decomposition from three scenarios and spatial decomposition with functional partitioning algorithms for $128\times128\times128$ system size in 100,000 time-steps.

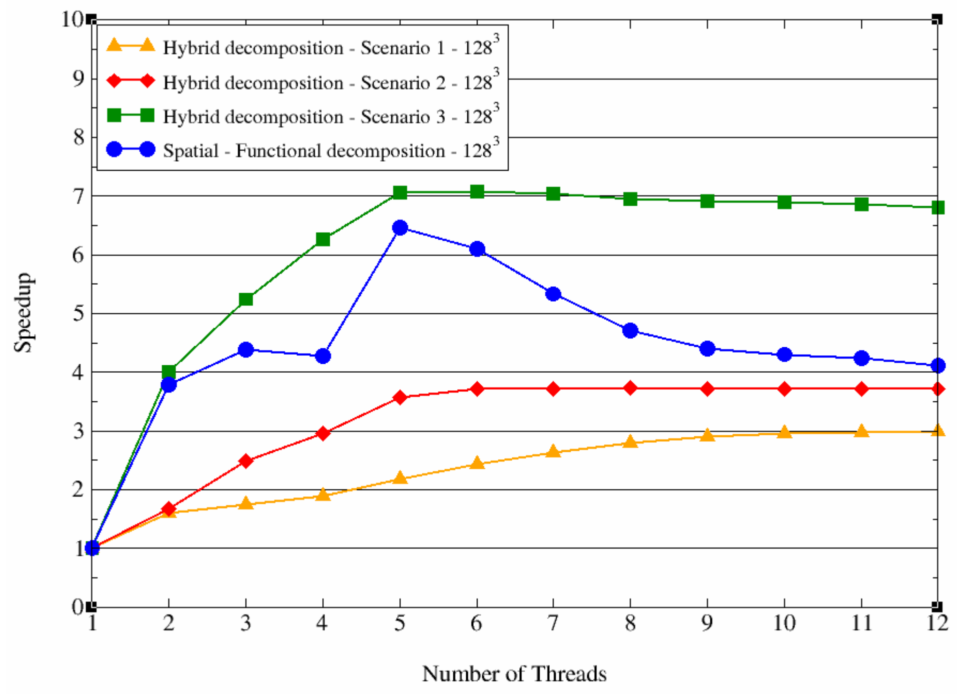


Figure 7.21: Speedup for four decomposition strategies.

Figure 7.21 indicates that the new hybrid decomposition algorithm based on the third scenario gains the best parallel speedup and performance for the CDS simulation system. Although the spatial decomposition based on the functional partitioning to some extent achieves better speedup compared to the first and second scenarios of hybrid algorithm the amount of the inherent parallelism overhead and false sharing is high. It should be noted that all results are executed with enabled HT.

7.3 Cell Dynamic Simulation on GPU

Over the last few years the highly parallel architecture of GPUs and the ability of GPGPU have increased the usage of GPUs in the area of HPC and PCs [122]. This ability and parallel architecture present considerable promise and different opportunities for scientific engineering and computing (e.g. real-time applications and heavy simulation) to solve expensive and time-consuming computational works more quickly and economically (e.g. less power consumption), without considering CPU clusters [170]. In addition, load balancing, scalability and portability issue of SIMD architectures are other rationales for the high utilisation of and demand for GPUs. Different applications such as CFD [171], particle physics [172], molecular dynamic and lattice Boltzmann method (LBM) [173, 174] are commonly used in HPC studies and widely investigated and implemented on GPUs; in fact, due to parallelisable nature of these applications, they are particularly suitable candidates to be implemented on the latter. CFD and particle physics are considered by a specific data dependency pattern called wavefront dependency, whose pattern according to Lampart [175] can be solved based on the hyperplane algorithm, whereby the calculation and computation of the value of each cell in a grid (i, j, k) depending on the values of three of its cell neighbours is computed by former calculations (i, j, k-1), (i, j-1, k) and (i -1, j, k). Molecular dynamic is characterised by calculations of short-range and long-range forces between pairs of atoms. The cell-based method is mainly used to calculate short and long ranges forces between pair of atoms. This method specifies the group of neighbouring atoms by splitting the whole computational work into cells and calculating the forces between an atom and the contents of some group of surrounding cells [172]. LBM can be used as a model of fluid flow by calculating linear advection and local relaxation operation between neighbouring cells [176]. These applications by nature are parallelisable, and implementing them on GPUs can achieve significant speedup of applications.

Optimisation of CUDA programming model sometimes can however be very difficult due to the different specifications of GPUs, including: (*i*) memory coalescence; (*ii*) number of threads; (*iii*) number of register per thread; *(iv)* the amount of shared memory; and (*v*) number of thread blocks. The efficient values of these parameters on one GPU's architecture and compute compatibility may not be efficient and optimal on other architectures with different compute compatibility, raising the issue of portability. Hence, this section presents the implementation, evaluation and the performance analysis of the CDS method on GPU with respect to the multiple architectures/platforms.

7.3.1 CUDA Parallel Algorithm of the CDS from Domain Level View

As discussed in section 7.2.1, the common approach to design a parallel algorithm is decomposition of the system into separate regions spread into different parallel tasks and executed concurrently. The decomposition method used in the CUDA implementation of the CDS on GPU is based on the fine-grained spatial decomposition. In the whole process, as demonstrated in multi-threaded parallel algorithm, the entire system is conceptualised as a 3D grid or cube, each box or sub-grid of which demonstrates a data cell of the system with the same size. In the method described, the whole system as a cube is decomposed into 3D grids, and according to the CUDA threads hierarchy architecture each cell or sub-grid of the system is assigned to one thread with three respective dimensions (x, y, z) responsible for all calculation, and groups of threads are clustered into blocks. According to the CUDA hierarchy architecture [17, 124], each kernel function is performed in a grid of threads. Each grid is divided into blocks (thread blocks) and each block is divided into a number of threads. From a parallelisation perspective for NIVIDIA GPUs, computational task is parallelised at two levels; the computation problem is divided into blocks involving groups of threads, with each thread block assigned to a SMX, and threads within a block spilt further into sets of 32 threads called warps. All the threads in a warp are executed in parallel and perform the same instruction.

In the spatial decomposition method due to the CUDA hierarchy model, the uniform block-cell linked data structure was used to be a shared data structure. This model of data structure by partitioning the entire domain/system into cells and grouping them into different numbers of blocks according to the CUDA architecture helps to implement the whole system on GPU efficiently and competently. Therefore, the compelling reasons for choosing spatial decomposition scheme based on the block-cell linked structure for CUDA parallel implementation of the CDS are: (*i*) high compatibility with GPU architecture; (*ii*) less maintain overhead or few numbers of memory accesses; and (*iii*) effective usability of thread and instruction levels parallelism. Figure 7.22 presents 2D CUDA thread hierarchy architecture for describing GPU computation of spatial decomposition method based on the block-cell link structure in time. It should be noted that due to the architecture of GPU, which consists of many threads each of which is responsible for each cell that must be computed, fine-grained distribution was selected for spatial decomposition method, whereas in multi-thread parallel algorithm on CPU coarse-grained spatial decomposition is used.

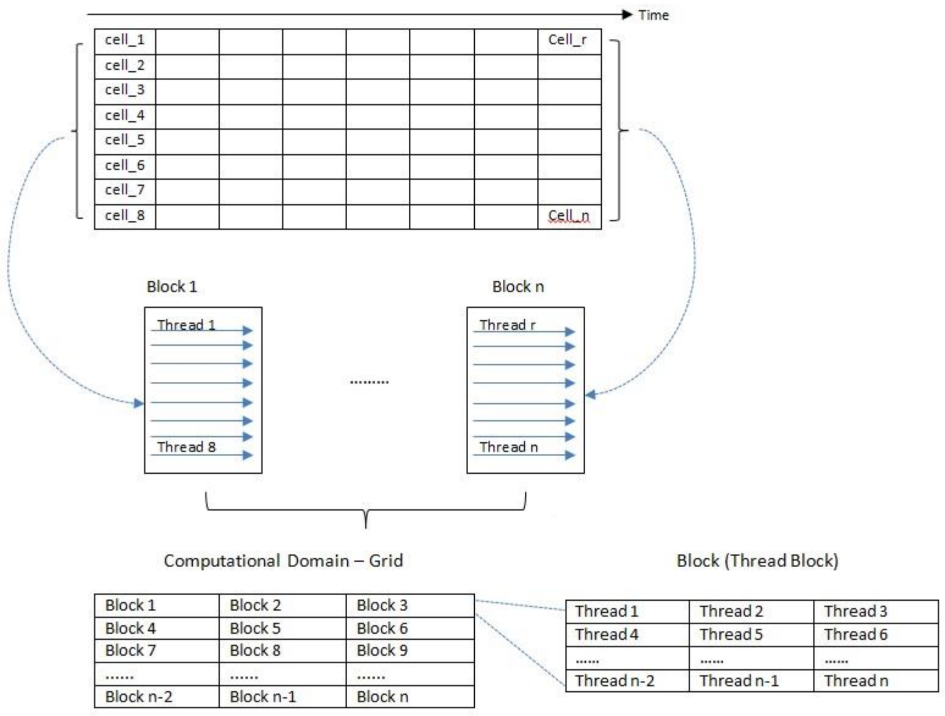


Figure 7.22: CUDA threads model of spatial decomposition scheme based on the block-cell link model.

According to the results (sections 7.3.6 and 7.3.7) of the implementation of the spatial decomposition method based on the block-cell link model, the following sections present this method as a suitable and appropriate choice for GPU-accelerated CDS simulation, which is to be expected to prevent workload imbalance by allocating enough resources for each data element (cell), enhancing system performance and reducing communication overheads/costs, ultimately decreasing the GPU's memory access time.

7.3.2 Memory Arrangement and Layout

According to the CUDA parallel algorithm of the CDS, the simulation code is parallelised in such a way that one thread is responsible to execute the CDS calculations at one spatial location $\psi(r)$ in the 3D grid/domain. Each thread utilises a unique and distinctive coordinate (thread index) to find the appropriate portion of the data to perform and process. Each thread accesses the portion of the data within DRAM once initially loaded. However, to comprehend the memory layout and the allocation of multi-dimensional arrays in DRAM, how C programming language accesses data elements of dynamically assigned multi-dimensional arrays must first be understood. It should be noted that CUDA C programming model was used in GPU implementation, therefore C programming language was taken into account in this investigation.

Ideally, a data element at the row j and column i in two-dimensional array can be accessed, such as d_pxi[j][i] array. However, on CUDA C programming language this type of access is not possible due to the number of columns in d_pxi array not being known at compiler time. One of the main reasons to consider dynamically allocated arrays is to permit changing the dimensions and sizes of arrays based on data size at run time [139]. Consequently, due to the information regarding the number of columns is unknown at compiler time; a dynamically allocated multi-dimensional array is not possible on CUDA C programming model. Subsequently, it is necessary to explicitly "flatten" or linearise a dynamically allocated multi-dimensional array into a one-dimensional array within the DRAM. In principle, due to the utilisation of a flat memory space in modern computer architectures, C programming language also linearises multi-dimensional array into a single dimension. To a certain extent, in C programming language the compiler has responsibility to convert or to flat multi-dimensional array into a single dimension array. Note that the memory space indicates a simplified view of accessing a memory by processor in modern computer architecture based on the address of each location that accommodates a byte of data.

There are two main techniques to flatten a multi-dimensional array into a single dimensional array: row-major layout and column-major layout. In row-major layout all the same row data elements are placed into consecutive and successive locations, then the rows are located serially one after another into the memory space. Column-major layout considers all the same column data elements to be located into consecutive locations and columns are placed serially into memory. C and CUDA C programming languages use row-major layout and column-major layout is mainly utilised by FORTRAN language, which is beyond the scope of this study. An example of 2D array/matrix H[j][i] where j=4 and i=4, which is flatted into a 16 element one dimensional array is presented in Figure 7.23.

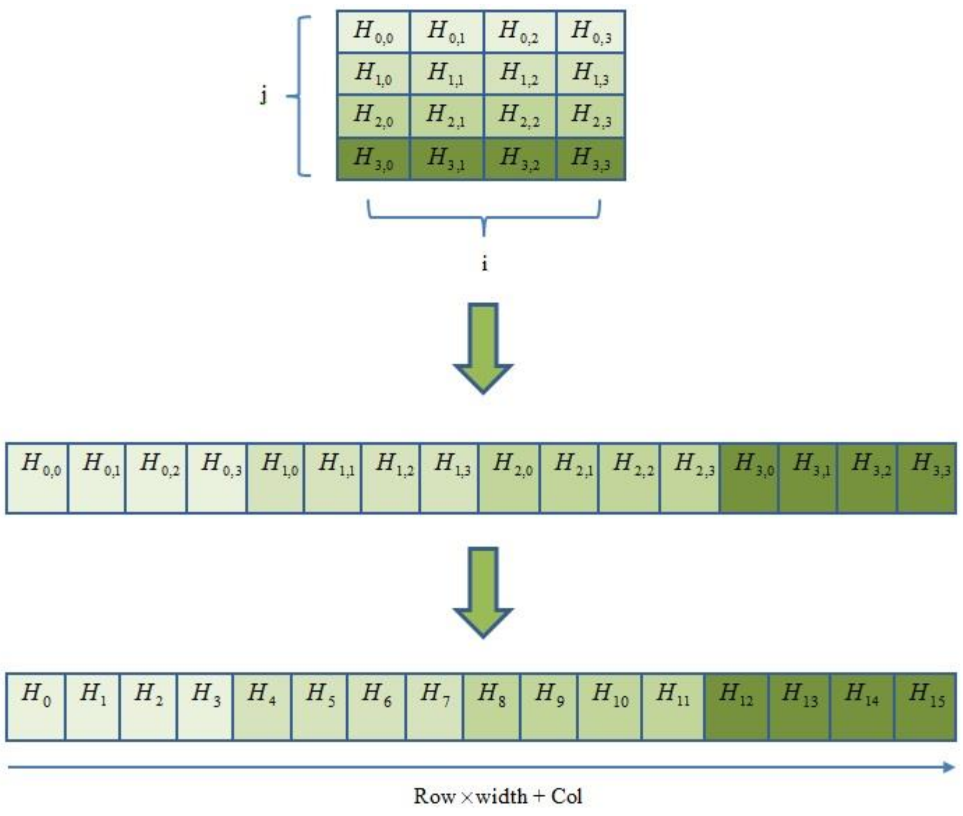


Figure 7.23: Row-major layout for two-dimensional array flatted into one-dimensional array.

It is evident that two-dimensional array is flatted into single dimension array based on the row-major layout technique. Thus, to access the corresponding index of the *H* element in row j and column i of two-dimensional array the following formula can be used:

$$H[j][i] = H[Row \times width + Column]. \tag{7.4}$$

The width refers to the width elements of array in each row, of which there are four in H[j][i] array. For instance, the equivalent index for $H_{3,2}$ is $3 \times 4 + 2 = 14$. In the literature the width of array is sometimes called “stride” of array.

Furthermore, the linearization of 2D arrays can be extended to 3D by considering another dimension. The linearized access to the *H* three-dimensional array can be written by the following formula:

$$H[k][j][i] = H[k \times N_j \times N_i + j \times N_i + i]. \tag{7.5}$$

Therefore, with row-major layout, multi-dimensional arrays/matrixes can be allocated into single dimensional array in global memory (DRAM) of GPU, which comprises the first row, followed by the second row, followed by the third row, and so on.

An important point in memory layout refers to coalesced memory. Considering coalesced memory is one of the important optimisation techniques that can be applied to efficient implementation of the GPU code. Coalesced memory is a technique that combines and groups non-sequential, small data transactions and global memory reads into larger and sequential data transactions. This technique helps the GPU execution to be more efficient in terms of global memory accessibility and allows consecutive threads within a warp to access contiguous memory with larger data transactions and bigger memory reads. In this work, this technique is considered to combining and linearizing three-dimensional arrays into one-dimensional arrays by using thread ID with respect to x, y, z coordinates (section 6.3.1). In addition, this way of storing data in global memory indicates SoA memory layout. As noted in section 7.1, the memory access patterns are very important for achieving high performance in the program on any parallel architecture and considering appropriate data layout for accessing memory has a direct impact on performance. Better and effective vectorisation and parallelisation can be provided when SoA memory layout maintains the unit-stride memory and consecutive memory locations. In order to uphold coalesced access to global memory and to prevent misaligned access to memory padding [177] is used in SoA layout. Figure 7.24 shows the difference between AoS and SoA memory layouts with padding.

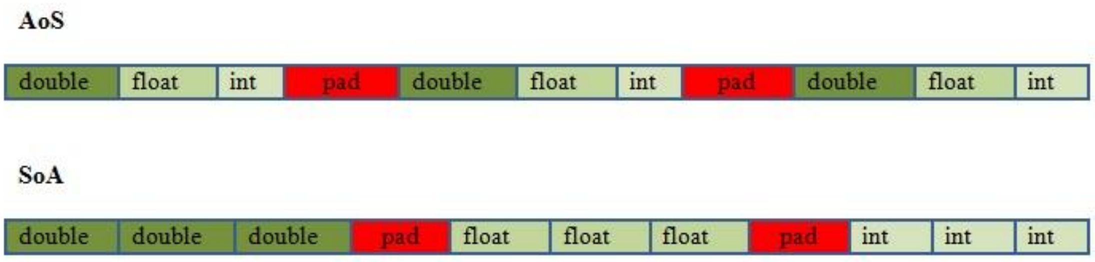


Figure 7.24: Difference between AoS and SoA.

Another factor in memory access optimisation related to coalesced memory is the combination and coalescing different requests of a warp into a single request to reduce DRAM bandwidth. This indicates that access to DRAM will be most effective when a warp (32 threads) load and access in the same 128-byte (in Kepler microarchitecture) segment or cache-line. Hence, the global memory can be accessed more efficiently by coalescing and combining all 32 requests from each thread in a warp into one request or transaction. To meet this requirement, alignment within streaming operations to a 128-byte boundary of cache-line is necessary. As discussed in section 7.1, alignment refers to the property of memory address that align a data stream to start at specific addresses, and can be determined as a numeric address to a power of 2 [178]. In alignment besides the address the data size is also important. When a datum address is aligned to its size it is known as “naturally aligned”.

However, to comprehend how the data alignment can be helpful for coalescing requests into single request, the investigation of loading operations from global memory is important. In general, there are two types of loads from global memory based on the Kepler GK104 microarchitecture: caching (default) and non-caching. Caching is a default mode of GPU for loading from memory with 128-byte line, by which it tries to use L1, then L2, and then global memory. Non-caching is not the default mode of GPU and needs to be compiled with different configuration for utilisation. Non-caching mode tries to use L2 and then global memory. In the present study, only caching load is considered. To understand this behaviour two scenarios are taken into account: cache loading with aligned requests and cache loading with misaligned request. In the first scenario, all 32 requests from each thread in a warp are aligned with consecutive 4-byte data size. The following figure presents cache loading with aligned requests.

