# Locality Data Properties of 3D Data Orderings with Application to Parallel Molecular Dynamics Simulations

A thesis submitted in partial fulfillment
of the requirement for the degree of Doctor of Philosophy

**Ibrahim Al Kharusi**

**2019**

**Cardiff University**
**School of Computer Science & Informatics**

## Declaration

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed ................................................... (candidate)

Date ..........................................

## Statement 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of PhD.

Signed ................................................... (candidate)

Date ..........................................

## Statement 2

This work has not been submitted in substance for any other degree or award at this or any other university or place of learning, nor is it being submitted concurrently for any other degree or award outside of any formal collaboration agreement between the University and a partner organisation).

Signed ................................................... (candidate)

Date ..........................................

## Statement 3

I hereby give consent for my thesis, if accepted, to be available in the University's Open Access repository (or, where approved, to be available in the University library and for inter-library loan), and for the title and summary to be made available to outside organisations, subject to the expiry of a University-approved bar on access if applicable.

Signed ................................................... (candidate)

Date ..........................................

## Word Count ........................................

(Excluding summary, acknowledgements, declarations, contents pages, appendices, tables,diagrams and figures, references, bibliography, footnotes and endnotes)

**To my family**
**for their patience and support.**

# Abstract

General-purpose computing on GPUs is widely adopted for scientific applications, providing inexpensive platforms for massively parallel computation. This has motivated us to investigate GPU performance in terms of speed and memory usage, specifically in relation to data locality in molecular dynamics simulations. The assumption is that enhancing data locality of these applications will lower the cost of data movement across the GPU memory hierarchy. In this research, we analyse spatial data locality and data reuse (temporal data locality) characteristics for row-major, Hilbert, and Morton data orderings, and hybrid variants of these, and assess their impact on the performance of molecular dynamics simulations (MDS). Data locality in MDS applications, based on the relationship between a bin and its neighbouring bins, that are generated using an approximately spherical stencil, previously has not been widely studied. In this research, a simple cache model is presented, and this is found to yield results that are consistent with timing results for the particle force computation obtained on NVIDIA Geforce GTX960 and Tesla P100 graphical processing units (GPUs). The NVIDIA profiling tool is used to investigate the execution time results and to observe the memory usage in terms of cache hits and the number of memory transactions. The analysis also provides a more detailed explanation of execution behaviour for the different orderings. To the best of our knowledge, this is the first study to investigate memory analysis and data locality issues for molecular dynamics simulations of Lennard-Jones fluids on NVIDIA's Maxwell and Tesla architectures.

# Acknowledgements

This thesis would not have been possible without the inspiration and support of a number of wonderful individuals — my thanks and appreciation to all of them for being part of this journey and making this thesis possible. I owe my deepest gratitude to my supervisor, Professor David Walker. Without his enthusiasm, encouragement, support and continuous optimism, this thesis would have been difficult to complete. Also, I express my warmest gratitude to Dr George Theodorakopoulos, Dr Christine Mumford, Dr Xianfang Sun for all their input and support.

Thanks to my parents and my wife for their support, encouragement and patience, and also to my lovely children, AlKhalil, Issra, Alaa and Ahmed, for being strong and patient when I was away from them. Thanks to my brothers and sisters, and all my family members and friends, who coped with my absence during my PhD journey. Hopefully, they will all be proud of me.

To me, there is nothing better than having good brothers and friends, so I feel a great sense of pride while offering them my deepest gratitude: Yahya, Saif, Hamed, Shathan, Abdullah, Ahmed, Adil, Yousef, Abdualrahman, Khalid, Mohammed, Sulieman, Salim Al-Mutani, Mazin Al-Shidhani, Dr Taimur Al Said, Dr Sahar, Dr Aseela Al-Harthi, Shahd Alahdal.

I must also thank my fellow research students in the School, and there are truly too many of you to name, without risking missing someone out.

Thanks to Oman, for educating us and imbuing within us the sense that ambition has no limit, and that through hard work we will achieve our dreams. We will continue to work tirelessly for you.

Thanks to the Information Technology Authority (Oman), for their trust and confidence whilst completing my studies. Thanks to Dr Salim Al Ruzaiqi (CEO of ITA), Omar Al Shanfari (Deputy CEO of ITA for Operations), the NDCC team, and the HR team for their continuous cooperation, support and motivation.

I would like also to thank the anonymous reviewers, whose comments helped us to improve our research work and enhance it before publication.

Most of all, foremost praise is to Allah, the Almighty, the greatest of all, on whom ultimately we depend for sustenance and guidance. I would like to thank Almighty Allah for guiding me through this journey and the continuous grace, mercy, and blessings have been with me throughout my life and even more so during my studies.

# Contents

# List of Publications

The work described in this thesis is partially based on the following publication [6].

- I. Al-Kharusi and D. W. Walker (2019). Locality properties of 3D data orderings with application to parallel molecular dynamics simulations. *The International Journal of High Performance Computing Applications*. `https://doi.org/10.1177/1094342019846282`.

# List of Figures

# List of Tables

# List of Algorithms

# List of Acronyms

**ASIC** Application-Specific Integrated Circuit

**AVL** Georgy Adelson-Velsky and Evgenii Landis

**BLAS** Basic Linear Algebra Subprograms

**CUDA** Compute Unified Device Architecture

**DEM** Discrete Element Method

**DMA** Direct Memory Access

**DNA** Deoxyribonucleic Acid

**DRAM** Dynamic Random Access Memory

**EAM** Embedded-Atom Method

**FAST** Fast Architecture Sensitive Tree

**FFT** Fast Fourier Transform

**FMM** Fast Multipole Method

**GPU** Graphical Processing Unit

**HCP** Hierarchical Charge Partitioning

**HPC** High Performance Computing

| | |
|---|---|
| **LJ** | Lennard Jones |
| **LRU** | Least Recent Used |
| **MDS** | Molecular Dynamics Simulation |
| **MKSA** | Metre, Kilogram Second, Ampere |
| **MPI** | Message Passing Interface |
| **MSM** | Multilevel Summation Method |
| **OpenCL** | Open Computing Language |
| **OpenMP** | Open Multi-Processing |
| **PBC** | Periodicity Boundary Condition |
| **PME** | Particle Mesh Ewald |
| **SIMD** | Single Instruction Multiple Data |
| **SIMT** | Single Instruction Multiple Threads |
| **SFC** | Space Filling Curve |
| **TLB** | Translation Lookaside Buffer |

*Chapter 1*

# Introduction

The exploitation of data locality and performance engineering are important issues in compiler and algorithm design where the aim is to fully exploit device architectures, especially in multicore/manycore systems, within the context of complex memory hierarchies governed by various rules and policies. This thesis investigates the data locality properties of our graphical processing unit (GPU) implementation of a molecular dynamics simulation. Different orderings of the particle data are investigated and compared in terms of observed performance and profiling data, and these results are interpreted in terms of the software and hardware features of the GPUs studied. To the best of our knowledge, this study has not been conducted before. In this study we approach the problem of data locality of molecular dynamics simulations on GPUs using execution time and a number of profiling measurements, especially those related to GPU execution efficiency and memory access.

## 1.1 Overview

Alongside the advances in computer architectures, processor speeds, and main memory, the efficient use of various levels of cache and other specialized memories remains essential to achieving high computation efficiency. Therefore, for scientific applications where execution time is critical, understanding the data locality properties of the application, and the execution platform, can enhance application performance through the optimal use of the execution platform. This is particularly true in GPUs where latency tolerance techniques based on the scheduling of threads are used to mask the disparity between the bandwidth to global memory and the GPU's peak execution speed. For example, for an NVidia P100 system, the global memory bandwidth is a maximum of 732 GB/s and the peak single-precision [4] performance is 9.3 Tflop/s. Thus, in the absence of latency tolerance, the expected execution speed is $732G/4$ Gflop/s, where $G$ is the number of floating-point operations per global memory access and floats are assumed to be 4 bytes. For $G = 1$, this is a factor of 50 less than the peak performance.

Data locality[112, p. 374] is a significant factor in the efficient use of hierarchical memory. When one item is moved from a lower level of memory to a higher level, other items that are nearby in memory are also moved along with it as the data are copied in blocks of fixed size, known as cache lines, rather than as individual elements. Most applications and computations are local in nature, so if items are stored in memory based on their location, when one item is moved from low-level memory into a higher level of memory, the other items upon which its processing depends will also be moved, and consecutively loaded instructions will process the consecutive data, thereby exploiting *spatial data locality* and improving performance. Likewise, data that is already in the higher level memory can be processed by the same or different instructions. This situation illustrates *temporal data locality*, where better performance is achieved by repeatedly accessing data while it is held in the higher levels of the memory hierarchy.

Space filling curves [17, 123], have recently been adopted in various applications to enhance computation through data ordering. A number of orderings based on space filling curves have been identified whose properties lead to variations in computation performance. Their property of grouping related data into virtual blocks means that the elements of a block can be held together in high level memory, if the block size is aligned with the size of the high level cache. In addition, processing data along the path of a space filling curve has a high potential for spatial and temporal data locality as the data elements are stored in consecutively in memory and elements are located within the virtual block close to other data elements upon which their processing depends.

Molecular dynamics simulation (MDS) is a useful tool that uses a deterministic approach to simulate a molecular system by following the movement of particles and their status at each time step of the simulation. The interaction forces and potential energy of a molecular system depend on the adopted numerical model. In this research the Lennard-Jones model [47, 72] is used to compute particle interactions, and deciding which particles interact is a compute-intensive task that depends on the distance between particles. In the Lennard-Jones model, particles that are sufficiently far apart do not interact. However, particles that are closer may interact, and each particle has a set of neighbouring particles with which it potentially interacts. Only computing the interactions with these neighbouring particles reduces the computational complexity for the inter-particle force computation.