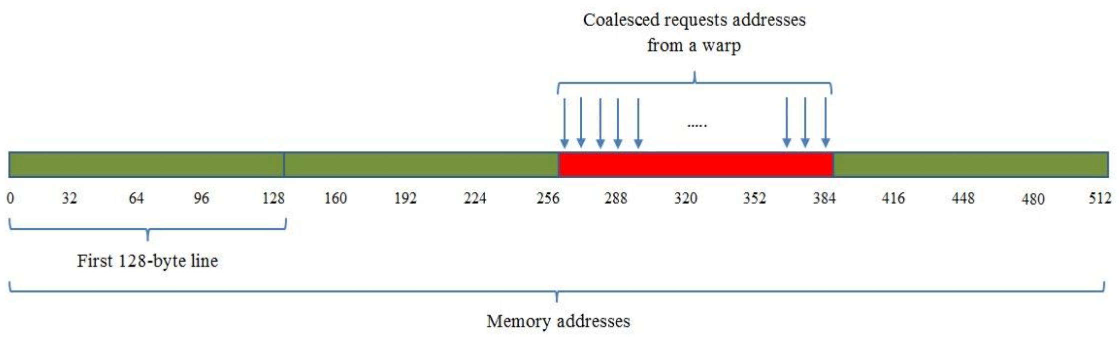


Figure 7.25: Aligned cache loading.

From the previous figure, all of the 32 requests from each thread in the warp are combined and coalesced into a single 128-byte cache-line or segment and the whole thread addresses of a warp are allocated within one segment. It is evident that in the first scenario addresses from a warp can be allocated when permuted (not in consecutive format) in one cache-line. In contrast, the second scenario illustrates all 32 requests from each thread in a warp when misaligned.

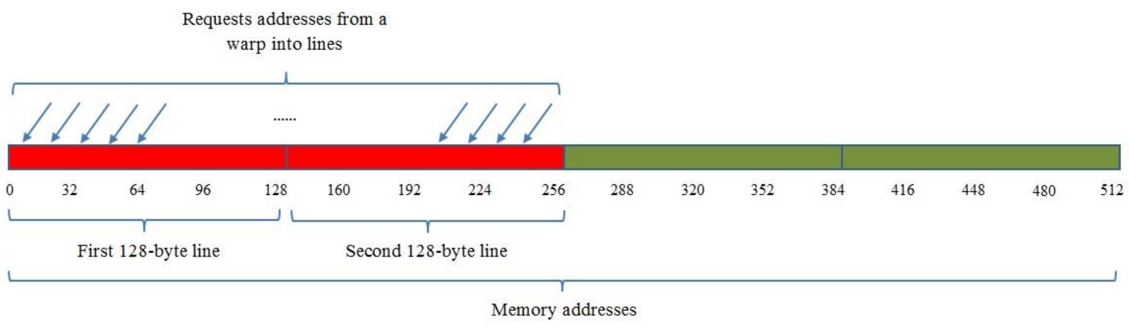


Figure 7.26: Misaligned cache loading.

Figure 7.26 shows all of the 32 requests in a warp when not coalesced into a single segment; they are combined into two cache-lines and the all thread addresses of a warp are allocated within two segments resulting in performance losses. Therefore, alignment and stride play very important roles in optimisation of memory access pattern and memory coalescing. Even though, these factors are essential features of optimisations of memory, striding through DRAM when dealing with multi-dimensional arrays is difficult and problematic due to the large strides between threads in a warp and between elements of array. For accessing multi-dimensional matrix, it is essential for threads to index the higher dimensions of the matrix which causes large strides between threads. To address this issue utilising on-chip shared memory is considered. Additionally, matching the x direction of the system to the innermost block size can be helpful for solving the problem of large stride. Another option 'shuffle' also exists, which will be addressed in the future work. Therefore, this issue is solved based on the coalesced transpose through shared memory. The implementation of the coalesced transpose through shared memory follows: (i) loading contiguous data from global memory into the shared memory from each thread within a warp; (ii) re-determining the array indices of shared memory; and (iii) synchronising all the threads, in order to make sure all the threads within a warp of one block complete calculations.

7.3.3 Communication and Data Transfer

Data transferring must be accomplished through PCI-Express interface (bus connectors) from CPU to GPU and vice versa. In CUDA programming model [131] this can be done by using cudaMalloc() and cudaMemcpy() functions. CudaMalloc() function is for allocating memory on the device and cudaMemcpy() function is for copying data stream from the host to the device and back again. The important point of data transferring or communication between CPU and GPU refers to the performance. In fact, multiple unnecessary Memcpy functions have a direct impact in reducing the performance. The following figure presents allocating memory in CUDA and copying data from CPU to GPU.

```
float* d_pxi;
  size_t size=Nz*Ny*Nx*sizeof(float);
  cudaMalloc((void**)&d_pxi,size)
```

```
float* h_pxi = malloc(size);
  // code to fill h_pxi
    cudaMemcpy(d_pxi,h_pxi,size,
      cudamemcpyHostToDevice);
```

Figure 7.27: Allocating an array on memory of GPU (left) and copying data from CPU to GPU (right).

Where d_pxi refers to the array on the device and h_pxi is an array on the host. The last term of cudaMemcpy function indicates the direction in which the data must be copied.

7.3.4 Threads and Thread Blocks Distribution

The number of threads that can be executed simultaneously in parallel varies according to GPU architectures and compute compatibilities. A maximum of 1024 threads are possible in a block in Kepler architecture with a minimum of 3.0 compute compatibility. In GPU devices there is a high amount of obtainable parallelism, and this large number of threads can be used to hide memory latency efficiently. On the other hand, on CPU devices this amount of parallelism is not accessible due to architectures whereby only instructions of one or two threads (with respect to HT) can be executed simultaneously in parallel. Therefore, to utilise this high volume of parallelism on GPU fine-grained distribution scheme for distributing threads and thread blocks based on the fine-grained spatial decomposition method must be considered. In fine-grained distribution the total number of threads and thread blocks are launched based on the domain size whereby each thread must be processed in each cell of the cube or grid-point of the system. It should be noted that according to the NVIDIA CUDA programming guide [131] the number of threads per block that can be used to achieve the best performance is recommended to be a multiple of warp size (32). Otherwise the GPU would waste the SM resources on active warp (e.g. registers) and increase the number of unusable and inactive threads in the last warp. In the whole computation processes of the CDS the total number of cubes' cell is determined by $N_T = N_z \times N_y \times N_x$. N_z, N_y and N_x are the numbers of cell/grid-point coordinates considered by each kernel function on GPU. Thus, the appropriate number of grids (grid size) is defined based on the following formulas in three dimensions:

$$\begin{aligned} DimGrid_z &= \left(N_z / DimBlock + 1\right) + \left(N_z \% Dimblock = 0 ? 0 : 1\right) \\ DimGrid_y &= \left(N_y / DimBlock + 1\right) + \left(N_y \% Dimblock = 0 ? 0 : 1\right) \\ DimGrid_x &= \left(N_x / DimBlock + 1\right) + \left(N_x \% Dimblock = 0 ? 0 : 1\right) \end{aligned} \tag{7.6}$$

DimBlock refers to the number of threads in a block which depends on the GPU architecture, simulation system size and available resources such as warps, shared memory and registers. The last term of the formula is for adding an additional block to avoid a lack of resources and to ensure that the DimGrid covers the entire system size. Percentage sign in the last part refers to modulo operator which gives the remainder of a division. Conditional expression (?:) in the last term is used for the evaluation of modulo operator, if the condition (remainder of modulo operator) is true (zero) then returns zero, otherwise returns one. Therefore, by considering this structure the kernels will perform for every grid cell of simulation domain from (1, 1, 1) to (N_z, N_y, N_x), regardless of the system size.

7.3.5 Heterogeneous CUDA-Based CDS Pseudo-Code from Logical Level View

As discussed earlier, in a single computer node GPU and CPU are discrete and separate processing components connected to each other through PCI-Express bus. This form refers to a heterogeneous system. In heterogeneous system computational tasks are processed based on heterogeneous computing, which divides the computational tasks between processing components to yield high performance. In this model of computing, data is initialised by the CPU/host and compute-intensive works are processed by GPU/device. As mentioned in chapter 6.3.3, due to the highly parallel nature of GPUs architecture and different types of memory on GPU, optimal performance can be achieved by executing intensive parallel works on GPU and fewer parallel or sequential works on CPU. Figure 7.28 illustrates CUDA-based CDS simulation pseudo-code.

```
Begin
Host program on CPU
1: Define variables
2: Read values of parameters from input files
3:  Allocate Host arrays
4:   Initialise random values for order parameter
5:   Allocate Device arrays
6:    Define time function
7:     Start to copy values from Host to Device
8:      Calculate number of threads and number of blocks based on the system size
          (number of cell)
9:       Discretised into three dimensions cube/grid and map each thread to one cell
  Kernel program implemented on GPU/Device
  10: Kernel 1 – Define and calculate boundary condition in x axis
  11:  Kernel 2 – Define and calculate boundary condition in y axis
  12:   Kernel 3 – Define and calculate boundary condition in z axis
    13:  For all time – steps --- (executes on the host)
  14:     Kernels 4 - 7 – Calculate first isotropised discrete Laplacian
  15:      Kernel 8 – Calculate Map function
  16:       Kernels 9 - 13 – Calculate second isotropised discrete Laplacian of the
  17:        Free energy functional and update boundary conditions
  18:      Kernel 14 – Calculate whole equation for $\psi(t+1,r)$ → Eq. (2.8)
19:       Copy back from Device to Host
20:      Reconstruct the data cell according to the results
21:     Write the outputs into files
22:    End for
23:   Free all Host and Device memory allocation
24:  Write the elapsed time
End
```

Figure 7.28: Heterogeneous pseudo-code of CUDA-based CDS simulation.

Figure 7.28 shows that the complex CDS simulation method is divided into a group of fundamental kernels to process the expensive computations and calculations. The major reasons for having a group of kernels that transfer a set of function calls from the host into the device are that it: (*i*) is easier to implement; (*ii*) is easier to maintain code accurateness; and (*iii*) maximises system performance by allocating enough resources for each kernel. Therefore, to obtain the maximum benefit of parallel computing of GPU, fourteen kernels were implemented for the calculations. According to the pseudo-code of CUDA program, each kernel has a responsibility to compute one part of the CDS simulation method in parallel. However, this method has limitations, as discussed with regards to kernel optimisation (kernel fusion) in section 7.3.7.

7.3.6 Performance Metrics and Results

This study used NVIDIA Quadro K5000 commodity based on the first generation of GK104GL Kepler microarchitecture. It comprises of eight streaming multiprocessors, each of which contains 192 cores. In total, there are two warp schedulers and two instruction dispatch units per each streaming multiprocessor, which are capable of launching and executing two warps concurrently and simultaneously [179]. Dual warp scheduler selects two warps and launches one instruction stream per clock cycle from each warp. Since warps execute independently in dual warp scheduler there is no dependency issue within the instruction stream and it can be obtained high performance. The warp scheduler helps to have an efficient share access to 64KB of configurable L1 cache/shared memory. Total numbers of multiprocessing streams have shared access to 512KB of L2 cache and 4GB of global memory. The GPU commodity clock rate is 706 MHz; a theoretical peak single precision performance refers to the 2.1 TFLOPS and the maximum memory bandwidth is 173GB/s.

There are two main metrics for specifying GPU performance: maximum computational performance and memory bandwidth. Maximum computational/arithmetic performance refers to the measurement of computational capability by calculating the number of floating points either single or double can be processed per second. Memory bandwidth indicates the amount of data that can be accessed in a specific time (seconds). It can be specified whether the program is compute or memory bounds by considering the ratio between a maximum arithmetic performance (single precision) and a peak memory bandwidth. In NVIDIA K5000 the ratio of these factors is: FLOP $\div$ bandwidth = 12.1. This value indicates that for increasing arithmetic throughput and computer resources, at least 13 floating point operations per memory access should be performed to achieve high FLOP performance. There are three main factors that must be considered to obtain high performance on GPU application: (*i*) decreasing/hiding instruction and memory latency; (*ii*) efficient utilisation of memory bandwidth; and (*iii*) efficient utilisation of computer resources or arithmetic throughput. Not considering these factors will be caused performance bound which are respectively: (*i*) latency bound; (*ii*) memory bound; and (*iii*) compute resource bound. Latency bound refers to the situation when both memory and compute utilisation are low. Memory bound is a situation when memory access/load and store speed is high. This pertains to the time of execution of program depending on the memory loads. Compute resource bound indicates that the resources are not used efficiently e.g. a high proportion of disabled warps relative to the total number. To address the performance bound issues the following points have been considered:

1. *Memory bandwidth:* to reduce memory bandwidth bound appropriate memory access pattern, layout and alignment are considered. In addition, efficient usages/metrics of GPU memory hierarchy for each kernel are investigated.

2. *Arithmetic resources bound:* to address this problem, warp execution efficiency should be considered. Warp execution percentage refers to the average percentage of active threads in every executed warp. In fact, improving warp execution efficiency can increase the utilisation of the resources. According to the GPU CUDA, to fulfil this requirement all threads in a warp should have the same branching behaviour, and reduce divergent branches (e.g. if - then - else) as much as possible in each kernel.

3. *Latency bound:* to decrease the latency bound, the occupancy of each streaming multiprocessor is important. Streaming multiprocessor occupancy refers to the ratio of the number of active warps to the maximum number of possible active warps that can be assigned on a GPU's multiprocessor [180]. There are three main points which play important for theoretical occupancy of the multiprocessor: (*i*) number of threads per block; (*ii*) amount of shared memory usage per block; and (*iii*) number of register utilisation per thread. However, the main reason for low achieved occupancy is that all streaming multiprocessors on GPU have different execution time-line and they do not remain equally busy during execution of kernels because of synchronisations and existent dependencies.

Furthermore, the high levels of performance can be achieved at lower occupancy by increasing instruction level parallelism (ILP) according to Vasily Volkov [181]. To hide or decrease the latency it is necessary to have high occupancy (more concurrent warps) or high instruction level parallelism (more independent instructions in a thread). In addition, different techniques (e.g. loop unrolling) that can be applied in program to increase ILP are employed by optimising compiler [112]. Hence, this study consider mainly the exploiting the thread level parallelism which comprises of task and data parallelisms. The following formulas illustrate how to calculate different metrics of performance [182]:

$$Required\_warps = Latency \times Throughput. \tag{7.7}$$

The required-warps (amount of parallelism) refers to the required number of warps to achieve high performance or hide latency. Latency is the time needed to process an operation and throughput indicates number of operations that can be processed per second in each streaming multiprocessor. Arithmetic latency takes approximately 10-20 clock cycles and memory latency takes between 400-800 cycles. A warp performs one instruction that corresponds to 32 operations and Kepler architecture needs 3,840 (20×192) operations per streaming multiprocessor to obtain full arithmetic utilisation. By dividing 3840 by 32, 120 warps per streaming multiprocessor are required to achieve full utilisation of compute resources. The maximum memory bandwidth for Quadro K5000 commodity is 173GB/s and memory frequency/clock rate is 2.7GHz, and 1 Hz is specified as one clock cycle per second, therefore it is possible to calculate the memory bandwidth from gigabyte per second to gigabyte per cycle: $173 \div 2.7$ = 64 Bytes/Cycle. By multiplying byte per cycle by Kepler memory latency, $64 \times$ 800=51200 bytes or 52KB of memory I/O needed to obtain high memory utilisation. It should be noted that this value is for the whole GPU device, not for each streaming multiprocessor. Finally, a knowledge this value and amount of data that each thread uses, the number of threads and warps to hide memory latency can be calculated. For instance, if each thread utilises 4 bytes for computation, consequently the number of threads can be calculated by 52KB $\div$ 4 bytes/thread = 13,000 threads or 13000 threads $\div$ 32 threads/warp = 407 warps required to hide memory latency.

In CUDA hierarchy model instructions are executed within each core of streaming multiprocessor, when one warp becomes idle e.g. waiting for result, the streaming multiprocessor switches to other active warp to continue execution. By having enough warps the cores of the GPU/device can be kept busy or occupied all the time, hiding latency. The context of each warp will remain on-chip of streaming multiprocessor during the whole lifetime of the warp. Hence, the switching cost between warps is negligible.

$$Theoretical\_occupancy = \frac{Active\_warps}{Maximume\_warps},$$

$$Achieved\_occupancy = (Active\_warps \div Active\_cycles) \div (Max\_warps\_per\_SM), \tag{7.8}$$

The other important metric to be considered for hiding arithmetic resource bound refers to divergence or branch of the control flow. The GPU execution is based on the warp and each thread within a warp executes same instruction. If however, some threads within a warp start to execute different instructions or take different paths, warp divergence will happen and degrade performance. Warp divergence or branch causes the serial execution of those threads which take a path (condition is true) and the rest of threads will be disabled until the primary condition becomes false. Thus, it is essential to reduce as much as possible the number of different execution branches within the same warp. Branch efficiency formula can be used to calculate the efficiency of warp divergence. Branch efficiency refers to the ratio of non-divergent of branches to total number of branches.

$$Branch\_efficiency = \left(\frac{branches - divergent\_branches}{branches} \right) \times 100. \tag{7.9}$$

It is also essential to achieve high memory metrics to avoid/hide memory bound and obtain high performance. To calculate the memory load efficiency of the kernel should first consider global memory throughput. Global memory load requested throughput refers to the number of data requested by instructions from the GPUs' global memory, and global throughput is a memory read productivity of the kernel. Finding global memory load helps in calculating the global memory efficiency, which refers to the percentage of coalesced access from global memory. A value of 100% indicates that all accesses are perfectly coalesced, representing the maximum utilisation of global memory bandwidth. Global memory load efficiency is the ratio of requested global memory load throughput to the required global memory load throughput.

$$Global\_efficiency = \left(\frac{global\_requested\_throughput}{global\_throughput} \right) \times 100. \tag{7.10}$$

Table 7.6 Compiler configuration and CUDA specification used for executing the CDS method used in this study.

GPU - CUDA Specifications	
Device	NVIDIA Quadro K5000
Architecture	Kepler GK104GL
CUDA Compute Compatibility	3.0
CUDA Driver Version	6.5
Compiler Configurations	
Operating System	Linux - OpenSUSE 12.3
Compiler	NVIDIA - NVCC
Compiler Version	6.0.1
Compiler Flags	-O3 -arch="sm_20"

Table 7.6:CUDA specifications and compiler configurations for the CDS simulation.

Although it is more common to have the same value of the architecture flag (–arch=”sm_30”) of compiler with compute compatibility (3.0) it was found that the architecture flag of compiler with “sm=20” yields better performance for CUDA-based CDS implementation. This may be due to the utilisation of 32-bit pointers for compiling, which needs less register, and faster utilisation of mathematical functions with fewer precision points.

Table 7.7 presents the reference result as a baseline for performance of the CDS simulation for $64 \times 64 \times 64$ system size in 100,000 time-steps, without any shared memory. The following results are calculated with the help of NVPROF (NVIDIA command-line profiler), which enables gathering of the performance metrics for GPU kernels, memory transfers and CUDA activities.