GPUs have a complex memory architecture and a high level of dependency between the hardware components. In addition, the application programmer has little direct control over the scheduling of threads or the movement of data between levels in the memory hierarchy. Therefore, application programmers are encouraged to follow best practices in programming style that coerce the compiler and the runtime system into running code efficiently, resulting in optimal utilization of the system architecture. NVIDIA graphics cards and the CUDA programming language are used in this study. CUDA is highly compatible with NVIDIA GPUs, as they are provided by the same vendor, which can result in high computation performance. In addition, NVIDIA profiling tools enable us to investigate the relation between the data locality properties of MDS applications and the ordering of particles in the GPU implementation.

## 1.2 Problem Definition

The allocation of data in memory, the degree of dependency between different memory locations, and the movement of data across the memory hierarchy, significantly affects application performance. In MDS applications, the generation of the neighbour lists (discussed in 2) engenders a high degree of data dependency.The ordering of the particles is based on which spatial bin they are located in, and in turn these bins usually have a row-major ordering. For example, Fig. 1.1 represents a simple 3-dimensional array of size $5 \times 5 \times 5 \times 5$ and an approximately spherical stencil of width 5 (for simplicity). The yellow shading shows the bins that the bin with index 62 interacts with for this stencil. The lower part of the figure shows how these bins are scattered in the memory, and this scattering can degrade application performance because data are fetched from the memory hierarchy in blocks. Consequently irrelevant data will also be fetched and a larger number of memory accesses will be required to fetch the data, compared with the case of consecutive data. Consecutive data have a high likelihood of being loaded by one memory access, subject to other factors, as explained in Chapter 2, such as cache size, cache line size, and the caching policies.

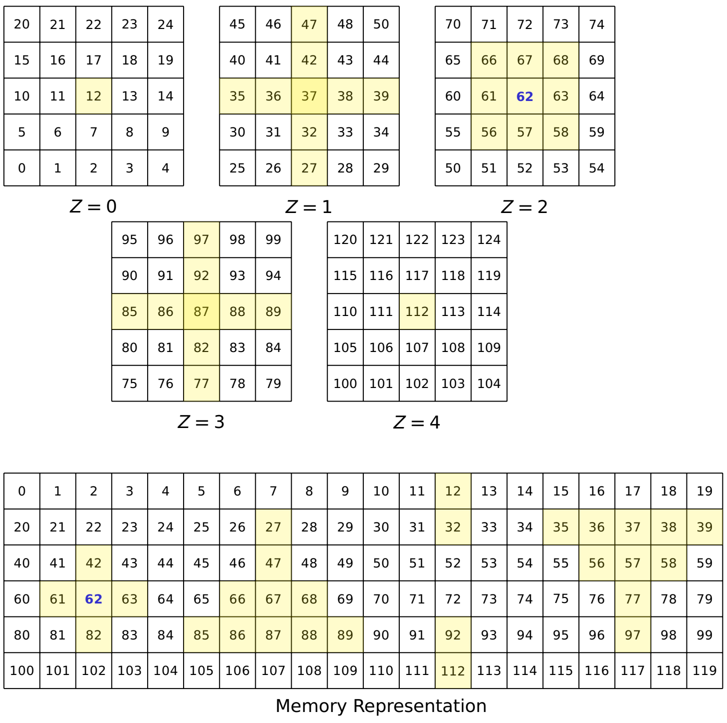


**Figure 1.1: Simple example of 3-dimensional array with approximately spherical stencil, and its representation in memory.**

A limited number of research works have addressed the data locality properties of MDS and most previous studies have focused primarily on reducing the number of neighbour list particles to make it, as close as possible, the same as the actual interaction list, as will be seen in Chapter 3. Exploring the data locality properties of MDS enables deeper understanding of the MDS computations and optimises their performance. In addition, different data ordering have different characteristics that distinguish them. Thus, their adoption lead, to differences in performance indicators in terms of execution time and memory hierarchy utilization. This type of research can then be applied to different types of application that might have different types of data dependency relationships. Data locality properties also have a high dependency on the memory hierarchy architecture, and therefore conducting this study on GPU architectures will examine the approach to analysing various data orderings and their utilization of the memory hierarchy.

Therefore, in this research, we will answer the following research questions:

- What is the impact of different computation stencils on the data locality properties of MDS applications?
- What is the impact of data locality on MDS applications implemented on GPUs?
- Do space filling curves (SFC) have better data locality properties compared with conventional row-major orderings?
- How should data locality and performance analysis be conducted for GPUs?

## 1.3 Hypothesis

Increases in processor speed have not been accompanied by a commensurate increase in memory speed. Therefore, to exploit a processor's capabilities, the various levels in the memory hierarchy must be efficiently utilized. Therefore, it is hypothesised that data locality enhancements in an application lower the cost of data movement across the memory hierarchy of a GPU by optimising data reuse in the higher levels of memory. This increases the likelihood of finding data in the first or the second level of cache, which, in turn, reduces the latency of global memory accesses. Molecular dynamics simulations have strong data dependencies as the force computation for a particle depends on its neighbouring particles. Therefore, we study the properties of various data orderings in the context of a molecular dynamics simulation on GPUs to investigate the optimal data ordering for this type of application.

## 1.4 Aims and Objectives

The main aim of this thesis is to present the relationship between different data orderings and computation stencils, and their performance in molecular dynamics simulations. It aims to provide a detailed analysis of the spatial and temporal data locality properties for each of the adopted orderings. The efficiency of an ordering is not only measured by the runtime of the application, but also by the properties of the stencil used, and a detailed analysis of the memory profile. In doing this, the aim is to increase knowledge of the data locality properties of molecular dynamics simulations on GPU platforms.

In order to achieve our aims we conducted the followings:

- The implementation of the space filling curves data orderings.
- The implementation of the cache model to explore the data locality of the implemented data orderings. This includes the analysis of the cubic and approximate spherical relationship with implemented different orderings.
- GPU version of the Lennared Jones implementation and conducting the required modification to the MiniMD version integrated with space filling curves data orderings.

- Executing the applications in two different GPUs and with various configurations and simulation's sizes.
- NVIDIA profiler is adopted for memory profiling and caompred with cache model results.

## 1.5 Thesis Contributions

The main contributions presented by this thesis are:

- The data locality properties for different orderings have been studied from three perspectives: the relation between a data ordering and an adopted stencil; the execution time of the force kernel on GPUs; and, memory profiling using the NVIDIA *nvprof* profiler. Most of the previous literature (as will be seen in Chapter 3) does not considered all these prespectives and they are limited to the execution time analysis. In regards to the GPU implementation and analysis they did not utilize NVIDIA profiler. The relation between the stencil and data orderings is not only valid for MDS applications but can be adopted for other applications. Different stencils have a different relationship with each of the adopted data orderings which requires to modify slightly our cache model.
- A GPU implementation of the Lennard-Jones force computation using CUDA, without introducing complex data structures to ensure the alignment with the original CPU version of the miniMD application. This principle enables a fair comparison between the CPU and GPU implementations. A number of literature adopted tree-based data structures which introduces additional overhead cost. In our implementation the modifications to any of the adopted data orderings are of cost $O(\infty)$.
- To the best of our knowledge, this is the first research which has been conducted into the data locality properties of molecular dynamics simulations, especially

for Lennard-Jones force computations on GPUs. The implemented cache model and NVIDIA profiler with the adopted orderings have not been covered by any previous literature. Thus, this will provide good understanding of MDS data locality in GPU by using different data orderings.

- To the best of our knowledge, this is the first GPU profiling study for the Lennard-Jones case to analyse the use of various levels in the GPU memory hierarchy. The adoption of NVIDA profiler is not only shows how to utilize such tool as part of MDS tool or any other applications but also to confirm the cache model analysis and the experiments results how they are aligned.

- This work also acts as a framework for investigating the data localities of other applications that cannot be readily tiled/blocked as independent data. The approach adopted in this research can be adopted as guideline to conduct performance analysis not only for MDS applications but with any other similiar applications that are implemented within GPU.

## 1.6 Thesis Structure

In this multi-disciplinary research, our thesis is structured as follows:

- Chapter 2 introduces background knowledge with regard to the concepts that are used in the rest of this thesis. A general background to the concept of data locality is given, and issues and limitations are discussed in the context of a GPU implementation. Space-filling curves are explained in terms of their geometric and arithmetic representations. It also explains the different types of data orderings that are used in this research.The various orderings are: row-major, Hilbert, Morton, hybrid Hilbert, and hybrid Morton. Molecular dynamics simulations are introduced, and the main principles explained. The justification for using the Lennard-Jones force model in our computations is explained. This chapter also provides details of the essential concepts and techniques surrounding the main principles of molecular dynamics simulations. Parallel implementation models, especially for general-purpose GPU architectures and programming models, are introduced. The GPU architecture and CUDA implementation are described in detail in order to understand the algorithm and the implementation chapter of this research. Finally, key factors in terms of optimization and performance analysis that need to be considered, and are adopted in this work, are identified.

- Chapter 3 presents a literature review conducted on all aspects of our study. It shows the research that has been carried out to optimize the data locality from the perspective of optimizing the computation and data order. The use of space filling curves for data ordering enhancements in various applications is explored. Data locality and optimization in molecular dynamics simulation studies conducted in this field are provided, especially in terms of their implementation on GPU platforms.

- Chapter 4 clarifies the relationship between data orderings and an approximately spherical stencil, which is widely adopted in various applications, especially in molecular dynamics simulations. In addition, a simple cache model is used to explore in detail the relation between our data orderings and the approximately spherical stencil.

- Chapter 5 provides an overview of the miniMD implementation that is used in our performance evaluations. Its various key modules and components are explained in detail, especially the Lennard-Jones force computation module, which is used as part of our research to evaluate the impact of various data orderings. Then we explain the implementation of our work and the modifications that have been incorporated into the original miniMD implementation. The force computation module for full and half neighbour list force computations on the GPU, and the ordering module, are the main modules that are described in this chapter.

- Chapter 6 describes all the various experiments, results and analysis that have been conducted in our research. In this chapter, the hardware used for conducting this research is described in detail. In addition, all the setup of the various experiments, the simulation size, the data orderings, and the platforms used in each experiment are identified prior to the discussion of the experimental results.

- Chapter 7 concludes the thesis, linking the thesis contributions to the relevant chapters, and discussing this work's limitations and making recommendations for future work.

## 1.7 Summary

To sum up, our thesis is as follows: to study and analyse the data locality properties of various data orderings for a molecular dynamics application implemented on NVIDIA GPUs. In this chapter, an overview of our research and its hypothesis were briefly introduced, and the aims and objectives of this work were described. In addition, the research contributions and thesis structure were provided. The next chapter provides more detailed information on the problem addressed.

# Chapter 2

# Background

This chapter introduces essential background information about the concepts that are used in the rest of this work. First, the concept of data locality is introduced, together with its importance as one of the factors that impact computation efficiency. A general overview of data locality obstacles, issues, and limitations will be presented. Specifically, GPU data locality problems will be defined. Then, some of the solutions that have been introduced in this field will be considered, before further details are provided in the literature review (Chapter 3). After that, data orderings and their contribution to data locality will be briefly introduced. MDS will be outlined, and the reasons for performing a data locality study for an MDS application will be explored. A number of simulation tools that have been investigated will be introduced, and the reason for selecting miniMD in this dissertation will be explained. Introductory MDS works have been performed on GPU platforms to investigate the key factors in maximizing GPU utilization. Parallel computing is then broadly discussed, especially computation on GPUs, which is a central aspect of this work. NVIDIA GPUs have been selected in this work, and the reasons for this will be explained, together with their architecture and features.

## 2.1 Data Locality

Data locality, also known as the principle of locality, is the propensity for existing data in high-level memory to be accessed repetitively before being expelled to a lower level in the memory hierarchy. Two types of data locality are distinguished: spatial data locality and temporal data locality. Spatial data locality represents how the data are consecutively ordered in the memory such that consecutive instructions access a contiguous data set, or in a thread-based implementation, the same instruction is applied to consecutive blocks of data. This feature is based on the fact that the data are fetched from low level memory into a high level memory in contiguous blocks known as cache lines. This reduces the number of data fetches needed for each instruction. On the other hand, temporal data locality, also known as reuse data locality, refers to the repeated use of a specific data and/or resource by the same instruction, such as processing the same single data item in a loop, or by other consecutive instructions that depend on the same data. As with spatial locality, a high degree of temporal data locality reduces the number of high latency requests for the same data from low-level memory, and as a result improves the overall application performance.

A brief background on the general cache architecture will be presented in order to identify the various memory components that affect computation performance. In addition, the next section will explore cache issues and identify various solutions, as well as how these could be enabled. Then, ways of enhancing data locality will be introduced, and related features and terminology will be discussed.

### 2.1.1 Cache Architecture

Rapid advances in processors (CPUs) and media has accelerated the movement of data between different device components, and as a result has left hierarchical cache and memory components as a bottleneck for achieving high computational performance. The memory hierarchy varies from one architecture to another according to a number of factors, such as the number of memory hierarchy levels, their size, the bandwidth and latency between levels, and the configuration policy that is applied. The top level of the memory hierarchy has the smallest, fastest, and most expensive memory per bit, while the largest, slowest, and cheapest memory per bit is at the bottom level [112, p. 12]. A memory hierarchy, as shown in Figure 2.1, is the simplest and most widely used way to evade the “memory wall" problem, defined by Wulf and McKee in [153]. For example, the recent Intel Coffee Lake architecture (for core i5/i7 8000 series processors) has six cores, each with an L1 instruction and data cache size of 32 KB, an L2 cache of 256 KB, and an L3 cache size of up to 2 MB per core (where the L3 cache is shared between the cores by a ring based connectivity [5, 8, 23]). Intel has announced that the size of all the caches will be doubled in the new Ice Lake microprocessor architecture, which is scheduled to be on the market in 2020 [9]. Another factor affecting cache performance is the cache replacement algorithm, also referred to as the cache policy. The most common policies are the Random, Least Recently Used (LRU), Partial LRU (PLRU), and Least Frequently Used (LFU) policies. Random replacement removes an arbitrary block to insert a new candidate block. It does not record any information about the request history, which is one of its advantages. LRU replaces the least-recently accessed data in the cache with the new candidate, if the cache is full. This process is maintained by using an age-bit to ensure the discarded block is the least-recently inserted or modified data. PLRU is an approximation for the LRU algorithm in which the access information is recorded by using a binary tree [73]. LFU records the number of accesses to each block in the cache and the block with the minimum number of accesses is discarded and replaced by the new block. Belady’s algorithm is based on future data use, rather than on historical information, wherein the evicted data are those which are not going to be used in the next instructions [136]. These algorithms have trade-offs between miss rate and latency time as they have a high dependency on the associativity degree.

Associativity organizes the cache blocks into $S$ sets, such that $S = B/N$ where $B$ and $N$ are the total number of blocks and the degree of associativity, respectively. If the

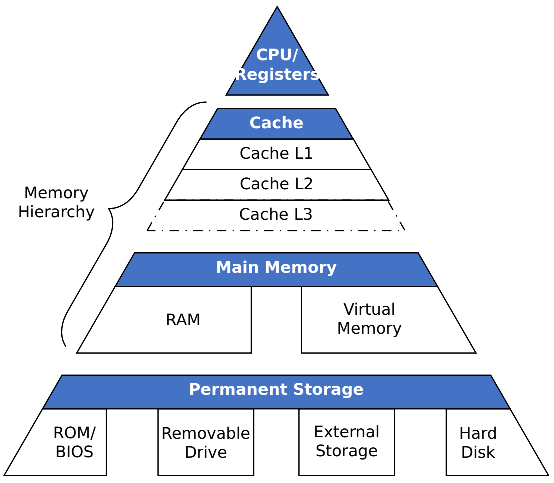


**Figure 2.1: High level logical presentation of memory hierarchy**

number of sets equals the number of cache blocks (i.e., $S = B$), then this case is known as direct mapping. On the other hand, if the associativity is equal to the number of blocks, then it is called a fully-associative cache. Otherwise, it is known as an $N$-way associative cache [62, p. 482]. The advantage of this is to reduce the need to search the full cache and optimize the cache utilization. In the fully-associative cache case, a search for an address requires the full cache to be scanned. Thus, if the requested data is situated at the end of the cache, all the preceding cache lines must be checked to fetch the requested data, which results in a high latency. Thus, a higher degree of associativity comes with the cost of increasing access time [112, p. 77].

The Translation Lookaside Buffer (TLB) is another type of cache that is located in the memory management unit. The TLB records recent virtual or physical translated addresses and, therefore, it resides between the processor and cache, or between different caches levels, or between the CPU and main memory. Thus, an architecture may have multiple TLBs. For example, in the Coffee Lake microarchitecture [5] the L1 data and instruction caches each have a dedicated TLB, and another is provided for the L2 cache. In general, TLBs are fully associative as they are small, and TLB misses are costly. In addition, various types of register are part of the CPU, varying from architecture to architecture. Normally they are very limited in size, and have specific functionality.

### 2.1.2 Types of Cache Miss

Having defined the memory hierarchy in a simple way, the different scenarios in which a cache does not fulfil a request will be described:

- A *compulsory miss* occurs the first time data in a block is requested starting with an empty cache. Therefore, the data is not yet loaded into the cache, and is not in the TLB either; nor is its value stored in any register. This is also known as a cold start miss. There is nothing that can be done to reduce the impact of this type of miss as the cache line and cache capacity are finite in size.

- A *capacity miss* occurs if the requested data or instruction is not available in the corresponding cache and the cache is full. Thus, this necessitates removing one of the cache lines currently in the cache according to the replacement policy and inserting the new cache line.

- A *conflict miss* occurs when many memory accesses are mapped to the same index set in a cache. The likelihood of a conflict miss increases with cache line size, since the number of blocks within the cache is large. A fully associative approach with a LRU replacement algorithm would eliminate conflict misses [76].

- A *coherence miss* results from being unable to maintain the consistency between various caches, especially in multiprocessing architectures and distributed systems [130]. For example, L3 caches are shared between the processors in the Coffee Lake architecture, as mentioned earlier. Thus, data coherency is essential to enable processors to load the recently updated data by any other processor. L3 data coherency, consequently introduces other issues, such as identifying the type of instruction accesses and preventing modification by different processors at the same time. Serialization and scheduling of processor modifications to mutually exclusive data requires a hardware and software implementation.

### 2.1.3 Cache Use Optimizations

The memory hierarchy facilitates large data bandwidths by accessing multiple data items in blocks and reducing the memory latency by enabling independent parallel tasks between processors and various cache levels. However, there are still opportunities for optimising the use of cache in order to achieve high performance. Such optimisation could require specific hardware and/or software features to overcome the performance impact of cache misses. These can be categorized as: reducing the penalty incurred by a cache miss; minimizing the number of misses; and reducing the number of memory accesses [125].

The penalty of a cache miss refers to the cost in time during which a processor is stalled waiting for the completion of a memory access [73]. In this scenario, the solution is focused on hiding the latency of loading the data from main memory into cache. This can be done using a *non-blocking* (or lockup-free) cache that allows the processor to run the next instruction assuming that next execution cycle does not involve the current missed data. In fact, it can allow one miss or multiple misses known as “hit-under-miss” and “miss-under-miss”, respectively. Therefore, this approach requires hardware and compiler-optimizations to handle all the missed addresses, their destinations and data dependencies, and to manage the data consistency due to various load and store misses, especially within the same cache line. Prefetching is another approach, where the data are loaded before they are needed. The prefetching might be triggered by hardware, or by an instruction, or by a combination of both. In general, software prefetching methods are based on the compiler’s ability to analyse a program’s behaviour and predict the required prefetching instruction as part of the program executable. On the other hand, hardware methods utilize spatial locality or the access pattern of the executed program dynamically (see [157] for a list of relevant hardware and software algorithms). One of the challenges for prefetching methods is to fill the pipeline with prefetched instructions rather than instructions ready for execution. In addition, prefetching methods may introduce too much overhead time.