No. threads	**No. blocks**	**Av. branch efficiency %**	**Total no. registers (kernels)**	**Av. total global load throughput GB/s**	**Av. global load efficiency %**	**Achieved global store efficiency %**	**Av. achieved occupancy (kernels) %**	**Elapsed time (Seconds)**
16,4,16	5,17,5 (425)	99.71	250	71.21	94.54	90.28	77.54	250.945
16,16,4	5,5,17 (425)	99.71	250	71.90	94.54	90.28	76.62	248.644
32,2,16	3,33,5 (495)	99.57	250	73.09	95.18	90.28	77.79	244.583
32,16,2	3,5,33 (495)	99.57	250	75.09	95.18	90.28	76.97	241.452
64,2,8	2,33,9 (594)	99.54	250	68.36	95.08	90.28	64.28	265.637
64,8,2	2,9,33 (594)	99.54	250	68.66	95.18	90.28	64.28	257.616
128,2,4	1,33,17 (561)	99.53	250	65.09	95.08	90.28	65.18	277.083
128,4,2	1,17,33 (561)	99.53	250	65.22	95.18	90.28	64.94	275.347

Table 7.7: Comparison of CUDA execution configurations and performance metrics for $64 \times 64 \times 64$ system size.

Table 7.7 shows that different CUDA configurations have different impacts on the execution time and performance. All the systems have run 1024 threads per block, but with different numbers of blocks per grid. These differences in the number of blocks are caused by different percentages of occupancy achieved and the other performance metrics. The fourth case obtains the best execution time with a total of 495 thread blocks; this is higher than in the first case (425), but less than for the fifth (594) and seventh (561) cases. This illustrates that having a high number of thread blocks does not always give the best results in terms of occupancy achieved, therefore the other criteria play an important role in performance. For instance, the achieved occupancy for the first case is higher than for the fourth, however the global memory throughput is less in the latter. The other important point found based on the results, relates to the value of the innermost dimension of a thread block. In fact, choosing the lowest value/number for the innermost dimension helps to improve the global memory throughput, which has a huge impact on the performance. The main reason for this phenomenon could be the memory access pattern. The logical layout of a three-dimensional thread block was flatted into one-dimensional physical arrangement by utilising the x dimension as the innermost dimension, y as the second dimension and the z as the outermost dimension. As mentioned in section 7.3.2, coalescing and stride are important factors in memory arrangement and access pattern. Therefore, by choosing the lowest number for the innermost dimension the DRAM bandwidth can be reduced, coalescing different requests of a warp into single request and solving the large stride issue. As the execution of the thread block is based on the warp execution and the dimension of thread block should always be multiple of the warp size (32), thus, a thread block which is a multiple of the warp size can prevent inactive threads in the last warp and enhance the occupancy. It should be noted that all results of metrics are calculated based on the average of total number of kernels (14) for the CDS simulation program and they do not represent any single kernel. The total numbers of registers per thread for instance refers to the sum of all registers used in all kernels per thread.

The following table shows different CUDA execution configurations for $128\times128\times128$ system size in 100,000 time-steps without any shared memory usage.

No. threads	**No. blocks**	**Av. branch efficiency %**	**Total no. registers (kernels)**	**Av. total global load throughput GB/s**	**Av. global load efficiency %**	**Achieved global store efficiency %**	**Av. achieved occupancy (kernels) %**	**Elapsed time (Seconds)**
16,16,4	9,9,33 (2673)	99.91	250	80.27	96.90	94.85	80.57	1672.49
32,16,2	5,9,65 (2925)	99.85	250	94.63	98.36	94.85	80.92	1422.35
64,8,2	3,17,65 (3315)	99.85	250	85.67	98.36	94.85	70.14	1641.62
128,4,2	2,33,65 (4290)	99.85	250	68.27	98.36	94.85	59.82	1881.55

Table 7.8: Comparison of CUDA execution configurations and performance metrics for $128\times128\times128$ system size.

Table 7.8, shows that the second case with total number of 1024 threads and 2925 thread blocks has gained the best results in terms of execution time, occupancy and global memory throughput. As mentioned earlier, each streaming multiprocessor contains a group of 32-bits (in total 64K for Kepler architecture) registers that can be distributed across threads and a fixed volume of shared memory that can be spread amongst the blocks. Therefore, the available number of registers and shared memory has direct impact on the number of thread block and warps that can simultaneously and concurrently assign on a streaming multiprocessor for a specific kernel. The resource availability (register and share memory) restricts the number of occupant blocks per streaming multiprocessor. In other words, the occupancy is specified by the number of registers and the volume of shared memory utilised by every block. For instance, kernel 8 for $64 \times 64 \times 64$ system size uses 12 registers per thread and the total number of thread blocks is 495, hence $496 \times 12 = 5952$ registers used for each block. This leads to the number of blocks per streaming multiprocessor $64 \times 1024 / 5952 = 11$. With the benefit of CUDA occupancy calculator, the occupancy of streaming multiprocessor can be calculated by a chosen device kernel. The impact of different block sizes and different number of registers per thread on GPU occupancy are shown in the figures. The red triangles on the following figures present the achieved streaming multiprocessor occupancy based on the chosen number of threads per block and register per thread.

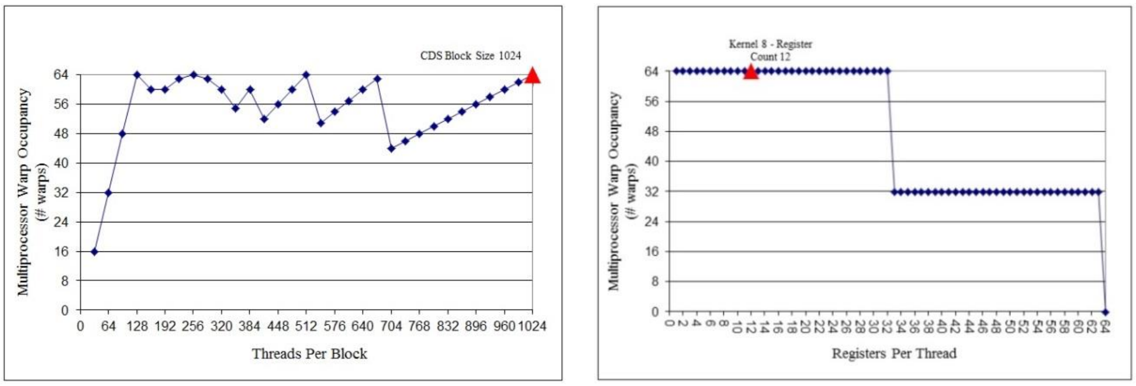


Figure 7.29: Occupancy of kernel-8 based on 1024 threads (left) and 12 register count (right).

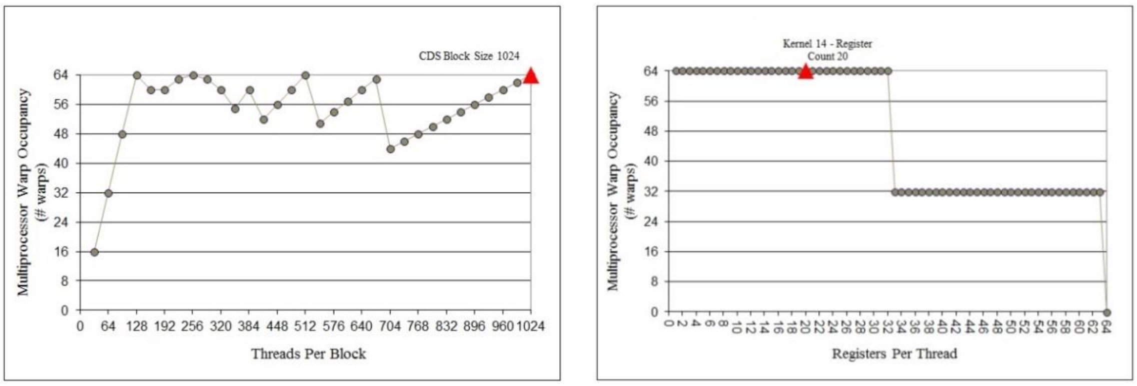


Figure 7.30: Occupancy of kernel-14 based on 1024 threads (left) and 20 register count (right).

It can be seen from the figures that by utilising 1024 threads per block and having the register number between the ranges of 1 to 32 per thread the highest number of occupancy can be obtained for each streaming multiprocessor. In this instance a high number of occupancy with the mentioned specifications indicates that the total number of active threads per streaming multiprocessor is 2048, the total number active warps per streaming multiprocessor is 64 and the number of active blocks per streaming multiprocessor is 2. According to Figures 7.29 and 7.30, high occupancy can be obtained with different numbers of threads, such as 128, 256 and 512 per block rather than only 1024 threads per block. However, the other important features to achieve high occupancy and hide latency include the number of active warps per streaming multiprocessor and the number of active thread blocks per streaming multiprocessor. The following table displays GPU occupancy data based on the different threads number. As discussed previously, the maximum numbers of active threads per SM and active warps per SM for commodity NVIDIA Quadro K5000 GPU are 2048 and 64, respectively.

GPU Occupancy Data	**128 - Threads**	**256 - Threads**	**512 - Threads**	**1024 - Threads**
Number of active Thread Blocks per SM	16	8	4	2

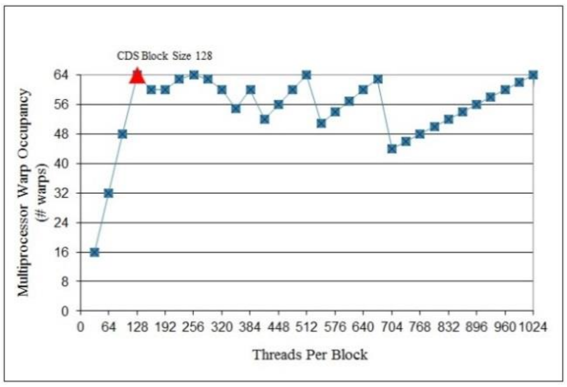
Table 7.9: Comparison of GPU occupancy data based on different threads number.

Table 7.9 shows the numbers of active thread blocks that can execute concurrently on one streaming multiprocessor and hide the latency for total threads number of 128 is higher than the others. Consequently, 128 threads rather than another number of threads is chosen as a thread block size for executing the performance metrics mentioned earlier for two system sizes of the CDS simulation method.

No. threads	**No. blocks**	**Av. branch efficiency %**	**Total no. registers (kernels)**	**Av. total global load throughput GB/s**	**Av. global load efficiency %**	**Achieved global store efficiency %**	**Av. achieved occupancy (kernels)**	**Elapsed time (Seconds)**
$64\times64\times64$								
16,4,2	5,17,33 (2805)	99.72	250	90.29	94.54	90.28	84.43	222.34
$128\times128\times128$								
16,4,2	9,33,65 (19305)	99.91	250	96.54	96.91	94.85	85.58	1361.57

Table 7.10: Comparison of CUDA execution configurations and performance metrics.

Table 7.10 shows that better performance can be achieved with 128 threads and total numbers of 2805 and 19305 thread blocks for $64\times64\times64$ and $128\times128\times128$ system sizes compared to using 1024 threads or any others. The main reason for this is that the number of active thread blocks per streaming multiprocessor helps to hide the latency. It should be noted that according to the CUDA programming guide [131], the amount of required occupancy that can saturate the latency depends on the computational problem and increasing occupancy does not always give higher performance. In fact, sometimes increasing occupancy by adding more registers, divergent branches and additional instructions reduces the performance and increases the wall-clock time of the kernels executions. However, to appoint an accurate volume of occupancy it is necessary to have a good balance between compute and memory utilisations. Figure 7.31 illustrates the occupancy of kernel 14 based on the 128 threads and 20 registers per thread.



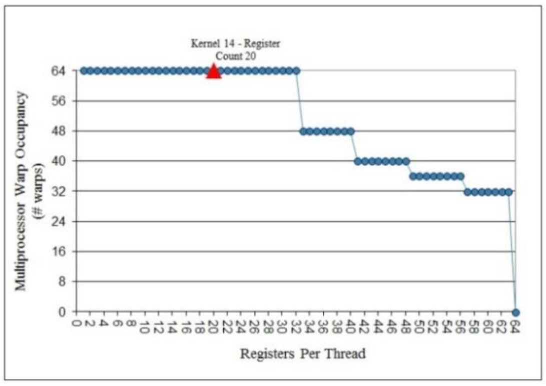
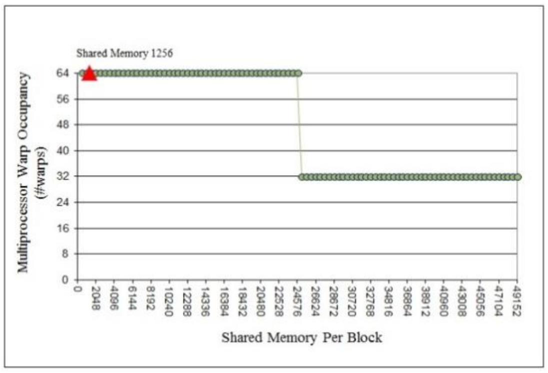


Figure 7.31: Occupancy of kernel-14 based on 128 threads (left) and 20 register count (right).

It can be seen from Figure 7.31 and Table 7.9 that using 128 threads per block and considering the register number between the ranges of 1 to 35 per thread the highest number of occupancy and active thread blocks can be achieved for each streaming multiprocessor. The main difference between Figure 7.31 and Figure 7.30 is number of registers that can be used to obtain the high occupancy.

The other important point for achieving high performance and occupancy for CUDA application is utilisation of shared memory. As discussed earlier, shared memory is on-chip memory, which is much quicker than the global memory. The maximum amount of shared memory for the commodity used in this study and Kepler microarchitecture with compute compatibility 3.0 is 49152 bytes that can be assigned per thread block, hence the whole threads within the block can access the same shared memory. The threads within the same block can share and access each other’s data through shared memory loaded from global memory. This offers a promising opportunity and capability to obtain high performance in parallel algorithm and solve coalescing of global memory and large strides issues. It should be noted that the utilisation of shared memory is useful and beneficial for repeat data access within the same thread or different threads in the same block. In addition, to avoid race conditions between threads within the block a barrier synchronisation should be considered between them. The following figure shows the impact of varying shared memory usage per block for 1024 and 128 threads and the amount of shared memory (1256 bytes) used in this work.



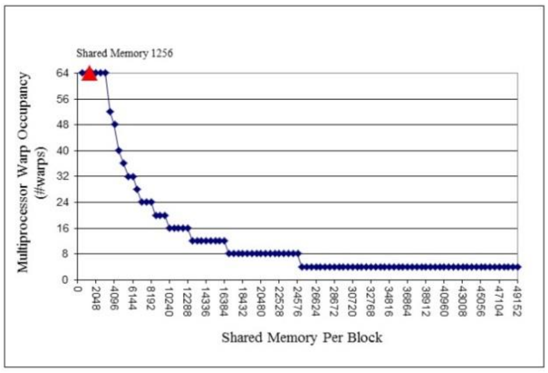


Figure 7.32: Amount of shared memory and the warp occupancy for 1024 threads (left) and 128 threads (right) per block.

When the block size is 1024 the amount of shared memory usage can be increased to 24576 bytes to achieve high occupancy without losing any occupancy. However, for 128 threads per block the volume of optimum shared memory usage is dropped to 4563 bytes. Table 7.11 illustrates different performance metrics based on 128 threads per block and 1256 bytes shared memory for $64\times64\times64$ and $128\times128\times128$ system sizes and 100,000 total time-steps.

No. threads	**No. blocks**	**Av. branch efficiency %**	**Total no. shared memory**	**Achieved global store efficiency %**	**Av. total global load throughput GB/s**	**Av. global load efficiency %**	**Av. achieved occupancy (kernels) %**	**Elapsed time (Seconds)**
$64\times64\times64$								
16,4,2	5,17,33 (2805)	99.72	1256	92.36	91.38	94.54	84.64	211.56
$128\times128\times128$								
16,4,2	9,33,65 (19305)	99.91	1256	95.95	97.56	96.91	86.37	1343.53

Table 7.11: Comparison of CUDA execution configurations and performance metrics with total 1256 shared memory usage and 253 registers.

From the table above it can be seen that by utilising shared memory the execution times are reduced, yielding better performance metrics. Although the shared memory usage improves the performance by solving large stride issue, its effect on the performance and specifically on the execution times is not significant. The main reason is that the implementation of CUDA-based

CDS simulation is not memory bound. If any program is limited by memory (memory bound) then using shared memory will have a huge impact on performance. Since all these metrics are reported based on the entire CUDA-based CDS executions, and there is no reported specific kernel separately and individually, to understand the execution behaviour of kernels better, two important kernels distinctly are taken into account with the help of NVIDIA NSIGHT visual profiler [75] for $64\times64\times64$ domain size. The following table presents all the execution configurations (block size, register usage, and shared memory usage) for kernels 8 and 9.

Occupancy per SM - Grid Size: [5,17,33] – Block Size: [16,4,2]		
	Kernel-8	**Kernel-9**
Variable	Achieved	Achieved
Active Warps	51.95	54.25
Occupancy	82.21%	84..82%
Threads		
Variable	Used	Used
Threads/Block	128	128
Registers		
Variable	Used	Used
Registers/Thread	13	26
Registers/Block	2048	4096
Shared Memory		
Variable	Used	Used
Shared Memory/Block	256	364

Table 7.12: Kernels 8 and 9 execution configurations for $64\times64\times64$ system size.

Figure 7.33 displays the execution time of GPU for two different system sizes in different time-steps using shared memory. Noted that all executions are performed on the same GPU architecture and floating point (single precision).

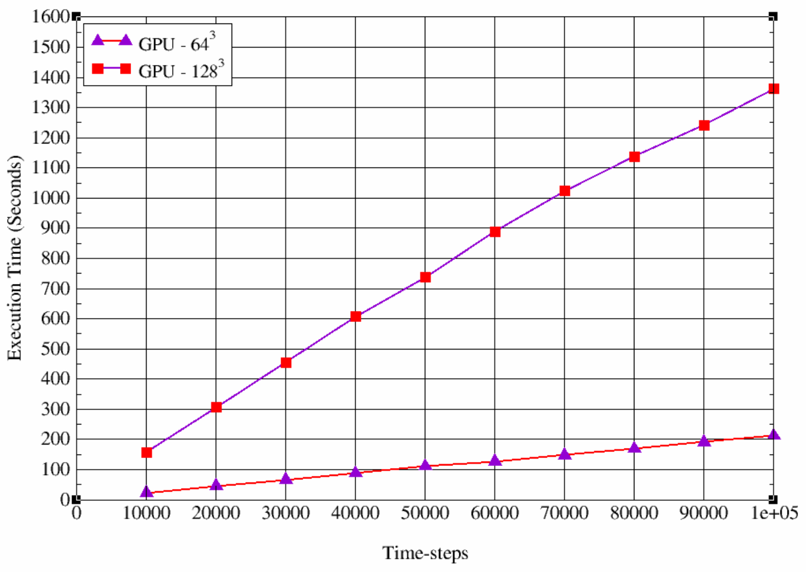


Figure 7.33: GPU elapsed times for $64\times64\times64$ and $128\times128\times128$ system sizes.