The second factor is reducing the number of misses, which could be accomplished by increasing the cache size, or by increasing the number of levels in the memory hierarchy, or by using additional buffers between various levels. For example, Jouppi in [75] proposed introducing three types of cache between the L1 and L2 caches: the miss, victim, and stream caches. As discussed earlier, the replacement algorithm and degree of associativity, if supported by the hardware architecture, can be adapted according to the program benefits. In addition, compiler optimization is an essential component in reducing cache misses, and this will be discussed in more detail in the next chapter. A common compiler optimsation is to order the instructions to minimize cache misses and to reduce data swapping between cache memory and registers. In multithreading implementations, a competitive miss cache is one of the key concerns, especially within shared caches [117].

Finally, access patterns, which are a focus of this work, are one of the important aspects that are in the hands of application developers and compiler designers. As compiler optimization contributes to reducing the number of cache misses, restructuring of the instructions with the required data plays a part in achieving high performance. In fact, improving the access pattern reduces the cache misses for various buffers and the memory hierarchy. The access pattern exploits the locality of reference principle, which has two aspects: *spatial* and *temporal* locality. Spatial locality refers to the degree to which data that are close in the cache space are used by successive instructions in a consecutive approach. Temporal locality (also called *reuse* or *intrinsic reuse*), refers to when data loaded into a cache are used repeatedly before being evicted . In this situation, the number of dispatches and loads will be reduced and excessive accesses to the lower cache levels are minimized. In addition, enhancing the access pattern can expose parallelism by vectorizing loop iterations.

In order to ensure the access pattern has good locality of reference, the data layout can be reordered so that the data are organized appropriately for an optimizing compiler. For example, if a compiler orders matrices in row-major order then the corresponding data should be in row-major order. Similarly, if a compiler orders matrices in column-major order, such as with Fortran compilers, then the data should be in column-major order. Prefetching methods order the required instructions and data based on the principle of locality of reference, and an algorithm's efficiency is based on the instructions and data orderings aligning for execution at the right time. Similarly, loop transformation methods are intended to change the order of execution of loops so that data are accessed with greater temporal locality.

There are a number of methods to enhance data locality, such as using sorting and search algorithms and adopting various conventional data structures: queues, lists and different types of tree (some of these methods will to be discussed in the literature review in Chapter 3). Padding has been found to reduce cross interference of array references [84]. However, if the padding is within an array it is normal to replace an array of structures by a structure of arrays, and this results in filling the cache line with data that is not going to be used. In addition, data merging from various data structures into one structure is another approach to enhance spatial locality. Array transposition can be applied for two-dimensional arrays, and has a similar result to loop interchanges, and enhances data locality. Reordering the data into blocks or tiles has significant spatial and temporal locality benefits, especially for parallel implementations.

In this work, the row-major, Hilbert (a space-filling curve), Morton and hybrid orderings are used to study the data locality properties of different orderings for molecular dynamics simulations. Background information on space-filling curves and molecular dynamics simulations is given in the following subsections.

## 2.2 Space-Filling Curves

The space-filling curve (SFC) was introduced by Peano as a continuous mapping of the line segment [0, 1] onto the unit square, and this concept was then further developed by Hilbert and Sierpinski. It is a significant tool used in various types of application, such as big data analysis [105], computer graphics, networking algorithms, and volume slicing. It has been used to ensure a high level of data access coherency, wherein data can be stored in memory in the traversal sequence furnished by the SFC traversal pattern. This dissertation focuses mainly on the Hilbert and Morton orderings, thus in this section the definition of a space-filling curve, its generation and related algorithms and applications will be introduced.

### 2.2.1 SFC Definition

In general, a space-filling curve maps the multi-dimensional space (or multi-dimensional data or key values) into a one-dimensional space (a single key value). Bader in [17, p. 17] has defined it as follows:

**Definition 2.1** (Space-filling Curve)**.** Given a curve $f_{\star}(\mathcal{I})$ and the corresponding mapping function: $f : \mathcal{I} \to \mathbb{R}^n$ , then $f_{\star}(\mathcal{I})$ is called a space-filling curve, if $f_{\star}(\mathcal{I})$ has a Jordan content (area for n=2 or volume for n=3 , . . . ) larger than 0.

The definition above assumes that for a continuous curve, if there is a surjective mapping from $\mathcal{I}$ onto $\mathbb{R}$ such that the curve traverses all the points of the unit square (or volume in three dimensions), then we call such a curve a space-filling curve. It should be noted that the Morton ordering discussed below does not correspond to an SFC because it lacks the necessary continuity property. In terms of our general definition, a SFC passes through all the various cell elements (coordinates) in the three-dimensional space and labels them by a single index without missing any one of those cells. This path represents the order in which those cells will be consecutively visited.

A SFC can be constructed based on geometric, grammatic, and arithmetic representations. The grammar case normally uses a geometric operation, which is adopted in our work, (see 2.3). Basic geometric and arithmetic constructions will now be introduced for generating Hilbert and Morton orderings.

### 2.2.2 Geometric Representation

A Hilbert curve can be defined recursively in terms of a basic pattern, $h_1$, composed of three unit vectors, as shown in figure Figure 2.2. The second level Hilbert curve, $h_2$, can then be defined by geometric operations on $h_1$. Similarly, the third level Hilbert curve can be defined by geometric operations on $h_2$ (see figure Figure 2.2). The geometric operations used are translation, reflection, and rotation [106] Thus, it is possible to construct $h_n$ from a previous curve $h_{n-1}$ by using the sequence of operations given in Table 2.1, where $w_{n-1}$ denotes the width of $h_{n-1}$. Note that $w_1 = 1$ and $w_n = 2w_{n-1} + 1$

| Operation Number | Operation(s) |
|---|---|
| 1 | Rotate $h_{n-1}$ by 90° anticlockwise about the origin and reflect in $y$-axis |
| 2 | Shift $h_{n-1}$ by $(w_{n-1} + 1, 0)$ |
| 3 | Shift $h_{n-1}$ by $(w_{n-1} + 1, w_{n-1} + 1)$ |
| 4 | Rotate $h_{n-1}$ by 90° anticlockwise about the origin, reflect in the $x$-axis, and then shift by $(w_{n-1}, 2w_{n-1} + 1)$ |

**Table 2.1: Construction of Hilbert curve $h_n$ based on previous $h_{n-1}$ curve where the edge length is maintained constant (without scaling down the next level to fit the area of previous level). The last point in each of the first 3 component parts is then connected to the first point of the next part.**

Anticlockwise rotation by angle $\theta$ can be performed by using matrix notation as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.1}$$

The reflections in the $x$ and $y$ axes can be performed using the following equations, respectively:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.2}$$

As an example, consider the construction of $h_2$ from $h_1$, where $w_1 = 1$ and $h_1$ has vertices $\{(0,0), (1,0), (1,1), (0,1)\}$. The first operation performs an anticlockwise rotation by 90° about the origin, followed by a reflection in the $y$-axis, as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 90 & -\sin 90 \\ \sin 90 & \cos 90 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x \end{bmatrix} \tag{2.3}$$

The second operation is a shift by $(w_1 + 1, 0) = (2, 0)$. The third operation is a shift by $(w_1 + 1, w_1 + 1) = (2, 2)$. The fourth operation performs an anticlockwise rotation by 90° about the origin, followed by a reflection in the $x$-axis and a shift of $(w_1, 2w_1 + 1) = (1, 3)$ as follows:

$$\begin{aligned} \begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} w_1 \\ 2w_1 + 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \cos 90 & -\sin 90 \\ \sin 90 & \cos 90 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 - y \\ 3 - x \end{bmatrix} \end{aligned} \tag{2.4}$$

This approach is also valid for generating level $h_1$ from the single coordinate point $h_0 = (0, 0)$, and only requires that the width value be taken as $w_0 = 0$ (as the length of a point is zero). Further details of the matrices for shifting, reflecting and rotating can be found in [85, p. 22, 23], where a general notation is provided.

Similarly, Morton order can be generated by using the operations provided in Table 2.2 to obtain the results shown in Figure 2.3. Obviously, all the operations are based on shifting without any rotation or reflection required. It is worth mentioning that the basic motif of the Morton curve shape can be presented in different forms, such an "N" or various flipped "Z"-patterns, depending on the definition of the $x$ and $y$-axis orientation and direction.

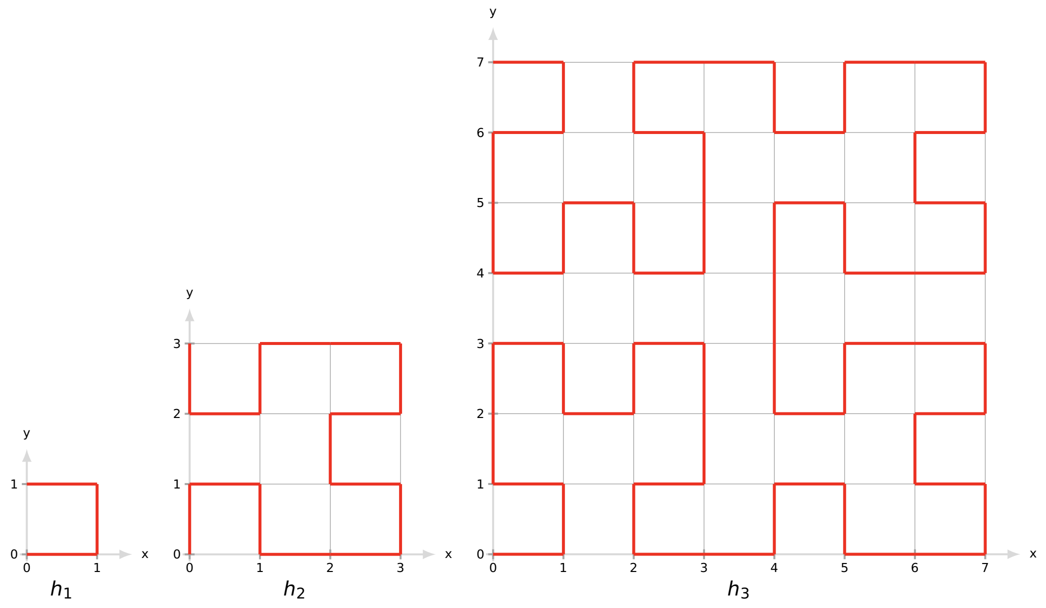


**Figure 2.2: Each of of the Hilbert orders ($h_1$, $h_2$, and $h_3$) is constructed based on the predecessor pattern ($h_{n-1}$) that is manipulated based on the defined operations in Table 2.1. The curve continuity feature connects all the points within the defined Hilbert size.**

| Operation Number | Operation(s) |
|---|---|
| 0 | Shift $m_{n-1}$ by $0, 0$ (copy) |
| 1 | Shift $m_{n-1}$ by $w_{n-1} + 1, 0$ |
| 2 | Shift $m_{n-1}$ by $0, w_{n-1} + 1$ |
| 3 | Shift $m_{n-1}$ by $w_{n-1} + 1, w_{n-1} + 1$ |