7.3.7 Kernel Fusion and Results of Performance Comparison

As discussed in section 7.3.5, the original idea for the logical and hardware levels of parallel implementation of the CDS simulation scheme on GPU was based on the decomposition of computational problems into a set of fundamental kernels to process the expensive computations and calculations of each equation in the CDS method. Although this idea has some advantages, such as the ease of implementing and maintaining the code, it also has some limitations. One of the main limitations for decomposition of computational problems into fundamental kernels is the reusability and accessibility of data for shared and global memories usages. GPU memories are classified into off-chip memory, which has larger size, lifetime of program, and higher latency; and on-chip shared memory, which is quicker, and has lower latency, smaller size and shorter kernel lifetime. On-chip memory should be used mainly when the frequent accessibility of data is high during the computation, therefore shared memory as on-chip memory can be utilised only for intermediate data during the kernel execution (lifetime of kernel) of the GPU, and for passing data between kernels; data must be stored in off-chip (global) memory. Consequently, to achieve high compute throughput of GPU and to exploit more parallelism, it is necessary to reduce the number of access of off-chip memory (global memory bandwidth) and to execute enough operations per data stream (stored or passed from the off-chip memory). However, it should be noted that adding more computations into a single kernel is not always useful and helpful [183]. Due to the finite available resources per streaming multiprocessor, adding more computations increases the utilisation of resources thus limiting the amount of occupancy and parallelism. Hence, determining the ideal distribution of computational problem into kernels is not an easy task.

To address the limitation of distributing computational works into a group of kernels, a fusion method can be considered. In this scheme, first all the computational works developed as standalone kernels are investigated and then those kernels that have more opportunity to improve the performance with respect to data locality are fused into one kernel. Different studies [183, 184] are also investigated the fusions of kernels in different fields. Wu et al. [185] considered fusing kernels to decrease the amount of memory bound in the GPU application. In [186], the authors alluded to the kernel fusions for improving the energy efficiency of GPU by reducing power consumption. Furthermore, these studies are mainly considered kernel fusions for memory bound GPU applications. However, the CUDA-based CDS method is not a memory bound application and the fusing of kernels is considered to improve performance and increase data reusability on on-chip memory. Thus, the following are the main advantages of kernel fusions:

1. Reducing the number of kernels which has a direct relation with kernels overhead. In fact, by fusing kernels, the amount of work that can be processed by fused kernel will

be increased, therefore reducing the total number of function/kernel calls and kernels overheads.

2. Decreasing the number of off-chip memory access or data transfer from or to DRAM.
3. Improving performance by having more transactions in flight to hide latency.
4. The impact of implicit optimisations by compiler can be increased due to the higher number of instruction in complex kernel.

The kernel fusion is sometimes not straightforward, and requires deep understanding of function/kernel behaviours and their impact on the entire program, whose original functionality should be maintained (as a primary specification). Therefore, the main challenges for kernel fusions are:

1. Utilising more resources such as shared memory and registers can reduce occupancy.
2. Different kernels use different optimal numbers of threads, and consequently different numbers of active warps to process data elements. Fusing kernels can cause suboptimal numbers of warps and threads, which reduces the performance of the program.
3. Sometimes it is difficult to maintain and evaluate the correctness of code due to the complexity of fused kernel.
4. Race condition can arise due to omitting the implicit global barrier between kernels in kernel fusions.

An example of kernel fusion is shown in Figure 7.34. Loads refer to the accessing data from global memory and saving into shared memory. Compute is computational processing of data within shared memory and store refers to the storing final results into global memory.

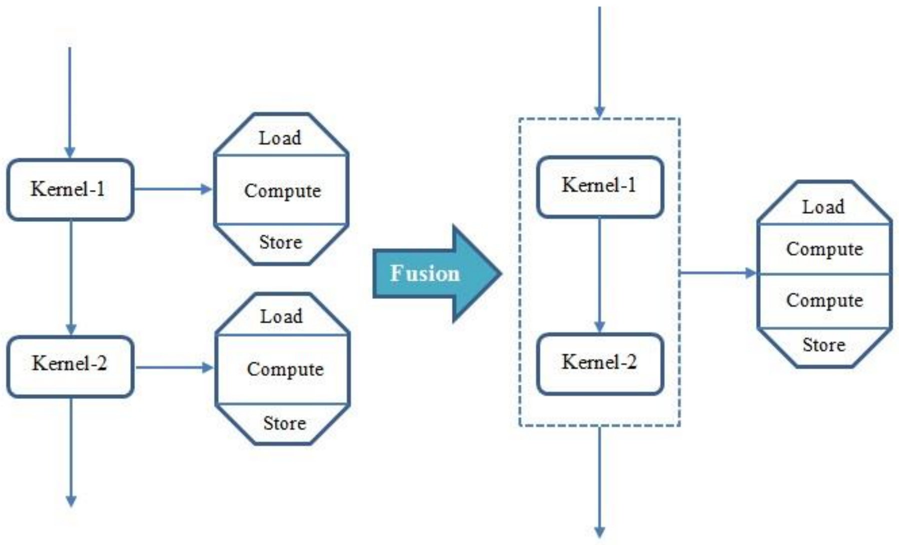
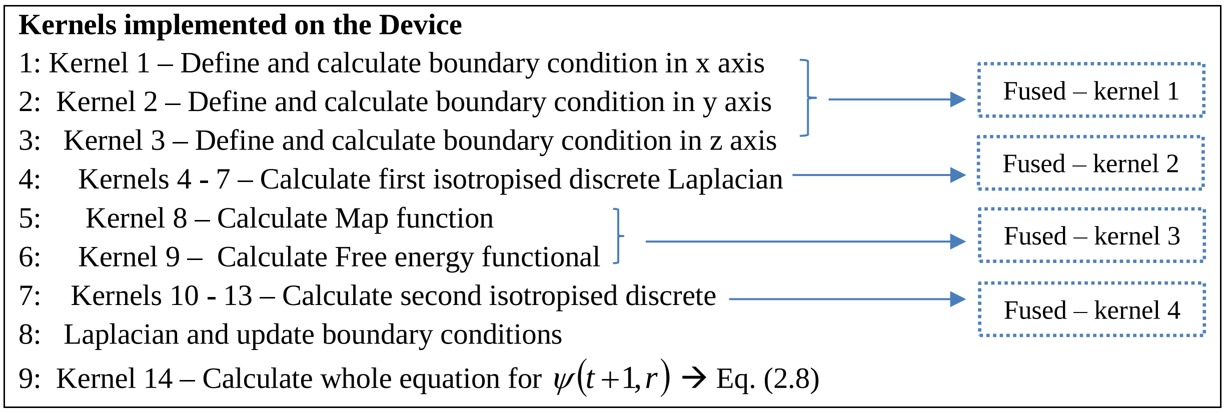


Figure 7.34: Illustration of kernel fusion.

Kernel fusion was based on the performance investigation of each kernel and consideration of data dependency between kernels. Therefore, the first step with the help of NVIDIA profiler is to analyse and profiler the performance of each kernel, identifying those that have more opportunity to improve performance when combined into one kernel. In the second step data dependencies between chosen kernels are considered. In this step only those kernels that can be securely fused without changing the semantics of the entire program (data dependencies) are selected. After investigating the mentioned steps in CUDA-based CDS method, it was found that nine kernels have the opportunity to be fused without influencing the data dependencies of the entire program. The following figures present the kernel fusions and the whole pseudo-code of CUDA-based CDS simulation after combining kernels.



Kernels implemented on the Device

```
1: Kernel 1 – Define and calculate boundary condition in x axis
2:  Kernel 2 – Define and calculate boundary condition in y axis
3:   Kernel 3 – Define and calculate boundary condition in z axis
4:     Kernels 4 - 7 – Calculate first isotropised discrete Laplacian
5:      Kernel 8 – Calculate Map function
6:     Kernel 9 –  Calculate Free energy functional
7:    Kernels 10 - 13 – Calculate second isotropised discrete
8:   Laplacian and update boundary conditions
9:  Kernel 14 – Calculate whole equation for $\psi(t+1,r)$ → Eq. (2.8)
```

Figure 7.35: Kernel fusions of CUDA-based CDS scheme.

```
Begin
Host program on CPU
1: Define variables
2: Read values of parameters from input files
3:  Allocate Host arrays
4:   Initialise random values for order parameter
5:   Allocate Device arrays
6:    Define time function
7:     Start to copy values from Host to Device
8:      Calculate number of threads and number of blocks based on the system size (number of cell)
9:       Discretised into three dimensions cube/grid and map each thread to one cell
  Kernel program implemented on GPU/Device
  10: Kernel 1 – Define and calculate boundary conditions in x, y, z axes
    11: For all time – steps --- (executes on the host)
  12:     Kernel 2 – Calculate first isotropised discrete Laplacian
  13:      Kernel 3 – Calculates Map function and free energy functional
  16:       Kernel 4 – Update boundary conditions and calculate second isotropised
  14:       discrete Laplacian
  15:      Kernel 5 – Calculate whole equation for $\psi(t+1,r)$ → Eq. (2.8)
```

```
16:       Copy back from Device to Host
17:       Reconstruct the data cell according to the results
18:      Write the outputs into files
19:    End for
20:   Free all Host and Device memory allocation
21:  Write the elapsed time
End
```

Figure 7.36: Pseudo-code of CUDA-based CDS simulation based on the kernel fusions.

Table 7.13 shows different performance metrics and the execution time for $64\times64\times64$ and $128\times128\times128$ system sizes in 100,000 time-steps based on the kernel fusion pseudo-code and without considering any shared memory usage.

No. threads	**No. blocks**	**Av, branch efficiency %**	**Av. total global load throughput GB/s**	**Av. global load efficiency %**	**Achieved global store efficiency %**	**Av. achieved occupancy (kernels) %**	**Elapsed time (Seconds)**
$64\times64\times64$							
16,4,2	5,17,33 (2805)	99.98	94.75	99.52	90.28	78.62	202.53
$128\times128\times128$							
16,4,2	9.33.65 (19305)	99.97	99.25	99.93	94.85	81.28	1288.31

Table 7.13: Comparison of CUDA execution configurations and performance metrics based on the kernel fusions with total number of 125 registers.

From the above table it can be seen that the occupancy percentage is decreased compared to that shown in table 7.10, which is not based on the kernel fusion algorithm and shared memory. The main reason for this is the increased number of register usage (35) per thread for the kernels 2 and 4. However, it is clear that the averages of branch efficiency, global throughput and load efficiency are increased, which affects performance. In addition, fusing kernels decreases the total numbers of kernel calls, thus kernel overheads are reduced. In fact, reducing the kernel overhead has a significant impact on the execution time. The following table illustrates performance metrics and the execution times for two system sizes based on the kernel fusion algorithm by considering shared memory usage.

No. threads	**No. blocks**	**Av, branch efficiency %**	**Total no. shared memory**	**Av. total global load throughput GB/s**	**Av. global load efficiency %**	**Av. achieved occupancy (kernels) %**	**Elapsed time (Seconds)**
$64\times64\times64$							
16,4,2 (5,17,33)	5,17,33 (2805)	99.98	512	93.51	99.51	78.84	195.52
$128\times128\times128$							
16,4,2	9.33.65 (19305)	99.97	512	98.25	99.93	81.47	1257.59

Table 7.14: Comparison of CUDA execution configurations and performance metrics based on the kernel fusions and shared memory usage with total 125 registers.

Table 7.14 illustrates that using shared memory helps to reduce slightly the execution time, promoting better performance in different metrics. It should be noted that the percentage of occupancy can be improved by decreasing the register number per thread. The reduction of the register number to 27 per thread was considered, resulting in similar elapsed time for both shared memory and non-shared memory usages, therefore the results are not reported here. Figure 7.37 displays the execution time of GPU for two different system sizes in different time-steps using shared memory based on the kernel fusion and not kernel fusion scheme. It is clear that kernel fusion improves the performance of the system. It should be noted that all executions were executed on the same GPU architecture and floating point.

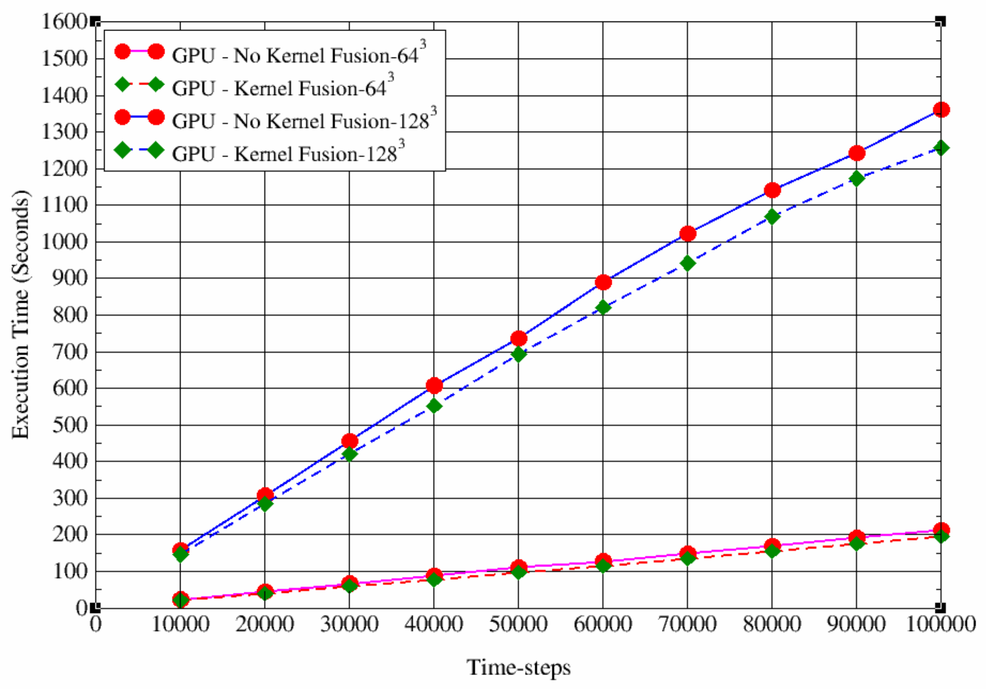


Figure 7.37: GPU elapsed times for $64\times64\times64$ and $128\times128\times128$ system sizes with and without kernel fusions.

Figure 7.38 illustrates the execution time comparison of the single-core, multi-core (six cores without hyper-threading), and many-core CDS implementation based on C and CUDA programming languages on CPU and GPU for $128\times128\times128$ system size. The main reason for this comparison is to show the system stability for executing simulations in different architectures (single, multi-core and many-core).

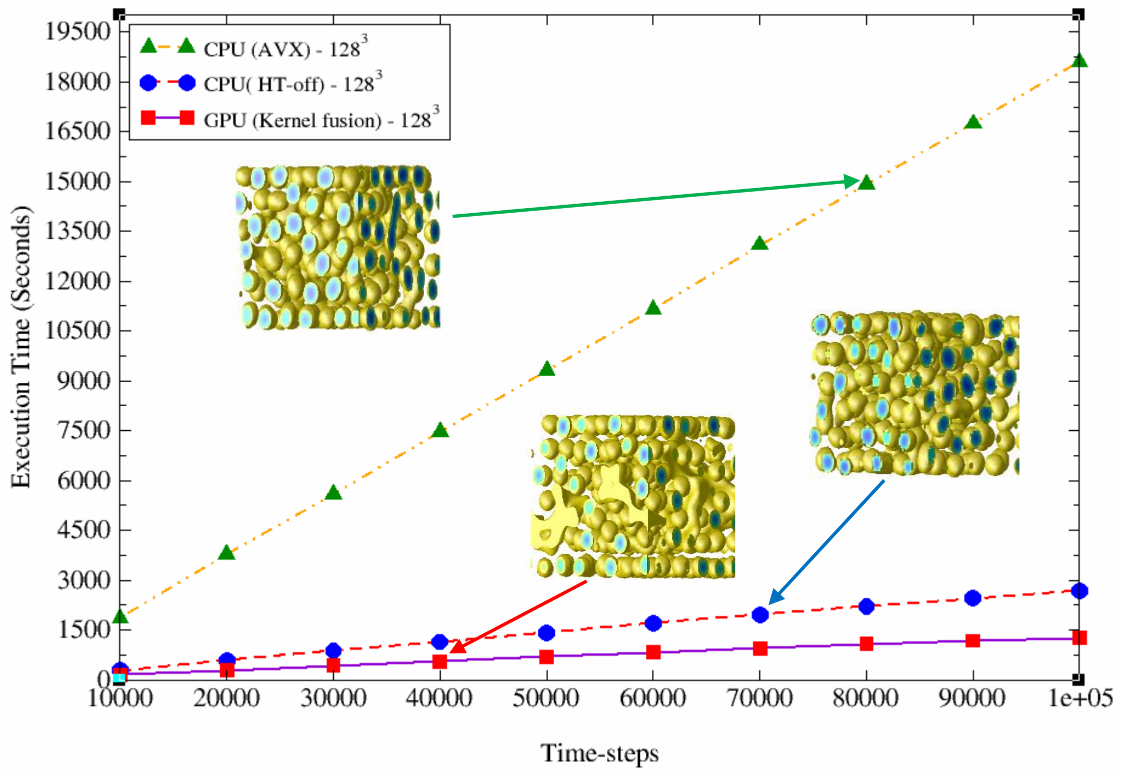


Figure 7.38: Speed comparison of single-core vs multi-core vs many-core GPU in $128\times128\times128$, arrows indicate diblock copolymer in time evolution.

From Figure 7.38, it can be seen that many-core GPU performs better when the number of time-steps and system sizes is increased. Table 7.15 compares the performance of the CDS simulation development based on the original (not optimised), AVX optimisation, and optimised multi-core implementations for two system sizes executed on an Intel Xeon E5-2420 processor.

Implementation/ System size	**Elapsed time (in seconds)**	**Speedup**		
		AVX – Orig.	**Multi-core – Orig.**	**Multi-core – AVX**
$64 \times 64 \times 64$ (HT – on)				
Original C (Sequential)	9810.55	<u>3.74x</u>	<u>28.19x</u>	<u>7.54x</u>
AVX	2623.04			
Multi-core	348.44			
$128 \times 128 \times 128$ (HT – off)				
Original C (Sequential)	83061.39	<u>4.45x</u>	<u>30.89x</u>	<u>6.92x</u>
AVX	18610.58			
Multi-core	2689.54			

Table 7.15: Comparison of execution times and speedups for CPU implementation with total 100,000 time-steps.