**Table 2.2: Morton geometrical operations for constructing $m_n$ based on the previous $m_{n-1}$ pattern.**

## 2.2.3 Arithmetic Representation

The geometric representation of a Hilbert or Morton ordering, based on [18, Chapter 4], can be used to provide a mapping between a position, $q$, in the ordering and the corresponding integer coordinate location. For example, in $h_2$ in Fig. 2.2, the point at index $q = 4$ along the curve corresponds to coordinate location $(2, 0)$, assuming that indexing starts at 0. A general arithmetic way of computing such mappings is required. For a Hilbert curve the basis of this mapping is:

- A given coordinate location is represented in terms of nested intervals, such that

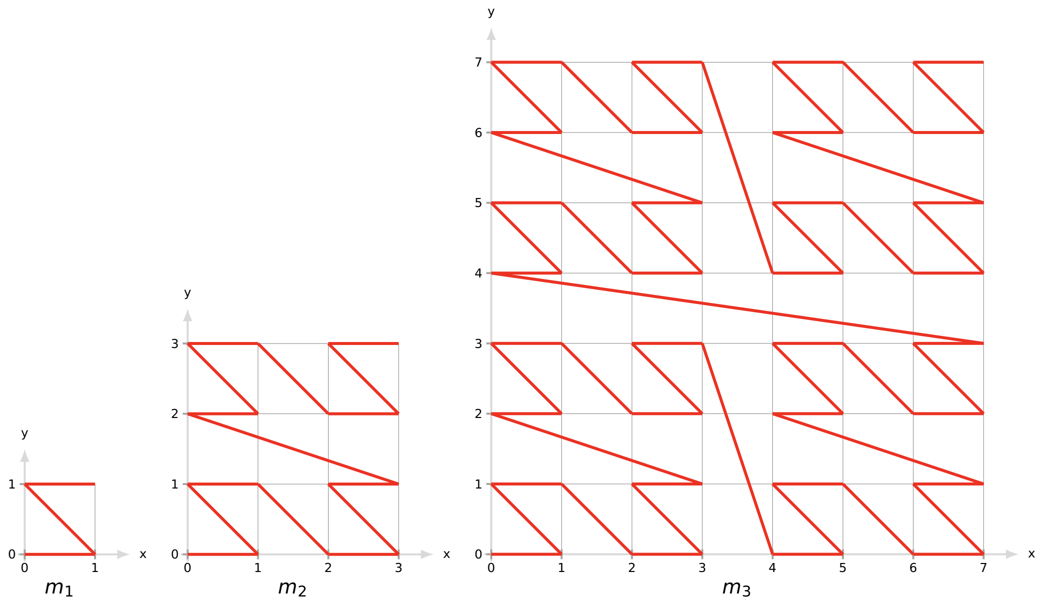


**Figure 2.3: Each of the Morton orderings ($m_1$, $m_2$, and $m_3$) is constructed based on the predecessor pattern ($m_{n-1}$) that is manipulated based on the defined operations in Table 2.2.**

each interval is one of the four quarters of its parent interval.

- The index within the ordering is then found by applying scalings and the geometric operations defined in Table 2.1 to the nested intervals representation.

Given an index space of size $2^n \times 2^n$, the nesting is expressed in terms of a quaternary (i.e., base 4) representation of the fraction $t = q/4^n$ as $t = 0_4.q_1q_2 \ldots q_n$. Let $h(t)$ be a vector representing the coordinate position of $t$ within the unit square. Then, as shown in [18, Chapter 4], $h(t) = H_{q_1}h(\tilde{t})$, where $\tilde{t} = 0_4.q_2q_3 \ldots q_n$. Thus, $h(t)$ is given by:

$$h(t) = H_{q_1} \circ H_{q_2} \circ \cdots \circ H_{q_n} h(0_4.0) \tag{2.5}$$

where $h(0_4.0) = [0\ 0]'$, and the $H_{q_i}$ represent the rotation, reflection and translation operations described in Sec. 2.2.2. The $H$ transformations are:

$$\begin{aligned}
H_0 &= \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} & H_1 &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix} \\
H_2 &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} & H_3 &= \begin{bmatrix} 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} - \alpha \\ 1 - \alpha \end{bmatrix}
\end{aligned} \tag{2.6}$$

where $\alpha = 0.5/(w_m + 1)$ and $w_m$ is width of the Hilbert pattern at the current depth of recursion. These equations are scaled versions of the geometric transformations introduced in Sec. 2.2.2. The scaling is necessary to keep all coordinate positions within the unit square.

For example, suppose $n = 3$ so the index space is $8 \times 8$, and $q = 56$. Then $t$ is the quaternary representation of $56/64$, which is $t = 0_4.320$, so $\tilde{t} = 0_4.20$. Thus,

$$\begin{aligned}
h(t) &= H_3 \circ H_2 \circ H_0 \begin{bmatrix} 0 \\ 0 \end{bmatrix} = H_3 \circ H_2 \begin{bmatrix} 0 \\ 0 \end{bmatrix} & (2.7) \\
&= H_3 \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} + \begin{bmatrix} \frac{3}{8} \\ \frac{7}{8} \end{bmatrix} = \begin{bmatrix} \frac{1}{8} \\ \frac{5}{8} \end{bmatrix} & (2.8)
\end{aligned}$$

where the width of the Hilbert curve in applying $H_3$ has been taken as 3. The value of $h(t)$ corresponds to the position $(1, 5)$ in the representation of $h_3$ in Fig. 2.2.

Computing the inverse mapping finds the parameter $t$ corresponding to $(x, y)$ on the Hilbert curve, such that $h(t) = [x\ y]^T$. First the quadrant, $q$, that the point lies in is found based on whether $0 \leq x < 0.5$ and $0 \leq y < 0.5$. The quadrants are labelled as follows:

| | |
|---|---|
| 3 | 2 |
| 0 | 1 |

The position within the quadrant $q$ is then found by applying the inverse operator $H_q^{-1}$ to the position, to obtain $[\tilde{x}\ \tilde{y}]^T = H_q^{-1}[x\ y]^T$. This process is then repeated recursively.

At each level of the recursion one quaternary digit of $t$ is found, starting with the most significant.

The inverse operators carry out the following transformations:

$$H_0^{-1} := \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} 2y \\ 2x \end{bmatrix} \qquad H_1^{-1} := \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} 2x \\ 2y-1 \end{bmatrix}$$

$$H_2^{-1} := \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} 2x-1 \\ 2y-1 \end{bmatrix} \qquad H_3^{-1} := \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} -2y+2+2\alpha \\ -2x-1-2\alpha \end{bmatrix}$$

Similarly, the Morton order can be arithmetically represented by using the same concept of Hilbert representation, but with different transformation matrices:

$$\begin{aligned} M_0 &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} & M_1 &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix} \\ M_2 &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} & M_3 &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix} \end{aligned} \tag{2.9}$$

An alternative approach to Morton ordering is to express it as a manipulation of the bitwise representation of the column and row array indices, $(x, y)$, to give the Morton index, $k$. For a $2^n \times 2^n$ array of points $k$ is obtained by interleaving the $n$ bits of $x$ and $y$. For example, suppose $n = 3$ and $(x, y) = (4, 5)$. Then 4 and 5 can be expressed as 100 and 101 in binary. Interleaving their bits gives 110010, so $k = 50$ as can be verified from the plot of $m_3$ in Fig. 2.3

## 2.3 Data Orderings

In computational molecular dynamics the simulation box is a 3-dimensional space and is divided into cells. Each of these cells can be labelled (indexed) according to its position in the simulation box. This section discusses how these 3-dimensional cells have been ordered in this work. Normally, the location of an item within a 3-dimensional array of size $M \times N \times P$ can be labelled by $(i, j, k)$, where $i$ is the row number, $j$ is the column number, and $k$ is the slab number. For convenience, row, column, and slab directions can be associated with the $x$, $y$, and $z$ axes, respectively; $0 \le i < M$, $0 \le j < N$, and $0 \le k < P$, are ordered in this work using linear (row-major), Hilbert and Morton orderings. Consequently, an *ordering* is a bijective mapping function, $\mathcal{O}$ , from $(i, j, k)$ to a linear index, $b$, as follows:

$$b = \mathcal{O}(i, j, k). \tag{2.10}$$

where $0 \le b < MNP$ may be interpreted as the offset in memory, measured in number of items, from the position of the first item. A memory address for an item can then be formulated as follows:

$$m_l = memory\_base\_address + b * data\_type\_size. \tag{2.11}$$

where $m_l$, $memory\_base\_address$, and $data\_type\_size$ represents the memory location (address), the first element address of the defined array, and the data type size of the array items, respectively, as shown in Figure 2.4. This is because the data in memory is actually allocated in a one-dimensional array regardless of the size and number of dimensions. The defined data ordering of an array affects the data layout in memory. Ensuring that adjacent matrix elements are next to each other in memory significantly affects the optimization of computations carried out on a regular spatial grid.