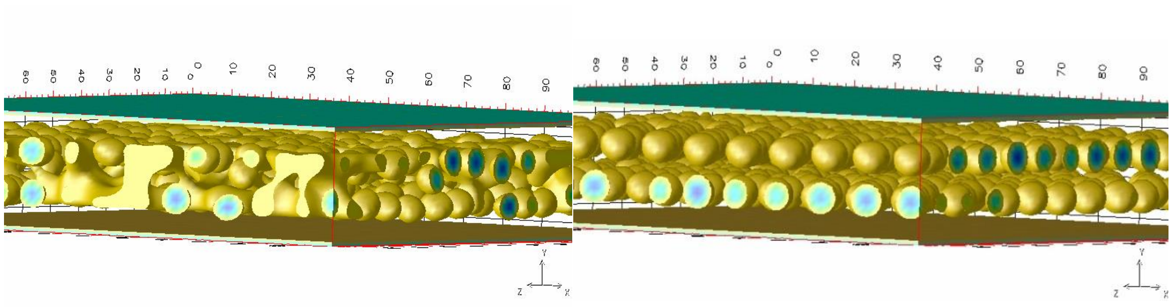
Table 7.15 shows that the multi-core CPU implementation provides the fastest execution time/speedup for the CDS simulation. In general, this shows that OpenMP as multi-threaded programming model is a suitable and appropriate tool to develop multi-thread program using the SIMD model of modern CPUs. However, according to the multi-core to AVX speedup volume between two system sizes, the speedup volume for the smaller size is 1.1x higher than the bigger domain size. This demonstrates that the GPU is more beneficial when dealing with large datum size. Table 7.16 displays the performance of the CDS simulation scheme development based on the original (not optimised), optimised multi-core, and GPU-CUDA kernel fusion implementations for two system sizes. The reader is reminded the elapsed time for the original (non-optimised), AVX optimisation and optimised multi-core implementations are the same as shown in Table 7.15.

Implementation/ System size	**Elapsed time (in seconds)**	**Speedup**		
		CUDA – Orig.	**CUDA – AVX**	**CUDA – Multi-core**
$64 \times 64 \times 64$				
GPU – CUDA	195.92	<u>50.31x</u>	<u>13.45x</u>	<u>1.78x</u>
$128 \times 128 \times 128$				
GPU – CUDA	1257.59	<u>66.08x</u>	<u>14.80x</u>	<u>2.14x</u>

Table 7.16: Comparison of execution times and speedups for CPU and GPU implementations with total 100,000 time-steps.

From Table 7.16, it can be comprehended that GPU–CUDA parallel execution is approximately ≈ 50, 14 and 1.78 times quicker than non-optimised single core, optimised single core, and multi-core executions on CPU for $64 \times 64 \times 64$ domain size; and ≈ 66, 15 and 2.14 times faster than non-optimised single core, optimised single core and multi-core processing for $128 \times 128 \times 128$ system size, respectively.

In addition, the other simulation of diblock copolymer sphere morphology with thin films (walls) under shear was performed to evaluate the accuracy of CUDA-based CDS method for different simulation parameters.



CUDA simulation result initial stage *t* =1000	CUDA simulation result final stage *t* =500000

Figure 7.39: CUDA simulation result of spherical morphology under shear.

The above figure shows the spherical morphology generated by CUDA-CDS implementation for a system of size $128 \times 26 \times 128$. The whole simulation was executed for up to 500,000 time-steps with 0.0003 shear rate and attractive wall to approach stable system (400,000 without shear and 100,000 more with shear). On account of CUDA hierarchy model and with respect to the system size, in total 910 blocks per grid in three dimensions (2, 7, 65), 1024 (128, 4, 2) threads per block, and 1256 bytes shared memory based on the kernel fusion algorithm were used to simulate the system.

7.4 Architecture Comparison

The graphs in Figure 7.40 illustrate the execution time of optimised implementations of the CDS simulation, executing on single and multi-core Intel Xeon E5-2420 processor and on a NVIDIA Quadro K5000 GPU, compared to the original implementation executing on a single core of the Xeon E5-2420 processor. The following points can be inferred from the figure: (*i*) the impact of optimisations of code on CPU; (*ii*) the performance impact between single and multi-core execution; (*iii*) the difference between multi-core CPU and many-core GPU performance for the CDS application; and (*iv*) the impact of system size between CPU and GPU.

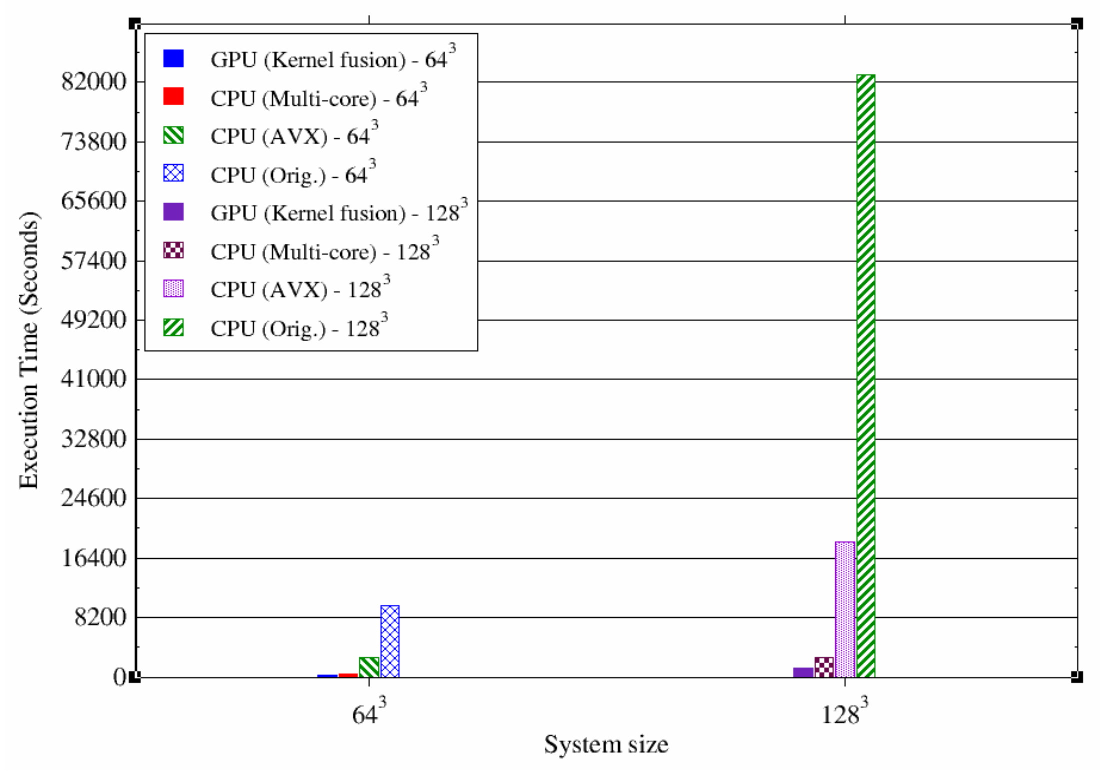


Figure 7.40: Comparisons between original, AVX optimised, CPU multi-core and GPU many-core implementations with total 100,000 time-steps.

Results are presented in single precision for two classes of the system size 64^3 and 128^3. Different system sizes for the CDS computational method were examined on CPU and GPU, and it was found that the CPU can only support a maximum 198^3 class of system size due to limited memory and hardware restrictions. GPU K5000, on the other hand, can comfortably execute 324^3 class of system size. In addition, the results show that the GPU performs better when domain size is increased. This indicates that the performance gap between the two hardware architectures increases with the problem/system size. Due to the cumulative system size the number of grid-cells in each direction will be increased and thus the volume of exploitable parallelism.

Due to the lightweight and lower-clocked cores of GPU compare to CPU, in the small class of problem size the difference between CPU and GPU performance is not significant, and even CPU can achieve better performance in a very small domain size (e.g. 16^3). However, the numbers of cores on GPU are much higher than CPU and remarkably faster (few clock cycles to execute a job); therefore, GPU can easily exploit the high amount of parallelism in big system and obtains better performance. Finally, according to the specifications of CPU and GPU (tables 6.1 and 6.2), the architecture of GPU can handle a big volume of datum size with high amount of parallelism. In addition, the introduction of different memories GPU hardware is a promising approach to the improve performance and achieve high performance computing.

7.5 Summary

The implementation of the CDS method on multi-core and many-core devices has been investigated. In the first section the optimisation of the CDS baseline code for CPUs was presented. The optimisation of the baseline code was based on the vectorisation and memory access pattern. Vectorisation was taken into account by considering data alignment based on the AVX instruction set and implicit directives. The memory access pattern was considered by SoA arrangement to reduce indirect accessing and to have a unit-stride memory access. It was shown that the optimisations are beneficial for the CDS program, improving the performance of the original and SSE4.2 implementations by up to 3.10x and 1.2x for the system size $64 \times 64 \times 64$, and 4.44x and 1.2x for $128 \times 128 \times 128$ domain size respectively. The second section demonstrated the implementation of the CDS simulation method on multi-core (multi-threaded) computer architecture. A hybrid decomposition algorithm based on the work-sharing constructs of OpenMP and regular grids data structure according to the CDS simulation method on the shared memory machine was presented. Three different scenarios were investigated to show the impact of the block size of data decomposition in the hybrid algorithm. The first two scenarios, based on the block size = 1000, and block size = 100, were not very efficient compared to the third scenario. The third (last) scenario, based on the function of the number of threads improved the speedups of the CDS scheme from 4.27 to 7.26 and 4.00 to 6.81 with the number of cores cumulative from 2 to 12 for $64 \times 64 \times 64$ and $128 \times 128 \times 128$ system sizes, respectively. In addition, it demonstrated that when the system size and the volume of workloads increased, hyper-threading was not beneficial; indeed, it decreased the speedup (Figure 7.15).

The last section considered the implementation, validation, and performances benchmark of the CDS method on GPU as a many-core accelerator. The spatial decomposition method based on the block-cell link model as a domain level algorithm of CUDA-based CDS simulation is presented. The proposed algorithm in domain level illustrates that the spatial decomposition method based on the block-cell link model is a suitable and appropriate choice to decompose tasks and to solve the computational problems correctly in a parallel environment (GPU). Different optimisations for memory management such as coalescing and usage of shared memory were taken into account to improve the performance and to solve a large stride issue. Different metrics were used to determine the occupancy, branch efficiency and memory throughput for each kernel and overall CUDA application and to specify the best CUDA execution configurations. In addition, kernel fusion as an optimisation for the logical and hardware levels of parallel implementation for the CDS simulation method on GPU was proposed. It was shown that kernel fusion by reducing the kernels overheads decreasing the number of off-chip memory access from or to DRAM and having more transactions in flight to hide latency helps to improve the speedup and consequently the performance of the program.

Therefore, GPU implementation[2] based on the kernel fusion heterogeneous algorithm is roughly ≈ 1.78 times quicker than multi-core executions on CPU for $64 \times 64 \times 64$ domain size; and approximately ≈ 2.14 times faster than multi-core processing for $128 \times 128 \times 128$ system size, respectively. Evidently, the domain and logical levels of algorithm presented in section 7.3.1 and the hybrid decomposition scheme demonstrated in section 7.2.1 are simple methods for achieving adequate and acceptable levels of performance of the CDS method on different computer architectures. In fact, these approaches are well matched to various parallel workloads, specifically in the case of GPUs.

[2] H. Soltani, D. Ly and W. Ahmed, " Accelerating Cell Dynamic Simulation for 3D Diblock copolymer Sphere Morphology using GPU ", *GPU Technology Conference*, San Jose, USA, March 2015.

CHAPTER EIGHT

8 Conclusions and Future Work

This study set out with five main objectives: to investigate the CDS simulation scheme as a computational method to model the phase separation of diblock copolymers; to comprehend the dynamic behaviour of particles in different time-steps based on the new computational technique; to investigate different optimisation technique for the CDS baseline code; to consider a parallel algorithm and programming model to solve time-consuming and massive computational processing of the CDS; and to implement a new parallel algorithm on multi-core and many-core devices. Therefore, these five objectives are located into four chief sets and their results are summarised in this chapter, which draws conclusions from the findings of this work and identifies the limitations of this study, and suggesting areas for future works.

8.1 Cell Dynamic Simulation Method

Chapter 2 explored the CDS method as a promising scheme and good example of a cellular automation to present interface dynamic in phase-separating domain. The main equations of the CDS (Ginzburg-Landau (TDGL) and Cahn-Hilliard Cook (CHC)) were considered and the impacts of external fields, such as a shear flow, were taken into account to simulate spherical morphology of diblock copolymer and to comprehend the nontrivial behaviour of the spherical morphology of diblock copolymer. Hence, the following salient points present the summaries of chapter 2 results:

- The main advantage of CDS technique compared to other simulation methods is coarse-grained discretisation, which rendered a closer relationship between the real world and laboratory conditions. In fact, this advantage of the CDS method allows exploration of the phase-ordering and the micro-phase separation occurrences in systems which are comparable with experimental works in terms of dimension and size [18, 33]. However, the CDS calculations are expensive.

- To achieve a stable system, the simulation was executed for up to 1000,000 time-steps without a shear and then 300,000 time-steps were run after applying shear flow. It was noted that the system in 100,000 time-steps becomes stable with respect to the shear flow and consideration of attractive wall.

- Systematically different values of the shear rate γ were considered and it was found out that when the shear flow was between the varieties of 0.001 and 0.0001 the domain obtains the spherical morphology with hexagonal order (perfect system). On the other hand, at 0.001 shear rate the system was spheres but not completely ordered, and at higher shear rate 0.005 the spheres were lengthened to ellipsoids and cylinders. Additional increase of shear transformed the entire system from spheres to cylinders. Therefore, shear flow rate had a direct impact of system morphology.

8.2 Multiple Particle Detection and Tracking

In chapter 3 a novel particle tracking technique for a spherical phase diblock copolymers under shear flow was implemented. The new particle detecting method was utilised in the output of the CDS program as computational data, used to describe the morphology of diblock copolymers sphere under shear. The main rationale for developing a new method for particle tracking was explained (understanding numerical representation of the particle positions and to comprehend the mobile behaviour of particles in different time-steps for uniform computational data). The method, design, implementation and validation of a new particle tracking scheme yielded the following major points:

- The new method proposed two frameworks with a total number of five steps to achieve the computational technique of tracking particles. In the detection framework, neighbouring search technique was used for detecting particles and reconstructs the time-lapse of detected particles. In the tracking framework, the centres of mass of particles were calculated and particles were tracked based on their centres of mass and the movements in each time step.

- Two scenarios were considered for calculating centre of mass of particles: stabilised and not stabilised. In first scenario, numbers of particles were fixed and there was no difference between the shapes of particles. In second scenario, called difficult scenario, the numbers of particles were altering and particles had different types of shapes and sizes.

- Statistical study was undertaken into account to specify distinguish value (number of grid point) for differentiating the shapes of particles in the difficult scenario. Consequently, 55 grid points were chosen to distinguish between particles' shapes. Based on the chosen limit, particles with less than 55 grid points were considered to be single and those with more than 55 grid points as mixed particles. When the particles were homogeneous, the fundamental idea to find the COM of a particle was to add up the coordinates of all points in the x and y directions separately and divide the sum by the whole number of grid points.

- Frequency as a statistical method was used to specify the movements of all particles in each time. Based on the frequency results on particles' movement in x and y directions, nine grid points were chosen as the limit of movements in x, and four grid points were selected as the limit of travels in y direction.

- To plot the track of each particle, the centre of mass of detected particle in each time step was used. The coordinates of particles were plotted in an x-y coordinate system to illustrate the track of particles. In addition, the dynamic movement and behaviour of one and more particles concurrently based on the new method was illustrated.

- The proposed method was examined with different particles and satisfactory results in terms of accuracy and concurrently tracking of particles had been achieved.

8.3 Optimisations and Multi-Core Implementation of CDS

The first section of chapter 7 investigated the optimisation of CDS algorithm based on the vectorisation, SSE4.2, AVX instruction set of SIMD and memory layout. The second section of chapter 7 considered a new hybrid decomposition algorithm based on the work-sharing constructs of OpenMP and regular grids data structure for implementation of CDS simulation method on multi-core CPU devices. The following salient points illustrate the design, implementation and performance analysis yielded by the first two sections of chapter 7:

- The main difficulties in optimisation of CDS baseline code were referred to data dependencies and memory access patterns.

- AVX instruction set, implicit directives and SoA memory arrangement were taken into account for the optimisations of CDS.

- It was presented that the optimisations were beneficial for the CDS program, improving the performance of the original and SSE4.2 implementations by up to 3.10x and 1.2x for the system size $64 \times 64 \times 64$, and respectively 4.44x and 1.2x for $128 \times 128 \times 128$ domain size.

- A hybrid decomposition algorithm was proposed for implementation multi-threaded CDS scheme. In hybrid decomposition algorithm, the spatial partitioning as a first step of hybrid algorithm considered the whole system as a grid and divided the whole grid into three-dimensional sub-grids. Then data decomposition as a second step of hybrid decomposition algorithm started to play a role. In data decomposition, block based decomposition was used to partition a group of cells into different blocks, and then mapped each block to a core/thread in the shared memory machine.

- Three different scenarios ((*i*) data partitioning based on the block size = 1000; (*ii*) data partitioning based on the block size = 100; and (*iii*) data partitioning based on the function of the number of threads) were considered to show the impact of the block size of data decomposition in the hybrid algorithm. The last scenario based on the function of the number of threads obtained the most efficient results.

- The speedups based on the last scenario were improved from 4.27 to 7.26 and 4.00 to 6.81 with the number of cores cumulative from 1 to 12 for $64\times64\times64$ and $128\times128\times128$ system sizes respectively.

- It was demonstrated that when the system size and the volume of workloads are increased the hyper-threading was not very beneficial and useful.

- The optimisations and hybrid algorithm in the first two sections of chapter 7 demonstrated that the CDS application is not limited for any specific platform or computer architecture and it is possible to execute on multiple platforms.

8.4 Many-Core GPU Implementation of CDS

The last section of chapter 7 explored the implementation of efficient CDS method on many-core GPU. The spatial decomposition method based on the block-cell link model as a domain level algorithm of CUDA-based CDS simulation was presented. It was shown that the spatial decomposition method based on the block-cell link model was a suitable and appropriate choice to decompose tasks and to solve the computational problems correctly on GPU. The following facts summarise the main achievements of this section in terms of method, implementation, performance analysis and validation:

- Fine-grained spatial decomposition was used as a decomposition method in CUDA implementation of the CDS on GPU.

- In the spatial decomposition method according to the CUDA hierarchy model the uniform block-cell linked data structure was used to be a shared data structure.

- Different optimisations for memory management, such as coalescing and usage of shared memory, were considered to improve the performance and to solve a large stride issue.

- Different metrics were used to determine the occupancy, branch efficiency, and memory throughput for each kernel and overall CUDA application and to specify the best CUDA execution configurations.

- Kernel fusion as an optimisation for the logical and hardware levels of parallel implementation for the CDS simulation method on GPU was illustrated. It was presented that by reducing kernel overheads, kernel fusion decreases the number of off-chip memory access from or to DRAM, and having more transactions in flight to hide latency helps to improve the speedup and consequently performance of the program.

- In our experience, GPU - CUDA is better matched to exploiting parallelism compare to other programming languages. In addition, there is no scalability issue in CUDA programming model.

- It was found that GPU is also better suited for big datum size. GPU can easily exploit the high amount of parallelism in big system and obtains better performance.

- The speedups based on the CUDA-based CDS simulation scheme are roughly 50x, 14x and 1.78x quicker than non-optimised single core, AVX optimised single core, and multi-core executions on CPU for $64\times64\times64$ domain size; and approximately 66x, 15x and 2.14x faster than non-optimised single core, optimised single core and multi-core processing for $128\times128\times128$ system size.

- These results demonstrated that it is possible to achieve a good performance by optimising and developing an application with heavy and expensive computational works on single-node computer.

- Validation and evaluation of CUDA program was investigated with different profilers (nvprof and nsight), system parameters (such as shear flow and domain size) and execution configurations (such as different grid and thread block size, registers and shared memory).

Taking all the chapters together, the work demonstrated in this thesis details CDS method as cellular computerisation technique for the consideration of dynamic behaviour of particles, implementation and evaluation of engineering program/application performance on multi-core and many-core parallel architecture. Therefore, the entire process entails: (*i*) implementation of CDS method on C programming language; (*ii*) consideration of CDS scheme for the spherical morphology of diblock copolymer with shear flow and without shear flow; (*iii*) development and evaluation of novel computational method for particle detection and tracking; (*iv*) exploring the concept of parallel computing and parallel execution; (*v*) investigating the concept of emerging GPU parallel architecture; (*vi*) baseline code optimisation; (*vii*) many-core implementation of optimised baseline code; (*viii*) optimisation and demonstration a clear implementation path for the CDS method on GPU many-core architecture based on the CUDA C; and (*ix*) evaluation and comparison between different parallel architectures in terms of execution times and speedups. Finally, the proposed algorithms for both multi-core and many-core architectures will improve code maintainability among varies generations of hardware models and also ease the code portability and scalability across new architectures and platforms.

8.5 Limitations of the Study

A primary limitation is the investigation and concentration on only one specific scientific application. Although this may limit the generality of proposed computational algorithms and programming methods, it should be noted that the CDS method comprises intensive and complex mathematical calculations with high exploitable parallelism. Furthermore, the parallel computing model and behaviours of the CDS are deployed and mutual in other applications such as computational fluid dynamics, for instance the numerical solution of mathematical problems defined by partial differential equations (PDEs) for which there are three main classical methods, but not limited to, for the numerical solution of PDEs [187]: (*i*) the finite difference method (FDM); (*ii*) the finite volume method (FVM); and (*iii*) the finite element method (FEM). All of these methods discretise a computational problem with infinite degrees of freedom into a finite domain/system.

The numerical solution of PDEs proposed in this work is based on the FDM of Taylor series expansion, which transforms the PDEs into numerical equations that determine the derivatives of a variable as the difference between variable values at different times and nodal points of lattice [187]. In other words, the FDM is based on the discrete derivative approximation, which has some approximation errors. The advantages of the FDM are that it is easier and faster to implement while its disadvantages are that it is limited to structured and regular grids, with the possibility of approximation errors (i.e. less accuracy) [187, 188]. The fundamental methodology of the FDM contains four steps: (*i*) discretising the computational system into series of the grid points (based on the structured lattice); (*ii*) the governing equations are discretised and transformed to algebraic form; (*iii*) approximating the first and second orders derivatives; and (*iv*) iteratively solving the group of linear algebraic equations.

The FVM method on the other hand includes discretisation of the integral form of the PDEs. This refers to the discretisation of the computational system into finite control volumes (cells).

The lattice/grid determines the boundaries of the cells and the computational node fits at the center of the cell [189]. Figure 8.1 illustrates an example of FDM discretisation.

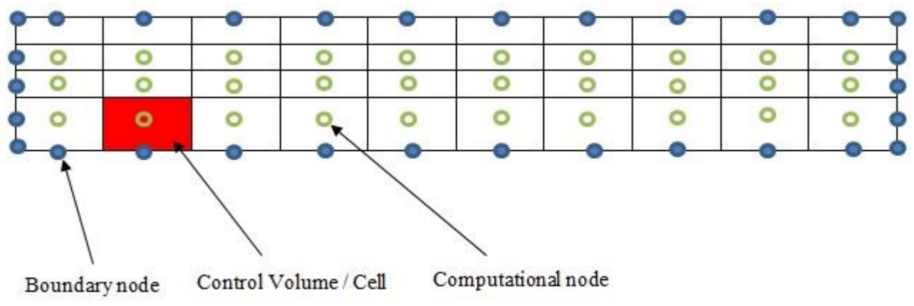


Figure 8.1: Example of FDM discretisation.

Disadvantages of this method include false diffusion when dealing with simple numeric. The main advantages of the FVM are that it is more accurate, it is not limited to structured grids and cell shapes and it achieves satisfaction of integral conservation over the cell/control volume [188, 189].

Generally, CFD refers to the usage of the numerical techniques to address and solve fluid dynamical issues (e.g. pertaining to air, water, liquid and thermal fluid). Numerical techniques with a broad range of methods can be used in CFD, such as those mentioned earlier, however due to the advantages and features of the FVM, fluid mechanics and computational fluid dynamics are traditionally using this method rather than FDM. For instance, the most well-known package of CFD, ANSYS FLUENT, is based on the FVM technique [190]. In addition to CDS based on FDM, Tang et al. [191] investigated and implemented phase separation patterns for diblock copolymers based on the FVM on spherical surfaces. They investigated phase separation on spherical geometry using FVM to solve CHC equation. In their method, FVM was used to address the CHC equation on spherical surface with icosahedral triangulation based on averaging Voronoi cells (made from triangular grid) to compute the Laplace operator [191]. According to their results, the FVM compared to the traditional FDM improves the speed and accuracy of the CDS calculations. However, to our knowledge, no work has been done before to implement the CDS based on the FVM for solving CHC equation on Cartesian surfaces. Therefore, considering the FVM as a numerical solution of PDEs for the CDS on Cartesian surfaces offers scope for future research work.

A secondary limitation is that the algorithms, optimisations and code implementations in this thesis are executed and evaluated only on two specific hardware architectures. With the advancement of new hardware architecture with higher specifications, the impact of proposed optimisations could be decreased. However, the proposed methods and algorithms as integral parts of any system design are strong enough to be compatible with a range of hardware and computer architectures to obtain good performance.

A third limitation refers to the use of the CUDA C programming model for the implementation of CDS simulation method on GPU. The host code of the CDS simulation scheme was written in C language and the device kernels were written in CUDA C. The main reasons for choosing the CUDA C programming model for developing the CDS method on GPU were: (*i*) free compiler of CUDA C; and (*ii*) compatibility between multi-core and many-core implementations which both are developed based on the C language. At the time of writing, NVIDIA supported the other programming languages such as FORTRAN, but only NVCC as a NVIDIA compiler for compiling the CUDA C application was free of cost.

Another potential limitation of this study is that the proposed optimisations and implementations in this work are mainly considered to accelerate speedup and to reduce the time execution of the CDS method by increasing arithmetic and memory throughputs (on many-core GPU). Consequently, the metrics that are used to determine the system performances in this work are the most direct metrics to measure performances. However, there are a number of additional metrics that can be considered for different purposes (e.g. power consumption) which are not considered in this work.

8.6 Future Work

Several avenues are left open for future work and research. Therefore, the main roads of future work based on the study demonstrated in this thesis have been elucidated by the following points:

- **3D implementation of novel particle detection and tracking technique**

  The proposed computational method of particle detection and tracking for a spherical phase diblock copolymer is based on the 2D results. Expanding a computational method to 3D can be more useful and beneficial for understanding the dynamical and mobility behaviours of particles in different time-steps.

- **Implementation of a new mathematical model for the CDS based on the finite volume method**

  The current mathematical model of the CDS is based on the finite different method. By exploring and considering a new mathematical model based on the finite volume method the limitations of structured/regular grids can be addressed and the accuracy of results improved.

- **Optimisations the baseline CPU code for future architectures**

  To consider the potential optimisations for proposed algorithm on future hardware architectures with wider and bigger SIMD width and studying the impacts of optimisations in performance of the CDS application.

- **A multi-CPU implementation of the CDS**

  The current multi-core implementation is based on the shared memory architecture. By implementing the CDS method on distributed memory architecture can solve the scalability issue and also execute larger domain size of the CDS without any consideration of memory limitations.

- **Optimisations on memory management of CUDA**

  As mentioned in chapters 6 and 7, CUDA programming model often requires the communication and sharing of data values between threads. This way was used in this study to communicate between threads within a warp was the utilisation of shared memory. However, NVIDIA introduced a new instruction to share data values between threads within a warp which called the “shuffle” [192]. The use of shuffle instruction has the following benefits: (*i*) the shuffle instruction is quicker than shared memory, because of less requirements of instruction (only one) but shared memory requires three instructions (read, synchronise, write); (*ii*) the shuffle instruction can utilise shared memory for other data or different purposes; (*iii*) shuffle can remove synchronisation between threads within a warp (__syncthreads()); and (*iv*) the potential occupancy limiters of shuffle are much less than shared memory.

- **Asynchronous streaming management**

  The other optimisation that can be investigated in future work refers to the asynchronous behaviour of kernels in different streams. In current GPU implementation all the GPU/device operations such as kernels and data transfers are in the default stream, which is synchronised. This means no operations will start until all other previous operations are completed. Noted that stream refers to a sequence operations that perform in order arrangement on GPU [193]. In asynchronous streams, there is more than one stream (non-default stream), thus operations can be executed concurrently. To consider asynchronous streams, the GPU must be supported concurrent copy and kernel execution; operations should be executed in different streams, efficient device resources such as registers, blocks and shared memory should be available, and the last and most important requirement of asynchronous streams is data dependency in concurrent operations.

- **Investigation of GPU – CUDA vector types**

  CUDA offers built-in vector and matrix data types such as int2, int4, float2, float4, double2. Investigation of vector types can be useful to understand their relative advantages and benefits in program performance. In addition, vector types may be beneficial for data values to be stored contiguously in memory (DRAM) or to improve memory bandwidth utilisation.

- **A multi-GPU implementation of the CDS**

  Implementing the CDS method across multiple GPUs or hybrid distributed-shared memory architecture will create an opportunity to execute very big system sizes and solve the bottlenecks of application. By considering heterogeneous hybrid architecture comprising a number of CPUs and GPUs, the issue of portability will be solved and different numbers of domain sizes can be executed without any problems. In this model, GPU+MPI should be used for accelerating and parallelising. However, memory access patterns and memory footprints should be considered on multi-GPU model.

- **Exploration of instruction level parallelism for better performance at inferior occupancy**

  Considering ILP per thread is another way to hide/decrease memory and arithmetic latencies and to improve program performance. In fact, by increasing parallelism between instructions in one thread good performance can be achieved even if the volume of occupancy is low.

Bibliography

[1] H. Suk and G. T. Yeh, "Development of Particle Tracking Algorithms for Various Types of Finite Particles in Multi-Dimensions," *Computers and Geosciences* , vol. 36, no. 2, pp. 564-568, 2010.

[2] G. Soni and B. M. Jaffar Ali, "Single Particle Tracking of Correlated Bacterial Dynamics," *Biophysical Journal* , vol. 84, pp. 2634-2637, 2003.

[3] F. Huang, E. Watson, C. Dempsey and J. Suh, "Real-Time Particle Tracking for Studying Intercellular Trafficking of Pharmaceutical Nancarriiers," *Cellular and Subcellular Nanotechnology*, vol. 91, pp. 211-223, 2013.

[4] H. Babcock, C. Chen and X. Zhuang, "Using Single-Particle Tracking to Study Nuclear Trafficking of Viral Genes," *Biophysical Journal* , vol. 87, no. 3, pp. 2749-2758, 2004.

[5] M. Vrljic, S. Nishinura, S. Brasselet, W. Moerner and H. Mconnell, "Translational Diffusion of Individual Class MHC Membrance Proteins in Cells," *Biophysical Journal*, vol. 83, pp. 2681-2692, 2002.

[6] K. Jaqaman and G. Danuser, "Computational Image Analysis of Cellular Dynamics: A case Study Based on Particle Tracking," *Spring Harbor Protocols*, vol. 4, no. 12, pp. 1-11, 2009.

[7] J. C. Crocker and D. G. Grier , "Methods of Digital Video Microscopy for Colloidal Studies," *Journal of Colloid and Interface Science* , vol. 179, pp. 298-310, 1996.

[8] S. Baek and S. Lee, "A New Two-Frame Particle Tracking Algorithm Using Match Probability," *Journal of Experiments in Fluids* , vol. 22, pp. 23-32, 1996.

[9] A. Ponti, A. Matov, M. Adams, S. Gupton and G. Danuser, "Periodic Patterns of Actin Turnover in Lamellipodia and Lamellae of Migrating Epithelial Cells Analysed by Quantitative Fluorescent Speckle Microscopy," *Journal of Biophysics* , vol. 89, pp. 3456-3468, 2005.

[10] V. Racine , M. Saches, J. Salamero, A. Trumbuil and J. Sibarita, "Visualisation and Quantification of Veside Trafficking on a Three-Dimensional Cytoskeleton Netwrok in Living Cells," *Journal of Microscopy* , vol. 225, no. 3, pp. 14-228, 2007.

[11] J. Dorn, G. Danuser and G. Yang, "Computational Processing and Analysis of Dynamic Fluorescence Image Data," *Methods in Cell Biology* , vol. 85, pp. 497-528, 2008.

[12] P. Matinsen, J. Blaschke, R. Knnemeyer and R. Jordani, "Accelerating Monte Carlo Simulations with an NVIDIA Graphics Processor," *Journal of Computer Physics Communications*, vol. 180, no. 1, pp. 1983-1989, 2011.

[13] M. Januszewski and M. Koster, "Accelerating Numerical Solution of Stochastic Differential Equations with CUDA," *Journal of Computer Physics Communications*, vol.

181, no. 1, pp. 183-189, 2010.

[14] J. D. Owens, D. Luebke, N. Govindaraju and A. E. Lefohn, “A Survey of General-Purpose Computation on Graphics Hardware,” *Eurographics, State of the Art Reports,* vol. 12, 2005.

[15] J. Nickolls and W. J. Dally, “The GPU Computing Era,” *IEEE Micro,* vol. 30, no. 2, pp. 56-69, 2010.

[16] “Computational Physics and GPU Programming,” [Online]. Available: http://quantumdynamics.wordpress.com/2012/03/28/computational-physics-gpu-programming-solving-the-time-dependent-schrodinger-equation. [Accessed July 2015].

[17] NVIDIA, “CUDA Toolkit Documentation - Developer Zone,” [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3zJMQTJ3E . [Accessed July 2015].

[18] M. Pinna, Mesoscale Modelling of Block Copolymer Systems, Berlin: VDM Verlag Dr.Muller Aktiengesellschaft & Co. KG, 2010.

[19] I. W. Hamley, J. A. Pople, J. P. A. Fairclough, N. J. Terrill, A. J. Ryan, C. Booth, G.-E. Yu, O. Diat, K. Almdal, K. Mortensen and M. J. Vigild, *J. Chem. Phys.,* vol. 108, pp. 6925-6929, 1998.

[20] A. V. Zvelindovsky and Ed., “Nanostructured Soft Matter,” Dordrecht, Springer, 2007, p. 630.

[21] S. Qi and Z. W. Wang, *Phys. Rev. E,* vol. 55, p. 1682, 1997.

[22] M. E. vigild, K. Almdal, K. Mortensen, I. W. Hamley, J. P. A. Fairclough and J. Ryan , *Macromolecules ,* vol. 31, p. 5702, 1998.

[23] N. Arora, A. Shringarpure and R. W. Vuduc, “Direct N-body Kernels for Multicore Platforms,” in *Proceedings of the International Conference on Parallel Processing, ICPP '09 - IEEE Computer Society,* Vienna, Austria, September 2009.

[24] J. Chhugani, C. Kim, H. Shukla , J. Park , P. Dubey, J. Shalf and H. D. Simon, “Billion-Particle SIMD Friendly Two-Point Correlation on LArgeScale HPC Cluster Systems,” in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* , Salt Lake City, UT, November 2012.

[25] Y. Oono and S. Puri, *Phys. Rev. A,* vol. 38, p. 434, 1998.

[26] J. Feng and E. J. Ruckenstein, *Chem. Phys.,* vol. 121, p. 1609, 2004.

[27] Y. Oono and S. Puri, *Phys. Rev. Lett.,* vol. 58, p. 836, 1987.

[28] S. Puri and Y. Oono, *Phys. Rev. A,* vol. 38, p. 1542, 1988.

[29] H. Kodama and M. Doi, *Macromolecules,* vol. 29, p. 2652, 1996.

[30] S. Komura and H. Kodama, *Phys. Rev. E,* vol. 55, p. 1722, 1997.

[31] M. S. O. Massunaga, M. Paniconi and Y. Oono, *Phys. Rev. E,* vol. 56, p. 723, 1997.

[32] J.-R. Roan and E. I. Shaknovich , *Phys. Rev. E,* vol. 55, p. 2109, 1999.

[33] S. R. Ren and I. W. Hamley, "Cell Dynamic Simulation of Microphase Separation in Block Copolymers," *Macromolecules,* vol. 34, no. 1, pp. 116-126, 2001.

[34] T. U. o. S. Mississippi, "Immiscible Polymer Blends," [Online]. Available: http://pslc.ws/macrog/iblend.htm. [Accessed July 2015].

[35] R. Fayt, P. Hadjiandreou and P. Teyssie, *J. Polym. Sci. Polym. Chem. Ed.,* vol. 23, p. 337, 1985.

[36] T. Obta and I. Aya, "Dynamic of Phase Separation in Copolymer-Homopolymer Mixtures," *Phys. Rev. E,* vol. 52, no. 5, pp. 5220-5229, 1995.

[37] M. Pinna, A. V. Zvelindovsky, X. Guo and C. L. Stokes, "Diblock Copolymer Sphere Morphology in Ultra Thin Films Under Shear," *The Royal Society of Chemistry - Soft Matter,* vol. 7, no. 1, pp. 6991-6997, 2011.

[38] A. V. Zvelindovsky and G. L. A. Sevink, *Euriphys. Lett.,* vol. 62, no. 1, pp. 370-376, 2003.

[39] I. W. Hamley, *Macromol. Theory Simul ,* vol. 9, p. 363, 2000.

[40] T. Ohta and K. Kawasaki, *Macromolecules,* vol. 19, p. 2621, 1986.

[41] M. Pinna, X. Guo and A. V. Zvelindovsky, *Polymer,* vol. 49, pp. 2797-2800, 2008.

[42] L. Leibler, *Macromolecules ,* vol. 13, p. 1602, 1980.

[43] M. Pinna, A. V. Zevlindovsky, S. Todd and G. Gold-Wood, "Cubic Phases of Block Copolymers Under Shear and Electric Fields by Cell Dynamics Smulation. I. Spherical Phase," *The Journal of Chemical Physics,* vol. 125, no. 15, pp. 1-10, 2006.

[44] T. Ohta, Y. Enomoto, J. L. Harden and M. Doi, *Macromulecules,* vol. 29, p. 2652, 1993.