### 2.3.1 Linear Ordering

There are two common linear orderings: row-major and column-major order. The row-major order is widely adopted by many recent compilers and is formalized as $\mathcal{O}_R(i, j, k) = j + (i + k * ldx) * ldy$, where $ldx$ is the maximum number of elements in a row and is the offset between adjacent items in the column direction, and $ldx{*}ldy$

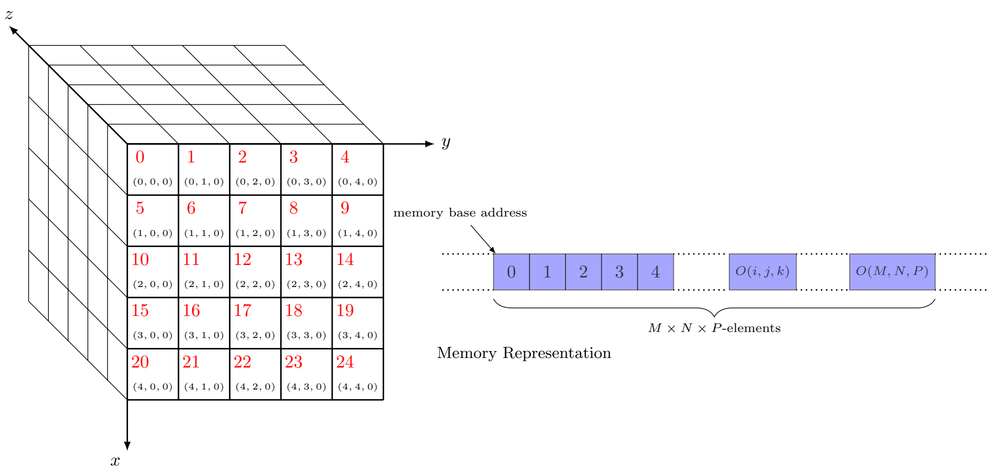


**Figure 2.4: Example of 3-dimensional array labelling and memory representation. In this case, $M = N = P = 5$, and $0 \leq i, j, k < 5$ .**

is the offset between adjacent items in the slab direction. For a row-major matrix, if the number of columns, $N$, equals $ldy$, and the number of rows, $M$, equals $ldx$, then adjacent items in the matrix are ordered contiguously in memory. Furthermore, without loss of generality, it will be assumed that 3-dimensional array is cubical, so $M = N = P$. Therefore, the row-major mapping is presented as follows:

$$O_R(i, j, k) = j + (i + k * M) * M \tag{2.12}$$

The corresponding column-major ordering swaps round the $i$ and $k$ indices. In fact, each permutation of $i$, $j$, and $k$ gives a different variant of the linear ordering, however, we restrict our attention to the one defined by Eq. 2.12.

For a linear ordering adjacent items in a matrix are adjacent in memory with respect to only one dimesion. Thus, for a matrix stored in row-major order, adjacent items in the same row are labelled consecutively by Eq. 2.12 and so are adjacent in memory. However, sequential continuity is violated when moving in the column or slab directions.

In addition, blocked linear orderings can be constructed by dividing the matrix items into a blocks of size $m \times n \times p$. Without loss of generality, it is assumed that the matrix size in each dimension is an exact multiple of the corresponding block size, so the number of blocks is $M/m \times N/n \times P/p$. This approach provides two levels of indexing for each dimension: an index for the block and an index within the block. This type of ordering will be discussed in more detail in Section 2.3.4.

### 2.3.2 Hilbert Ordering

The Hilbert curve is a well-known example of a space-filling curve, and continuously maps the interval unit $\mathcal{I}$ onto the unit $d$-dimensional cube, $\mathcal{Q}$. Based on a geometric generation process, in three dimensions a cube can be subdivided into eight equal sub-cubes. Similarly, in two dimensions a square can be divided into four quadrants. For the sake of clarity, Figure 2.5 illustrates each of the first four sub-squares divided into four further smaller sub-squares, each of which has a length that is half that of the parent square, so the area is reduced by one-fourth. In addition, the sub-squares are connected by a pattern as a result of rotation and/or reflection of the parent square. The continuity of the curve and sequential labelling of the derived sub-square is maintained such that the exit point of each sub-square is the entry point for the adjacent sub-squares along the pattern path.

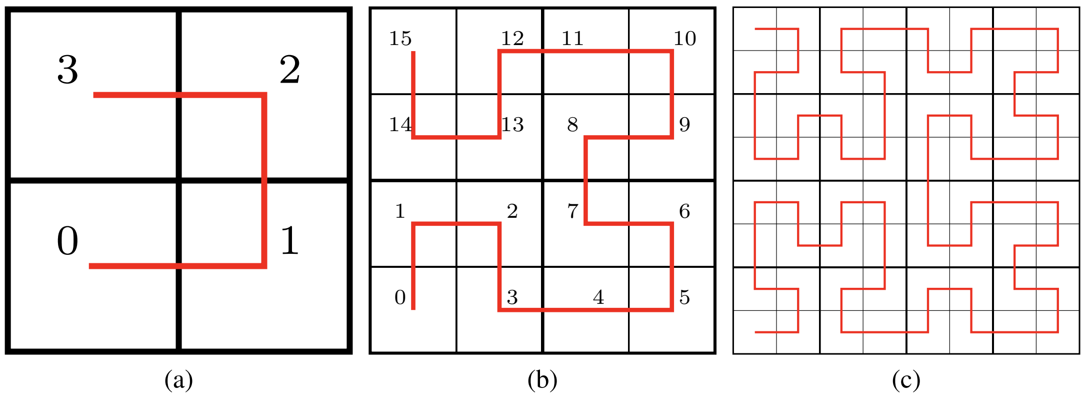


**Figure 2.5: (a), (b), and (c) are examples for Hilbert curves generated for orders $n = 1$, $n =$, and $n = 3$, respectively, and the number of labels is $4^n$ .**

In Figure 2.5, for $n = 1$, the first generated Hilbert curve takes the shape of ⊐, although the Hilbert curve could be initiated with alternative motifs such as ⊔, ⊓, and ⊏. The choice of initial motif does not affect the recursive generation of the Hilbert pattern, but it does have slightly different grammar rules. Furthermore, the Hilbert depth and dimension defines the total number of labels, $M = 2^{dn}$, where $d$ and $n$ represent the Hilbert dimension, and recursion depth (level), respectively, and $n \geq 1$. The relation between Hilbert depth $n$ and $n - 1$ can be represented by a quad-tree. A square with index $b$ of depth $n$ divides into sub-squares that are indexed from $4b$ to $4(b + 1) - 1$.

There are different approaches to represent the generation of a Hilbert curve. A Lindenmayer system (or L-system) in terms of parallel rewrite rules [93, 118] is provided here as an example of how to generate the Hilbert curve in Figure 2.5c as follows:

$$\begin{aligned} X &\rightarrow +YF - XFX - FY+ \\ Y &\rightarrow -XF + YFY + FX- \end{aligned} \tag{2.13}$$

where $F$ means draw a line segment with specific length, "+" means turn 90° right, and "−" means turn 90° left. Applying these rewrite rules recursively generates the Hilbert curve to the corresponding recursive depth. For example, applying the rewrite rules once yields:

$$\begin{aligned} X \rightarrow\ & +(-F + F + F-)F \ - \ (+F - F - F+) \\ & F(+F - F - F+) \ - \ F(-F + F + F-)+ \end{aligned} \tag{2.14}$$

Another grammar-based representation proposed by Bader [18] denotes the motifs as $H = \sqcap$, $A = \sqsupset$, $B = \sqsubset$, and $C = \sqcup$, respectively. Then, for each motif, the following rewrite rules are defined:

$$\begin{aligned} H &: A \uparrow H \rightarrow H \downarrow B \\ A &: H \rightarrow A \uparrow A \leftarrow C \\ B &: C \leftarrow B \downarrow B \rightarrow C \\ C &: B \downarrow C \leftarrow C \uparrow A \end{aligned} \tag{2.15}$$

where the letters on the left side define the initial motif and the letters and arrows on the right side give the corresponding rewrite rule for generating the Hilbert curve. The arrows in the rewrite rule for a motif follow the motif shape from initial point to end point. For example, Figure 2.5a is an initial motif represented by $A$ and applying the rewrite rule 2.15 produces Figure 2.5b. Recursively applying the rewrite rules 2.15 twice draws the Hilbert curve shown in Figure 2.5c, as follows:

$$\begin{aligned} A &\Longrightarrow H \rightarrow A \uparrow A \leftarrow C \\ &\Longrightarrow (A \uparrow H \rightarrow H \downarrow B) \rightarrow (H \rightarrow A \uparrow A \leftarrow C) \uparrow \\ &\quad (H \rightarrow A \uparrow A \leftarrow C) \leftarrow (B \downarrow C \leftarrow C \uparrow A) \end{aligned} \tag{2.16}$$

In a similar way, a 3-dimensional Hilbert curve can be drawn using rewrite rules of the form given in Eq. 2.14 or 2.15, although they are rather more complicated than for the 2D case. For example, the following L-system generates a 3D Hilbert curve:

$$\begin{aligned} X &\rightarrow \wedge < XF \wedge < XFX - F \wedge >> XFX \vee F \\ &\quad + >> XFX - F > X- > \end{aligned} \tag{2.17}$$

where the meanings of the symbols are given in Table 2.3.

| Symbol | Meaning |
|---|---|
| F | Draw line segment |
| + | Yaw 90° |
| − | Yaw -90° |
| ∧ | Pitch 90° |
| ∨ | Pitch -90° |
| < | Roll 90° |
| > | Roll -90° |

**Table 2.3: Meaning of the symbols in the rewrite rule for a 3D Hilbert curve.**

The mapping, $O_H$, gives the index of the Hilbert order for each matrix element. Tracing the 3D Hilbert curve path using the rewrite rule in Eq. 2.3 enables the tracking of the corresponding $(i, j, k)$ index in 3-dimensional space, giving the inverse of the $O_H$

mapping. In a 3D context, the symbol F means “increment or decrement the value of $i$, $j$, or $k$, depending on the current orientation of the axes”. As an example the Hilbert ordering for a $4 \times 4 \times 4$ array is shown in Figure 2.6.