[45] A. Shinozaki and Y. Oono, *Phys. Rev. E,* vol. 48, p. 2622, 1993.

[46] I. Rychkov, *Macromol. Theory Simul.,* vol. 207, no. 14, 2003.

[47] G. Arya, J. Rottler, A. Z. Panagiotopoulos, D. J. Srolovitz and P. M. Chaikin, *Langmuir,* vol. 21, pp. 11518-11527, 2005.

[48] A. Chremos, K. Margaritis and A. Z. Panagiotopoulos, *SoftMatter,* vol. 6, pp. 3588-3595, 2010.

[49] E. Meijering, O. Dzyubachyk and I. Smal, "Methods for Cell and Particle Tracking,"

Imaginh and Spectrodcopic Analysis of Living Cells, vol. 54, pp. 183-200, 2012.

[50] K. Jaqaman, D. Loerke, M. Mettlen, H. Kuwata, S. Grinstein, S. Schmid and G. Danuser, “Robust Single-Particle Tracking in Live-Cell Time-Lapse Sequences,” *Nature Methods* , vol. 5, pp. 695-702, 2008.

[51] R. D. KEANE and R. J. Adrian, “Theory of Cross-Correlation Analysis of PIV Imagess,” *Applied Scientific Research* , vol. 49, no. 3, pp. 191-212, 1992.

[52] S. Baek and S. Lee, “A NEW Two-Frame Particle Tracking Algorithm Using Match Probability,” *Experiments in Fluids,* vol. 22, no. 2, pp. 23-32, 1996.

[53] J. Crocker and D. Grier, “Particle Tracking Using IDL,” [Online]. Available: http://www.physics.emory.edu/faculty/weeks//idl/. [Accessed January 2014].

[54] D. Reid, “An Algorithm for Tracking Multiple Targets,” *IEEE Transactions on Automatic Control* , vol. 24, no. 6, pp. 843-854, 1979.

[55] R. Burkard and E. Çela, “Combinatorial Optimisation, Linear Assignment Prolems and Extensions,” *Kluwer Academic, Dordrecht,* vol. 23, pp. 75-112, 1999.

[56] A. Genovesio, T. Liedl, V. Emiliani and W. Parak, “Multiple Particle Tracking in 3-D+T Microscopy: Method and Application to the Tracking of Endocytosed Quantum Dots,” *IEEE Trans Image Process,* vol. 15, pp. 1062-1070, 2006.

[57] L. Ji and G. Danuser, “Tracking Quasi-Stationary Flow of Weak Fluorescent Singnals by Adaptive Multi-Frame Correlation,” *Journal Microscience* , vol. 220, pp. 150-167, 2005.

[58] J. Beltman , A. Maree and R. De Boer, “Analysing Immune Cell migration,” *Nature,* vol. 9, pp. 789-798, 2009.

[59] C. Bakal, J. Aach, G. Church and N. Perrimon, “Quantitative Morphological Signatures Define Local Signalling Networks Reulating Cell Morphology,” *Science* , vol. 316, pp. 1753-1756, 2007.

[60] T. Schlick, “Molecular Modeling and Simulation,” *Interdisciplinary Aplied Mathematics Series,* vol. 21, pp. 272-276, 2002.

[61] B. Carter, G. Shubeita and S. Gross, “Tracking Single Particles: Quantitative Evolution,” *Physical Biology* , vol. 28, no. 2, pp. 60-72, 2005.

[62] S. Arya, M. Mount and R. Silverman, “An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions,” *Journal of the ACM,* vol. 45, no. 6, pp. 891-908, 2000.

[63] V. W. Lee et al., “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU,” in *In Processings of the ACM/IEEE International Symposium on Computer Arcitecture, ISCA '10,* Saint-Malo, France , 2010.

[64] J. J. Dongarra, P. Luszczek and A. Petitet, “The LINPACK Benchmark: Past, Present and Future,” *Concurrency and Computation: Practice and Experience,* vol. 15, no. 9, pp.

803-820, 2003.

[65] J. D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” *IEEE Compter Society Technical Committee on Computer Architeecture (TCCA) Newsletter,* pp. 19-25, December 1995.

[66] P. M. Kogge and T. J. Dysart, “Using the TOP500 to Trace and Project Technology and Architecture Trends.,” in *In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking Storage and Analysis.,* Seattle, WA, 2011.

[67] D. Baily et al., “The NAS Parallel Benchmarks.,” Technical Report RNR-94-007, NASA Ames Research Centre., 1994.

[68] T. Spelce, “ASC Sequoia Benchmarck Codes.,” 24 June 2013. [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks/. [Accessed July 2016].

[69] S. L. Graham, P. B. Kessler and M. K. Mckusick, “Gprof: A Call Graph Execution Profiler,” in *In Proceedings of the SIGPLAN Symposium on Compiler Construction,* Boston, MA, 1982.

[70] N. Nethercote and J. Seward, “A Framwork for Heavyweight Dynamic Binary Instrumentation,” in *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, 2007.

[71] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. Bhalergo and A. Jarvis, “Parallel File System Analysis Through Application I/O Tracing,” *The Computer Journal,* 2012.

[72] O. Perks, S. D. Hammond, S. J. Pennycook and S. A. Jarvis, “WMTools - Accessing Parallel Application Memory Utilisation at Scale,” in *In Proceedings of the European Conference on Computer Performance Engineering, EPEW ’11*, Berlin, 2011.

[73] O. F. Perks, S. D. Hammond, S. J. Pennycook and S. A. Jarvis, “WMTrace - A Lightweight Memory Allocation Tracker and Analysis Framework,” in *In Proceedings of the UK Performance Engineering Workshop, UKPEW ’11*, Bradford, UK, 2011.

[74] P. J. Mucci, S. Browne, C. Deane and G. Ho, “A Portable Interface to Hardware Performance Counters,” in *In Proceedings of the Department of Defense HPCMP Users Group Conference IEEE,* Monterey, CA, 1999.

[75] NVIDIA Developer Zone, “Nsight-Eclipse-Edition-Getting-Started-Guide,” Graphics Card , [Online]. Available: http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/#axzz3gSx3oG8a. [Accessed July 2015].

[76] NVIDIA, “Nsight-Eclipse,” [Online]. Available: https://www.clear.rice.edu/comp422/resources/cuda/html/nsight-eclipse-edition-getting-started-guide/index.html. [Accessed July 2015].

[77] C. Garcia, R. Lario, M. Prieto and F. Tirrado, “Vectorisation of Multigrid Codes Using SIMD ISA Extensions,” in *In Proceedings of the International Parallel and Distributed*

Processing Symposium, IPDPS '03, IEEE Computer Society, Nice, France, 2003.

[78] J. J. Dongarra and A. R. Hinds, "Unrolling Loops in FORTRAN," *Software - Practice and Experience*, vol. 9, pp. 219-226, 1979.

[79] K. S. MacKinley, S. Carr and C. -W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems,* vol. 18, no. 4, pp. 424-453, 1996.

[80] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar and P. Dubey, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?," in *In Proceedings of the International Symposium on Computer Architecture, ISCA '12, IEEE*, Portland. OR, 2012.

[81] M. Boyer, D. Tarjan, S. T. Acton and K. Skadron, "Accelerating Leukocyte Tracking using CUDA: A Case Study in Leveraging Manycore Coprocessors," in *In Processings of the IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

[82] D. A. Jacobsen, J. C. Thibault and I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," in *In Proceedings of the AIAA Aerospace Sciences Meeting*, Orlando, FL, 2010.

[83] T. Shimokawabe and et al., "An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code," in *In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis,* New Orleans, LA, 2010.

[84] OpenMP, "OpenMP Application Program Interface," November 2012. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf. [Accessed April 2015].

[85] "Technical Report on Directives for Attached Accelerators," Technical Report TR1, OpenMP Architecture Review Board, Champaign, IL, 2012.

[86] R. Dolbeau, S. Bihan and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *In Proceedings of the Workshop on GeneralPurpose Processing on Graphics Processing Units, GPGPU '07*, Boston, MA, 2007.

[87] OpenACC, "OpenACC 1.0 Specification," OpenACC Directives for Accelerators, November 2011. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf. [Accessed August 2015].

[88] S. Lee, S. -J. Min and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimisation," in *In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '09,* Raleigh, NC, 2009.

[89] S. Lee and J. S. Vetter, "Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing," in *In Proceedings of the ACM/IEEE International*

Conference for High Performance Computing, Networking, Storage and Analysis, SC '12, Salt Lake City, UT, 2012.

[90] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall and P. H. Kelly, "Performance Analysis of the OP2 Framework on Many-core Architectures.," *SIGMETRICS Performance Evaluation Review,* vol. 38, no. 4, pp. 9-15, 2011.

[91] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina , M. Barrientons, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve and P. Hanrahan, "Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers.," in *In Proceedingd of the International Conference for High Performance Computing, Networking, Storage and Analysis* , Seattle, WA, 2011.

[92] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. A. Herdman, I. Miller and S. A. Jarvis, "Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters," *Computer Science - Research and Development,* vol. 26, no. 4, pp. 175-185, 2011.

[93] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath and T. D. Uram, "GROPHECY: GPU Performance Projection from CPU Code Skeletons," in *In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11,* Seattle, WA, 2011.

[94] G. R. Mudalige, S. D. Hammond, J. A. Smith and S. A. Jarvis, "Predictive Analysis and Optimisation of Pipelined Wavefront Computations," in *In Proceedings of theWorkshop on Advances in Parallel and Distributed Computational Models, APDCM '09,* Rome, Italy, 2009.

[95] "How to Implement Performance Metrics in CUDA C/C++," Graphics Card, [Online]. Available: https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/. [Accessed August 2015].

[96] V. S. Adve and M. K. Vernon, "Performance Analysis of Mesh Inteconnection Networks with Deterministic Routing," *IEEE Transactions on Parallel and Distributed Systems,* vol. 5, no. 3, pp. 225-246, 1994.

[97] S.-H. Chiang and M. K. Vernon, "Characteristics of a Large Shared Memory Production Workload.," in *Proceedings of the International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP '01,* Cambridge, MA, 2001.

[98] M. A. Heroux and et al., "Improving Performance via Mini-applications," Technical Report SAND2009-5574, Sandia National Laboratories, Albuquerque, NM, 2009.

[99] T. Mattson, "A "Hands-on" Introduction to OpenMP," [Online]. Available: http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf. [Accessed April 2015].

[100] M. McCool, J. Reinders and A. Robison, "Structured Prallel Programming: Patterns for Efficient Computation," *Elsevier,* vol. 22, p. 61, 2013.

[101] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on*

Electronic Computers, vol. 15, no. 5, pp. 757-763, 1966.

[102] WiseGEEK, “Vector Processor,” Conjecture Corporation, [Online]. Available: http://www.wisegeek.com/what-is-a-vector-processor.htm . [Accessed May 2015].

[103] B. Barney, “Introduction to Parallel Computing,” Lawrence Livermore National Laboratory, [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/. [Accessed March 2015].

[104] S. Spacey, W. Luk, P. H. J. Kelly and D. Kuhn, “Improving Communication Latency with the Write-Only Architecture,” *Journal of Parallel and Distributed Computing,* vol. 72, no. 12, pp. 1617-1627, 2012.

[105] L. Leslie, “How to make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs,” *IEEE Transactions on Computers* , vol. 28, no. 9, pp. 690-692, 1979.

[106] J. L. Hennessy and D. A. Patterson, “Computer architecture / a quantitative approach,” in *3rd International Thomson,* San Francisco, 2002.

[107] M. J. Quinn, Parallel Architectures, Parallel Programming in C with MPI and OpenMP, Boston: McGraw Hill, 2004.

[108] D. Ibaroudene, Motivation and History, Parallel Processing, San Antonio, TX: St. Mary's University , 2008.

[109] G. Conte, S. Tommesani and F. Zanichelli, “The long and winding road to high-performance image processing with MMX/SSE,” in *IEEE Int'l Workshop on Computer Architectures for Machine Preception,* 2000.

[110] D. E. Culler, J. P. Signh and A. Gupta, Parallel Computer Architecture - A Hardware/Software Approach, Morgan Kaufmann Publishers, 1999.

[111] B. R. Rau and J. A. Fisher, “Instruction-level Parallel Processing: History, Overview, and Perspective,” *Journal of Supercomputing,* vol. 7, no. 2, pp. 9-45, 1993.

[112] N. P. Jouppi and D. W. Wall, “Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines,” in *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-III,* Boston, MA, 1989.

[113] J. Reinders, “Intel Developer ZONE - AVX-512 Instructions,” Intel, July 2013. [Online]. Available: https://software.intel.com/en-us/blogs/2013/avx-512-instructions. [Accessed June 2015].

[114] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero and X. Martorell, “Extending OpenMP* with Vectore Constructs for Modern Multicore SIMD Architectures,” in *In Proceedings of the International Workshop on OpenMP, IWOMP ’12,* Rome, Italy, 2012.

[115] F. Darema, “SPMD model: past, present and future, Recent Advances in Parallel Virtual

Machine and Message Passing Interface," in *8th European PVM/MPI Users' Group Meeting,* Santorini/Thera, Greece, 2001.

[116] M. Pharr and W. R. Mark, "A SPMD Compiler for High-Performance CPU Programming," in *In Proceedings of Innovative Parallel Computing: Foundations & Applications of GPU, Manycore and Heterogeneous Systems, InPar '12,* San Jose, 2012.

[117] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue,* vol. 6, no. 2, pp. 40-53, 2008.

[118] D. Geer, "Chip Makers Turn to Multicore Processors," *Computer,* vol. 38, no. 5, pp. 11-13, 2005.

[119] H. El-Rewini and M. Abd-El-Barr, Advanced Computer Architecture and Parallel Processing, Wiley-Interscience, 2005.

[120] N. Manchanda and K. Anand, "Non-Uniform Memory Access (NUMA)," May 2010. [Online]. Available: http://cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf. [Accessed May 2015].

[121] D. Patterson and J. L. Hennessy, Computer Architecture: A Quantitative Approach (4th ed.), Burlington, Massachusetts: Morgan Kaufmann, 2006.

[122] J. D. Owens, M. Huston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, 2008.

[123] M. Harris, W. Baxter, T. Scheureman and A. Lastra, "Simulation and Computation: Simulation of CLoud Dynamics on Graphics Hardware," *ACM SIGRAPH/EURPGRAPHICS Workshop on Graphics Hardware,* vol. 92, 2003.

[124] NVIDIA, "NVIDIA's CUDA Documentation," [Online]. Available: http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf. [Accessed June 2015].

[125] J. Dongarra and P. Beckman, "International Exascale Software Project," in *In Proceedings of the 2010 International Supercomputing Conference,* Hamburg, 2010.

[126] W. Thomas and R. Daruwala, "Performance Comparison of CPU and GPU on a Discrete Heterogeneous Architecture," in *International Conference on Circuits Systems, Communication and Information Technology Application (CSCITA),* 2014.

[127] M. Pharr and R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005.

[128] N. Corporation, "The GeForce 6 Series GPU Architecture," [Online]. Available: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter30.html . [Accessed August 2015].

[129] D. Luebke, G. Humphreys and N. Res, "How GPUs Work," *Computer,* vol. 40, no. 2, pp. 96-100, 2007.

[130] K. Group, “OpenCL,” 2010. [Online]. Available: http://www.khronos.org/opencl. [Accessed September 2015].

[131] NVIDIA, “CUDA Programming Guide Version 6.0,” 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3zhavvdyM . [Accessed February 2015].

[132] “NVIDIA Unveils Next Generation CUDA GPU Architecture - Codenamed Fermi,” October 2009. [Online]. Available: http://www.cdrinfo.com/Sections/News/Details.aspx?NewsId=26054 . [Accessed August 2015].

[133] W. Thomas and R. D. Daruwala, “Performance Comparison of CPU and GPU on a Discrete Heterogeneous Architecture,” in *In IEEE - International Conference on Circuits, Systems, Communications and Information Technology Applications*, 2014.

[134] M. E. Garland, “Parallel Computing Experiences with CUDA,” *IEEE Micro*, vol. 28, no. 4, pp. 13-27, 2008.

[135] NVIDIA, “Kepler Architecture,” [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html . [Accessed August 2015].

[136] NVIDIA, “Tuning CUDA Application for Kepler,” 2015. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#axzz3zhavvdyM. [Accessed September 2015].

[137] NVIDIA, “Kepler Tuning Guide,” Graphics Card, 2015. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/#axzz3kyVd6kvW. [Accessed September 2015].

[138] J. McKennon, “GPU Memory Types - Performance Comparison,” August 2013. [Online]. Available: https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/. [Accessed September 2015].

[139] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors, Second Edition: A Hands-on Approach, Morgan Kaufmann Elsevier , 2013.

[140] J. Liang, K. Li, L. Shi and Y. Liao, “Accelerating Dynamics Simulation of Solidification Processess of Liquid Metals using GPU with CUDA,” in *In IEEE 27th International Symposium on Parallel & Distibuted Processing*, 2013.

[141] N. Gupta, “CUDA Programming,” [Online]. Available: http://cuda-programming.blogspot.in/2013/01/thread-and-block-heuristics-in-cuda.html. [Accessed October 2015].

[142] R. Amorim, G. Haase, M. Liebmann and R. Weber dos Santos, “Comparing CUDA and OpenGL implementations for a Jacobi Iteration,” in *In 2009 International Conference on High Performance Computing & Simulation, Technical Report SFB-Report, IEEE*, 2009.

[143] J. Cohen and M. Molemaker, “A Fast Double Precision CFD Code Using CUDA,” *In*

Parallel Computational Fluid Dynamics, vol. 66, pp. 2-17, 2009.

[144] C. E. Figaard, “Introduction G2X: Porting GADGET2 to CUDA Carsten Eia Frigaard to N-body Solvers,” [Online]. Available: http://www.astro.lu.se/compugpu2010/resources/Frigaard.pdf. [Accessed October 2015].

[145] V. Simek, R. Dvorak, F. Zboril and J. Kunovsky, “Towards Accelerated Computation of Atmospheric Equations Using CUDA,” in *In Proceedings of the UKSim 2009: 11th International Conference on Computer Modelling and Simulation,* Washington, DC, USA, 2009.

[146] N. Bell and M. Garland, “Efficient Sparse Matrix-Vector Multiplication on CUDA,” NVIDIA Technical Report, 2008.

[147] D. M. Trombly, V. Pryamitsyn and V. Ganesan , *Journal of Chemical Physics* , vol. 134, no. 15, 2011.

[148] F. Drolet and G. H.Fredrickson, “Combinatorial Screening of Complex Block Copolymer Assembly with Self-Consistent Field Theory,” *Physical Review,* vol. 83, no. 21, 1999.

[149] F. A. Detcheverry, D. Q. Pike, U. Nagpal, P. E. Nealey and J. J. De Pablo, *Physical Review Letters,* vol. 102, no. 14, 2009.

[150] U. Nagpal, M. Muller, P. F. Nealey and J. J. de Pablo, “Directed Self-assembly of Block Co-polymers for Nano-manufacturing,” *ACS Macro Letters,* 2012.

[151] Intel Corporation, “A Guide to Vectorization with Intel C++ Compilers,” 2012. [Online]. Available: http://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf. [Accessed October 2015].

[152] C. Lomont, “Introduction to Intel Advanced Vector Extensions,” June 2011. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions. [Accessed October 2015].

[153] C. Ozdogan, G. Dereli and T. Gagin, *Comput. Phys. Commun,* vol. 148, p. 188, 2002.

[154] D. B. Wang, F. B. Hsiao, C. H. Chuang and Y. C. Lee, “Algorithm Optimisation in Molecular Dynamics Simulation,” *Comput. Phys. Commun,* vol. 177, no. 7, pp. 551-559, 2007.

[155] Z. Yao, J. S. Wang, G. R. Liu and M. Cheng, “Improved Neighbor List Algorithm in Molecular Simulations Using Cell Decomposition and Data Sorting Method,” *Comput. Phys. Commun,* vol. 161, no. 2, pp. 27-35, 2004.

[156] R. Hayashi and S. Horiguchi, “A Parallel Molecular Dynamics Simulation by Dynamic Load Balancing Based on Permanent Cells,” *Transactions of Information Processing Society of Japan,* vol. 40, no. 5, pp. 2152-2162, 1999.

[157] L. Nyland, J. Prins and R. H. Yun, “Achieving Scalable Parallel Moleculaar Dynamics

Using Dynamic Spatial Domain Decomposition Techniques," *Journal of Parallel and Distributed Computing,* vol. 47, no. 2, pp. 125-138, 1997.

[158] Stanford Computer Graphics Laboratory, "The Stanford 3D Scanning Repository," 2012. [Online]. Available: http://graphics.stanford.. [Accessed October 2015].

[159] S. Guntury and P. J. Narayanan, "Raytracing Dynamic Scenes on the GPU Using Grids," *IEEE Transactions on Visualisation and Computer Graphics,* vol. 18, no. 1, pp. 5-16, 2012.

[160] J. Kalojanov, M. Billeter and P. Slusallek, "Two-level Grids for Ray Tracing on GPUs," *Computer Graphics Forum,* vol. 30, no. 2, pp. 307-314, 2011.

[161] D. M. Beazley and P. S. Lomdahal, "Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5," *Parall. Comp.,* vol. 20, no. 2, pp. 173-195, 1994.

[162] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon and A. White, Source-book of Parallel Computing, San Francisco, 2003.

[163] J. Benzi and M. Damodaran, "Parallel Three Dimensional Direct Simulation Monte Carlo for Simulating Micro Flows," *Parallel Computational Fluid Dynamics: Implementations and Experiences on Large Scale and Grid Computing,* p. 95, 2007.

[164] P. Agrawal, A. D. Agrawal, M. L. Bushnell and J. Sienicki , "Superlinear Speedup in Multiprocessing Environment," in *First International Workship on Parallel Processing,* Bangalore, 1994.

[165] Intel, "Intel VTune Amplifire," [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe. [Accessed January 2015].

[166] D. T. Marr and et al., "Hyper-Threading Technology Architecture and Micro architecture," *Intel Technology Journal,* vol. 6, no. 1, 2002.

[167] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi and R. Rooholamini, "An Empirical Study of Hyper-Threading in High Performance Computing Clusters," [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.465.4841&rep=rep1&type=pdf . [Accessed November 2015].

[168] S. Casey, "How to Determine the Effectiveness of Hyper-Threading Technology with an Application," Intel, April 2011. [Online]. Available: https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application/. [Accessed November 2015].

[169] A. Valles, "Performance Insights to Intel Hyper-Threading Technology," Intel, November 2009. [Online]. Available: https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology. [Accessed November 2015].

[170] A. Grama, A. Gupta, G. Karypis and V. Kumar, "Principles of Parallel Algorithm Design - Decomposition Techniques," in *Introduction to Parallel Computing,* Addison-Wesley.

[171] S. Scott, “A System Prespective on Exascale,” in *In Proceedings of the 2010 International Supercomputing Conference*, Hamburg, Germany, 2010.

[172] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan and S. K. Weeratunga, “The NAS Parallel Benchmarks,” *International Journal of High Performance Computing Applications,* vol. 5, no. 3, pp. 63-73, 1991.

[173] J. A. Anderson, C. D. Lorenz and A. Travesset, “General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units,” *Journal of Computational Physics,* vol. 227, no. 10, pp. 5342-5359, 2008.

[174] J. T¨olke, “Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA,” *Computing and Visualization in Science,* vol. 13, no. 1, pp. 29-39, 2008.

[175] W. Li, X. Wei and A. Kaufman, “Implementing Lattice Boltzmann Computation on Graphics Hardware,” *The Visual Computer,* vol. 19, no. 7-8, pp. 444-456, 2003.

[176] L. Lamport, “The Parallel Execution of DO Loops.,” *Communications of the ACM,* vol. 17, no. 3, pp. 83-93, 1974.

[177] C. Obrecht, F. Kuznik, B. Tourancheau and J. Roux, “A New Approach to the Lattice Boltzmann Method for Graphics Processing Units.,” *Computers and Mathematics with Applications.,* vol. 61, no. 12, pp. 3628-3638, 2010.

[178] D. Bacon, S. Graham and O. Sharp, “Compiler Transformations for High-Performance Computing,” *ACM Computing Surveys (CSUR),* vol. 26, no. 4, pp. 345-420, 1994.

[179] NVIDIA, “NVIDIA - Quadro K5000, GPU leverages the new NVIDIA Kepler architecture,” [Online]. Available: http://www.pny.eu/data/products/brochures/nvidia%20quadro%20k5000%20by%20pny%20datasheet.pdf. [Accessed December 2014].

[180] C. NVIDIA, “GPU Occupancy Calculator,” CUDA SDK, 2012. [Online].

[181] V. Volkov, “Better Performace at Lower Occupancy,” 2011. [Online]. Available: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf. [Accessed November 2015].

[182] J. Cheng, M. Grossman and T. McKercher, Professional CUDA C Programming, John Wiley & Sons, 2014.

[183] J. Fousek, J. Filivpovic and M. Madzin, “Automatic Fusions of CUDA-GPU Kernels for Parallel Map,” in *In Second International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART),* 2011.

[184] S. Sato and H. Iwasaki, “A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming,” *In Programming Languages and Systems,* vol. 59, 2009.

[185] H. Wu, G. Diamos, A. Lele, J. Wang, S. Cadambi, S. Yalamanchili and S. Chakradhar,

“Optimizing Data Warehousing Applicaations for GPUs Using Kernel Fusion/Fission,” in *In Proceedings of the Multicore and GPU Programming Models, Languages and Compilers Workshop,* Shanhai, China, 2012.

[186] G. Wang, Y. Lin and W. Yi, “Kernel Fusion: An Effective Method for better Power Efficiency on Multithreaded GPU,” in *In IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM),* 2010.

[187] S. Yip (ed.), “FINITE DIFFERENCE, FINITE ELEMENT AND FINITE VOLUME METHODS FOR PARTIAL DIFFERENTIAL EQUATIONS,” in *Handbook of Materials Modeling.,* Netherlands, Springer, 2005, pp. 1-32.

[188] A. Bakker, “Applied Computational Fluid Dynamics,” 2006. [Online]. Available: http://www.bakker.org/dartmouth06/engs150/05-solv.pdf. [Accessed September 2016].

[189] J. F. Wendt (ed.), “Introduction to Finite Volume Methods in Computational Fluid Dynamics,” in *Computational Fluid Dynamics* , Berlin, Springer, 2009, pp. 275-278.

[190] “CFD Online,” CFD, [Online]. Available: http://www.cfd-online.com/Forums/main/287-finite-difference-vs-finite-volume.html. [Accessed September 2016].

[191] P. Tang, F. Qiu, H. Zhang and Y. Yang, “Phase Separation Patterns for Diblock Copolymers on Spherical Surfaces: A Finite Volume Method,” *Physical Review,* vol. 72, no. 4, pp. 16710-16717, 2005.

[192] M. Hariss, “CUDA Pro Tip: Do the Kepler Shuffle,” NVIDIA - Graphics Card, February 2014. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/.

[193] S. Rennich, “CUDA C/C++ Streams and Concurrency,” 2011. [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf. [Accessed January 2016].

Appendix A

The source-code of the implementation of the cell dynamics simulation method in C

Some parts of the CDS source-code

```
//******************************************************************
// The following part is for calculating the
// first isotropised discrete Laplacian.
// apxi1 = ⟨⟨Ψ(t,n)⟩⟩ − ψ(t,n) within
// Γ(t,n) = f(ψ(t,n)) − ψ(t,n) + D[⟨⟨Ψ(t,n)⟩⟩ − Ψ(t,n)]
// with consideration of the boundary conditions.
//******************************************************************
            for (k=1;k<=nz;++k) {
             for (j=1;j<=ny;++j) {
              for (i=1;i<=nx;++i) {
               pxi0[i][j][k]=pxi[i][j][k];
                aapxi1[i][j][k]=c1*(pxi[upx[i]][j][k] +
                 pxi[downx[i]][j][k]
                 + pxi[i][upy[j]][k] + pxi[i][downy[j]][k] +
                   pxi[i][j][upz[k]]
                 + pxi[i][j][downz[k]]);
//*****************************************************
               bapxi1[i][j][k]= c2 * (pxi[downx[i]][upy[j]][k] +
                pxi[downx[i]][downy[j]][k]
                + pxi[upx[i]][upy[j]][k] + pxi[upx[i]][downy[j]][k]
                + pxi[i][downy[j]][upz[k]]
                + pxi[i][downy[j]][downz[k]]
                + pxi[i][upy[j]][upz[k]] + pxi[i][upy[j]][downz[k]]
                + pxi[downx[i]][j][upz[k]]
                + pxi[downx[i]][j][downz[k]]
                + pxi[upx[i]][j][upz[k]]
                + pxi[upx[i]][j][downz[k]]);
//*****************************************************
              capxi1[i][j][k]= c3 *
              (pxi[downx[i]][downy[j]][upz[k]]
              + pxi[downx[i]][upy[j]][upz[k]]
              + pxi[downx[i]][downy[j]][downz[k]]
              + pxi[downx[i]][upy[j]][downz[k]]
              + pxi[upx[i]][downy[j]][upz[k]]
              + pxi[upx[i]][upy[j]][upz[k]]
              + pxi[upx[i]][downy[j]][downz[k]]
              + pxi[upx[i]][upy[j]][downz[k]]);
//*****************************************************
```

```
            apxi1[i][j][k]=aapxi1[i][j][k]
             + bapxi1[i][j][k]
             + capxi1[i][j][k];
             }
            }
           }

//*************************************************************
// This part is to calculate the Free energy functional:
//  $\Gamma(t,n) = f(\psi(t,n)) - \psi(t,n) + D[\langle\langle\Psi(t,n)\rangle\rangle - \Psi(t,n)]$
// map1= $\Gamma(t,n)$
//*************************************************************
           for (k=1;k<=nz;++k) {
            for (j=1;j<=ny;++j) {
             for (i=1;i<=nx;++i) {
              map1[i][j][k]= f[i][j][k] +d * (apxi1[i][j][k]
              - pxi[i][j][k])
              - hx[i] * ((pxi[i][j][k] + r * 2.0) / 2.0)
              - hy[j] * ((pxi[i][j][k] + r * 2.0) / 2.0)
              - hz[k] * ((pxi[i][j][k] + r * 2.0) / 2.0);
             }
            }
           }

//*************************************************************
// The following part is for calculating the
// second isotropised discrete Laplacian of the
// free energy functional, apxi2= $\langle\langle\Gamma(t,n)\rangle\rangle - \Gamma(t,n)$ within
// $\psi(t+1,n) = \psi(t,n) - \{\langle\langle\Gamma(t,n)\rangle\rangle - \Gamma(t,n) + b\psi(t,n)\}$,
// with consideration of the boundary conditions
//*************************************************************
           for (k=1;k<=nz;++k) {
            for (j=1;j<=ny;++j) {
             for (i=1;i<=nx;++i) {
              aapxi2[i][j][k]=c1 * (map1[upx[i]][j][k] +
              + map1[downx[i]][j][k]
              + map1[i][upy[j]][k] + map1[i][downy[j]][k]
              + map1[i][j][upz[k]] + map1[i][j][downz[k]]);
//**************************************************
              bapxi2[i][j][k]=c2 * (map1[downx[i]][upy[j]][k]
              + map1[downx[i]][downy[j]][k]
              + map1[upx[i]][upy[j]][k]
              + map1[upx[i]][downy[j]][k]
              + map1[i][downy[j]][upz[k]]
              + map1[i][downy[j]][downz[k]]
              + map1[i][upy[j]][upz[k]]
              + map1[i][upy[j]][downz[k]]
              + map1[downx[i]][j][upz[k]]
              + map1[downx[i]][j][downz[k]]
              + map1[upx[i]][j][upz[k]]
              + map1[upx[i]][j][downz[k]]);
//*************************************************
              capxi2[i][j][k]=c3
              * (map1[downx[i]][downy[j]][upz[k]]
              + map1[downx[i]][upy[j]][upz[k]]
              + map1[downx[i]][downy[j]][downz[k]]
              + map1[downx[i]][upy[j]][downz[k]]
              + map1[upx[i]][downy[j]][upz[k]]
              + map1[upx[i]][upy[j]][upz[k]]
              + map1[upx[i]][downy[j]][downz[k]]
              + map1[upx[i]][upy[j]][downz[k]]);
//*************************************************
              apxi2[i][j][k] = aapxi2[i][j][k]
              + bapxi2[i][j][k]
              + capxi2[i][j][k];
             }
            }
           }

//************************************************************
// The following part is for the time evolution of
// the order parameters calculations (Pxi(t+1,n)) with
// consideration of shear and long-range interaction:
```

$$\psi(t+1,n)=\psi(t,n)-\{\langle\langle\Gamma(t,n)\rangle\rangle-\Gamma(t,n)+b\,\psi(t,n)+$$

// $$\frac{1}{2}\gamma y\left[\psi(n_x+1,n_y,n_z,t)-\psi(n_x-1,n_y,n_z,t)\right]\}$$

```
//************************************************
         for (k=1;k<=nz;++k) {
          for (j=1;j<=ny;++j) {
           for (i=1;i<=nx;++i) {
            pxi[i][j][k] = pxi0[i][j][k]+ deltat *
            (-0.5 * sh* (float) j * (pxi0[upx[i]][j][k] -
            pxi0[downx[i]][j][k]) + e * (pxi0[i][upy[j]][k]
            + pxi0[i][downy[j]][k] - 2.0 * pxi0[i][j][k]) -b
            * pxi[i][j][k] + map1[i][j][k] - apxi2[i][j][k]);
           }
          }
         }
```

Appendix B

The Whole Pseudo-Code of Particle Tracking Computational Algorithm

Part I - Particle detection

```
Repeat – outer loop for all time steps
 Begin Inner loop-level 1
  Reading Pxi values
End Inner loop-level 1
Step A – applying the detection method and PBC
  Begin Inner loop-level 2
  Find the first particle if (Pxi (i, j, k) = 1)
   Modify the boundary conditions
    Start to search the nearest neighbours
    If neighbours are equal 1 then
    Change the current coordinates to the new coordinates and modify the Pxi values
Step B – counting the number of particles
    Count the number of particles
    Specify the number of particles
Step C - writing the results
    Final checking for detecting particles
     Writing the results into files
   End Inner loop – Level 2
 End Repeat – outer loop
```

Part II - Calculating centre of mass

```
 Reading detection output files
  Repeat   for all time steps
     Read Pxi values
     Repeat for all particle numbers (C)
     Counting the number of grid points belong to the particle
  End repeat
 Repeat for all particle numbers
    Initialise A to 0
    Initialise B to 0
Step A – initialising the values
  If the particle has grid points in left width boundary then
    Set A=20
   End if
 If the particle has grid points in right width boundary then
    Set B=20
 End if
 If the particle has grid points in up length boundary then
```

```
    Set M=20
 End if
 If the particle has grid points in down length boundary then
    Set N=20
 End if
Step B – finding the single particles
 If it is a single particle (if the number of grid points is less than 55)then
   If A=20 and B=20 (If the particle has gird points in both width boundaries) then
    If the grid points are close to the right width boundary then
     Set comx = comx + grid point coordinate (i)
  Else if the grid points are close to the left width boundary then
     Set comx = comx + grid point coordinate (i) -127
 End if
 If M=20 and N=20 (If the particle has gird points in both length boundaries) then
   If the grid points are close to the up length boundary then
     Set comy = comy + grid point coordinate (j)
  Else if the grid points are close to the down length boundary then
     Set comy = comy + grid point coordinate (j) -127
 End if
 If A=20 and B=20 and M=20 and N=20 (If particle has grid points in all boundaries (it is in
the corners)) then
 If the grid points are close to the right width boundary then
     Set comx = comx + grid point coordinate (i)
  Else if the grid points are close to the left width boundary then
     Set comx = comx + grid point coordinate (i) -127
  Else if the grid points are close to the up length boundary then
     Set comy = comy + grid point coordinate (j)
  Else if the grid points are close to the down length boundary then
     Set comy = comy + grid point coordinate (j) -127
 End if
   Else if the particle is not in the boundaries then
     Set comx = comx + grid point coordinate (i)
     Set comy = comy + grid point coordinate (j)
 End if
Step C – finding the mixed particles
 If it is a mixed particle (if the number of grid points is more than 55) then
   If A=20 and B=20 (If the particle has gird points in both width boundaries) then
     Find the width and length of the mixed particle considering width boundary condition.
   If M=20 and N=20 (If the particle has gird points in both length boundaries) then
     Find the width and length of the mixed particle considering length boundary condition.
 End if
   If A=20 and B=20 and M=20 and N=20 (If particle has grid points in all boundaries (it is in
the corners)) then
     Find the width and length of the mixed particle considering both length and width boundary
condition.
 End if
Step D – calculating and writing the centre of mass
    If the particle is single then
    Calculate and write the centre of mass in to the files
```

```
End if
    If the particle is mixed then
   Calculate two centre of mass considering its direction (horizontal, oblique (ascending,
descending)) and  write the outputs into the files
   End if
End Repeat

                    Part III - Tracking the next position

Reading the centre of masses output files
  Initialise detected particle (The specific particle which we want to detect- ji)
 Repeat for all time steps
    Repeat for all
     Read centre of mass of detected particles in first time -step
 End repeat
  Repeat for all
   Read centre of mass of detected particles in next time step
  End repeat
 Read Pxi values
  Repeat for all particle numbers (C)
   If the particle number in first time step is equal to the variable ji then
Step A – Find the nearest particle and name the new particle as an initial particle
  Finding the nearest particle base on the finding the nearest centre of mass
Step B - Considering periodic boundary condition (PBC)
Periodic boundary conditions apply in five different situations.
   1- PBC in width boundary
   2 - PBC in length boundary down -middle
   3 - PBC in length boundary up -middle
   4 - PBC in length and width boundary up
   5 - PBC in length and width boundary down
  End repeat
 Write the outputs in the files.
End repeat
```