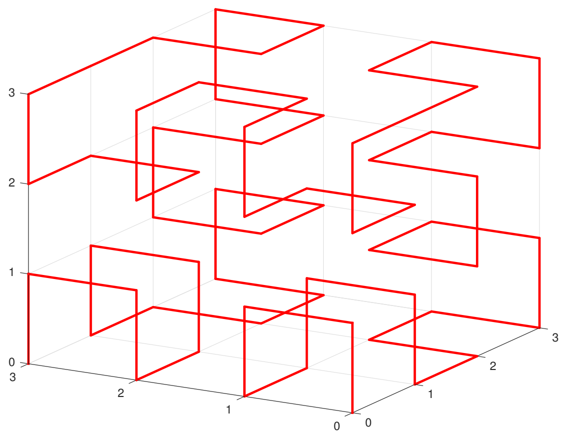


**Figure 2.6: Three-dimensional Hilbert ordering for a $4\times4\times4$ array. The index, $b$, increases by 1 each time the red path passes from one location to another, starting with index 0 at $(0,0,0)$ and ending with index 63 at $(3,0,0)$.**

Using Table 2.3 generates six rotation matrices: yaw, pitch and roll 90° corresponding to the symbols +, ∧ , < represented as follows:

$$Y = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2.18}$$

where −90° rotation matrices corresponding to the symbols −, ∨, and >, are given by $Y^T$, $P^T$, and $R^T$ as follows:

$$Y^T = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad P^T = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \quad R^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \tag{2.19}$$

The orientation of the axes is defined by a heading vector, **H**, and two other mutually orthogonal vectors, denoted by **L** and **U**. These orientation axes depend on the first

Hilbert pattern initialized. Initially, we set **H**, **L**, and **U** to form the orientation matrix $D$, where they define its columns, as follows:

$$\mathbf{H} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \tag{2.20}$$

The defined orientation can be mapped with the rewrite rule such that: a roll corresponds to a rotation of 90° about the **H** axis; a pitch to rotation of 90° about the **L** axis; and a yaw to rotation of 90° about the **U** axis. Therefore, according to the desired recursion depth, the resulting string is processed from left to right, then by post-multiplying the orientation matrix by the rotation matrix corresponding to the current symbol. Thus, $D \leftarrow D\Theta$, where $\Theta$ is one of the six rotation matrices. When an F is encountered a line segment of length 1 is added to the path in the current direction of **H** (the first column of $D$).

### 2.3.3 Morton Ordering

As with Hilbert ordering, Morton ordering uses a recursive method that requires $M = 2^m$, where $M$ is the number of items along an axis. First, we start with the two-dimensional case of an $M \times M$ row-major array. Morton order is applied to this array by dividing it into a $M/2 \times M/2$ sub-array. This division is continued recursively until $m - 1$ levels of recursion have been applied, at which point the sub-array size is $2 \times 2$, as shown in Figure 2.7 .

In 3-dimensional space, the Morton ordering of an $M \times M \times M$ array can be constructed in a similar recursive way. In this case, the array is first divided in a $2 \times 2 \times 2$ array of blocks each of which is an $M/2 \times M/2 \times M/2$ sub-array in row-major order. This approach is then applied recursively by partitioning each sub-array in the same way, so that after $m$ recursions sub-arrays of size $2 \times 2 \times 2$ are achieved. Figure 2.8 illustrates

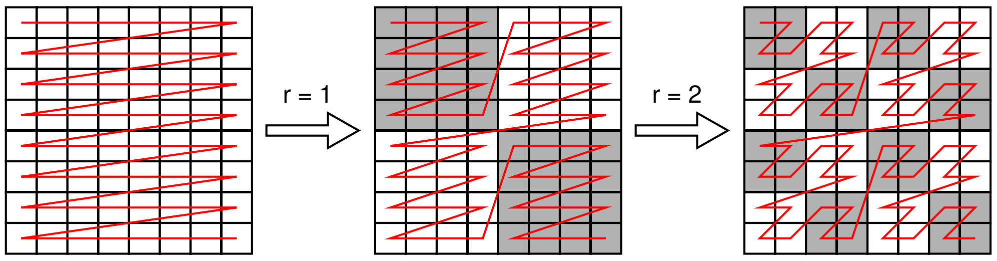


**Figure 2.7: The left-hand part of the figures is the original $M \times M$ array. The middle part shows the result of Morton ordering to level $r = 1$. The right-hand part of the figure shows the Morton ordering to level $r = 2$. In each recursion, the size of sub-array is $M/2^r \times M/2^r$ as shown by the shading which highlights the division into sub-arrays. Each square is labelled starting from the top left corner until the end of the bottom right corner.**

an example of Morton ordering for a $4 \times 4 \times 4$ array.

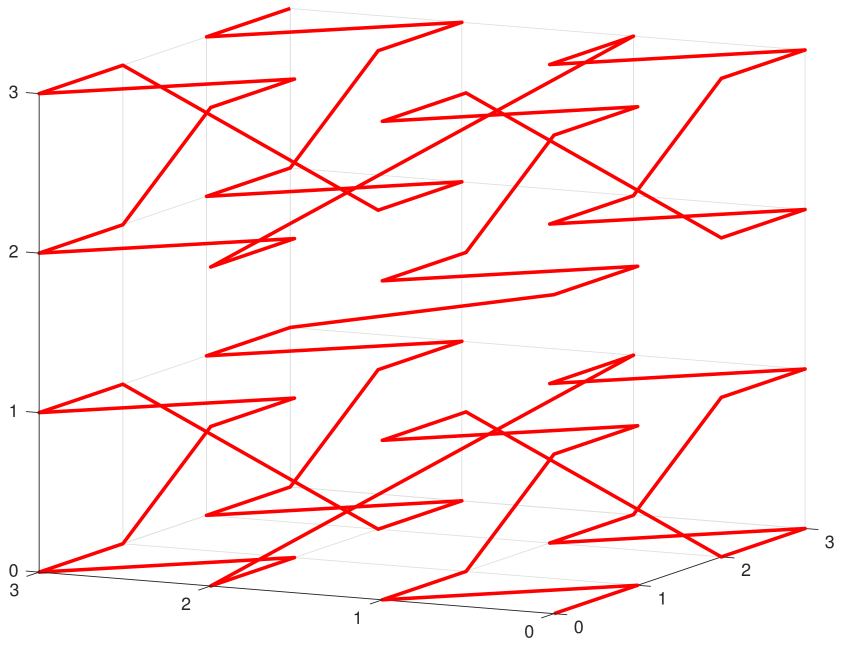


**Figure 2.8: Three-dimensional Morton ordering for a $4\times4\times4$ array. The index, $b$, increases by 1 each time the red path passes from one location to another, starting with index 0 at $(0, 0, 0)$ and ending with index 63 at $(3, 3, 3)$.**

Bit manipulation, is another way that is used to obtain the index, $b$, of a Morton ordering. Given a row-major array, the bits of the Morton index $b$ for a given element at

$(i, j, k)$ is obtained by interleaving the bits of $i$, $j$ and $k$:

$$k_{m-1}j_{m-1}i_{m-1}k_{m-2}j_{m-2}i_{m-2}\dots k_1j_1i_1k_0j_0i_0 \tag{2.21}$$

### 2.3.4 Hybrid Orderings

Hybrid orderings are constructed by splitting an array into sub-arrays such that each sub-array is internally ordered by a specific ordering that may be different from how they are externally ordered. Therefore, the 3D array of size $M \times M \times M$ can be divided into sub-arrays, each of size $T \times T \times T$, where $M = 2^m$ and $T = 2^t$ , then each item within the sub-array can be sorted, for example, by row-major order, while a Hilbert or Morton ordering can be applied between them (provided the number of sub-arrays is the same power-of-two in each direction). The index, $b$, of the ordering consists of $3m$ bits; the lower $3t$ bits are used to encode the ordering within each sub-array, and the upper $3(m - t)$ bits encode the ordering between sub-arrays. In this scenario, the index $b$ for $(i, j, k)$ can be calculated by finding out first which sub-array $(i, j, k)$ belongs to according to the ordering of sub-arrays. This involves manipulating the upper $(m - t)$ bits of $i$, $j$, and $k$. The relative index within the sub-array is obtained by manipulating the lower $t$ bits of $i$, $j$, and $k$.

Table 2.4 gives an example of some combinations of $m$ and $t$ bit sizes. The total number of elements generated is $2^{3m}$. The number of elements within each sub-array is computed based on the size of $t$ bits, $2^{3t}$. Thus, the number of generated sub-arrays (sub-blocks) is $2^{3(m-t)}$. When the sizes of $m$ and $t$ are equal, the number of generated sub-arrays is only one and the internal elements are ordered with row-major order in this case.

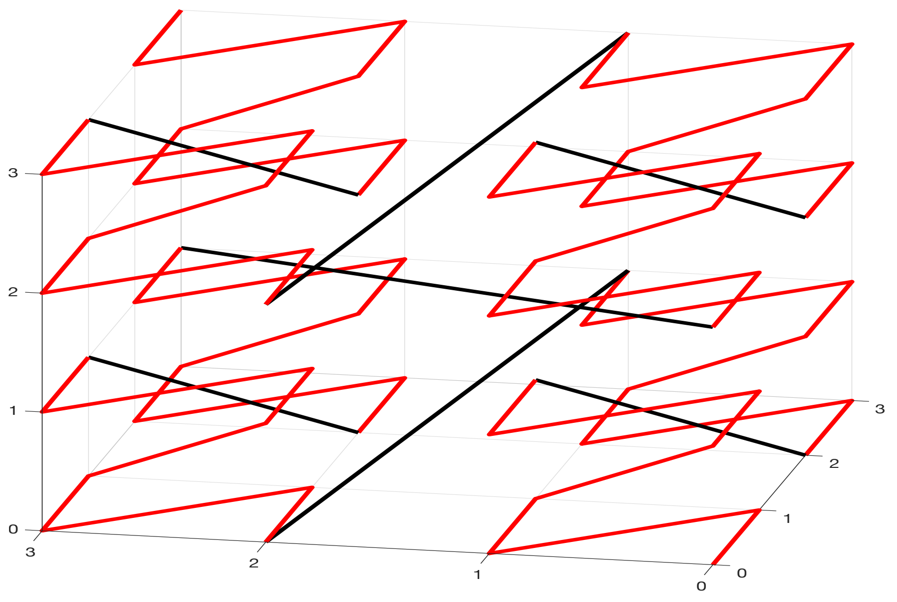


**Figure 2.9: Three-dimensional hybrid ordering by row-major and Morton for a $4 \times 4 \times 4$ array. In this example, $t = 1$ and $m = 2$. Each sub-array consists of $2^{3t}$ items ordered by row-major, presented by the red line. The sub-arrays are ordered by Morton ordering and sorted according to the black line path.**

| | | Internal Order Row-major ordering | | Outer Order Morton-ordering | |
|---|---|---|---|---|---|
| $m - bits$ | Total number of elements | $t - bits$ | Number of elements | $m - t$ | Number of sub-blocks |
| 1 | 8 | 1 | 8 | 0 | 1 |
| 2 | 64 | 1 | 8 | 1 | 8 |
| 2 | 64 | 2 | 64 | 0 | 1 |
| 3 | 512 | 1 | 8 | 2 | 64 |
| 3 | 512 | 2 | 64 | 1 | 8 |
| 3 | 512 | 3 | 512 | 0 | 1 |
| 4 | 4096 | 1 | 8 | 3 | 512 |
| 4 | 4096 | 2 | 64 | 2 | 64 |
| 4 | 4096 | 3 | 512 | 1 | 8 |
| 4 | 4096 | 4 | 4096 | 0 | 1 |
| 5 | 32768 | 1 | 8 | 4 | 4096 |
| 5 | 32768 | 2 | 64 | 3 | 512 |
| 5 | 32768 | 3 | 512 | 2 | 64 |
| 5 | 32768 | 4 | 4096 | 1 | 8 |
| 5 | 32768 | 5 | 32768 | 0 | 1 |

**Table 2.4: Examples of the relation between the number of elements within each sub-block, that are ordered using row-major, and the number of sub-blocks that are ordered using Morton-ordering. Note, if $m - t = 0$ , there will be only one block for which the elements are ordered using row-major ordering.**

## 2.4 Molecular Dynamics Simulations

Molecular simulation is a computational technique used to calculate the time dependent behaviour of a molecular system in which aggregated molecules are represented by particles. Each molecular system has it is own defined attributes and behaviour as a function of time, such as the inter-particle force law, and particle trajectories. These determine the thermostatic attributes of the system, such as its potential energy, temperature, and pressure. In a molecular simulation, the particles interact and their location evolves according to the laws of classical physics [2]. There are two techniques for molecular system simulation: dynamic and Monte Carlo simulation. Dynamic simulation is based on Newton's equations of motion using a time-stepping algorithm in which time can be scaled from pico-seconds to nano-seconds. In addition, dynamic simulation supports various thermodynamic, structural and dynamic properties. On the other hand, a Monte Carlo simulation is based on a statistical ensemble using a random walk algorithm in which there is no true analogue of time. The sampling method can be user-selected and this approach also supports various thermodynamic, structural and dynamic properties. The dynamic and Monte Carlo approaches are distinct from simulations that require in-depth atomic modelling in which *quantum mechanics* is used. Such cases are based on the solution of the time-dependent Schrödinger equation using semi-empirical *ab initio* methods, or simplified approximation methods, such as the Born-Oppenheimer approximation. Quantum mechanics is probabilistic as it calculates the probability of a particle being at a certain place at a certain time [72, p. 27].

Molecular dynamics simulation (MDS) has been widely used in various applications, such as biomolecules, electronic properties and dynamics, surfaces, liquids, and crystal structures. There are a number of challenges in MDS, such as simulation run time, simulation size, and how realistic the adoption of classical forces is within the defined molecular system. Therefore, this dissertation focuses on molecular dynamics simulation, with particular emphasis on run time issues.

### 2.4.1 Main Principles

Given a set of particles with known positions and velocities, molecular dynamics simulations proceed through a series of discrete time steps. In each time step, the pairwise interactions between the particles determines the force on each particle, and each particle is then moved under the influence of this force according to Newton's Second Law of Motion: $\mathbf{F_i} = \mathrm{m_i}\mathbf{a_i}$, where $\mathrm{m_i}$ is the mass of particle $i$ and $\mathbf{F_i}$ is the force acting on it, which induces an acceleration $\mathbf{a_i}$. Thus, given the initial state, $S(t_0)$, at time $t_0$, it is possible to evolve the system through a series of small time steps to find the state at subsequent times: $S(t_0 + \Delta t), S(t_0 + 2\Delta t)$, and so on. The time step, $\Delta t$, is chosen to ensure the accuracy and stability of the method. In updating the system state, new velocities and positions are found for each particle by integrating the equation of motion. Although the trajectories of individual particles may not be accurately computed as the simulation progresses, the molecular dynamics method aims to ensure that the macroscopic properties of the system as a whole are statistically correct [47].

### 2.4.2 Molecular Interaction

Molecular interactions can be categorized as being due to intramolecular and intermolecular forces. Intramolecular forces are between atoms within a single molecule or compound, and include all types of chemical bonds. Intermolecular interactions represent the attraction or repulsion that acts between multiple molecules. Intermolecular interaction is electrostatic in nature which results from the force between positively and negatively charged species. In general, interactions can be classified as *long-range*, where the interaction potential varies as the inverse of the distance between the particles, and *short-range*, where the interaction potential decreases rapidly with distance [132]. Examples of long-range interactions are electrostatic, induction and dispersion interactions, while short-range interactions include exchange and repulsion.

In *molecular mechanics* (another term for molecular dynamics), particles relations are described in terms of “bonded atoms”, which have been deformed from some idealized geometry due to non-bonded *van der Waals* and *Coulombic* interactions [65]. Thus, the molecular mechanics “energy” of a molecule can be described in terms of the total contribution of bonded and non-bonded interactions [29, p. 3] as follows:

$$\begin{gathered}
\mathbf{E}_{total} = \mathbf{E}_{bonded} + \mathbf{E}_{non-bonded} \\
\mathbf{E}_{bonded} = \mathbf{E}_{stretching} + \mathbf{E}_{bending} + \mathbf{E}_{torsion} + \mathbf{E}_{improper} \\
\mathbf{E}_{non-bonded} = \mathbf{E}_{vdw} + \mathbf{E}_{coulombic}
\end{gathered} \tag{2.22}$$

These contributions are illustrated in Fig. 2.10.

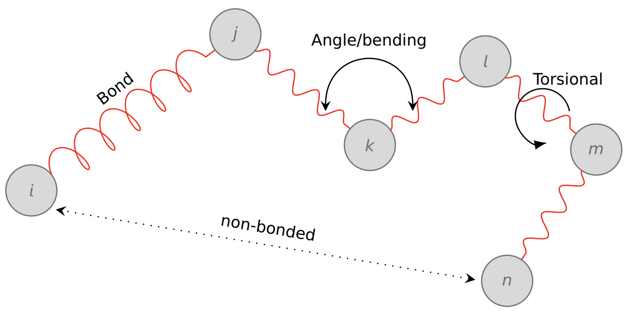


**Figure 2.10: Graphical illustration of interaction forces.**

### Bonded Forces

As presented in Equation 2.22 and Figure 2.10, the bonded potential energy is composed of *stretch*, *bend* (also known as *angle* energy), *dihedral* (*or torsion*) contributions, which have an ideal form, and an *improper dihedral* contribution, which arises from torsion between two different planes, for further details on their definitions and their computation refer to [91, p. 48].

### Non-bonded Interactions

Non-bonded interactions are mainly used to mimic interactions arising from the electronic distributions surrounding different particles. These interactions, as mentioned above, are intermolecular in nature. According to [70, p. 133], these types of interaction are supposed to be governed by quantum mechanics (due to the overlap of the electronic clouds of two interacting particles), however there is no theoretical equation that can describe their interaction and therefore a number of empirical potential functions have been introduced. As given in Equation 2.22, the two components for non-bonded interactions are van der Waals and coulombic (electrostatic energy). Van der Waals' energy, $E_{vdw}$, describes the attraction and repulsion between particles that are not directly bonded [72, p. 34], and represents the non-polar part that is not linked to electrostatic energy. Three empirical functions have been introduced for computing van der Waals' interactions: the *Buckingham*, *Morse*, and *Lennard-Jones (LJ)* potentials. In our study, we focus on the *Lennard-Jones* potential, which is used to calculate the long-range attractive interaction and short-range repulsive interaction between particles. The LJ interaction energy between particles $i$ and $j$ is given by:

$$E_{LJ}(R^{AB}) = \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \tag{2.23}$$

where $A_{ij}$ and $B_{ij}$ are positive constants depending on the types of particles $i$,$j$ (more details about how to determine the values of $A$ and $B$ are given in [47, 72]). The repulsive part of the LJ curve is produced by the $r_{ij}^{-12}$ term, and the attractive part by the $r_{ij}^{-6}$ term. Therefore, this equation can be rewritten as follows:

$$E(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2.24}$$

where $r$ is the separation between the particles, $\sigma$ is the separation at which the potential is zero, and $-\epsilon$ is the minimum potential, which occurs at $r = r_m = 2^{1/6}\sigma$. Thus, for the Lennard-Jones potential the force is:

$$\mathbf{f}_{ij}(r_{ij}) = \frac{12\epsilon}{r_{ij}}\left[\left(\frac{r_m}{r_{ij}}\right)^{12} - \left(\frac{r_m}{r_{ij}}\right)^{6}\right]\hat{\mathbf{r}}_{ij} \tag{2.25}$$

If the particle separation is greater than $r_m$ the particles are attracted, and if the separation is less than $r_m$ they are repelled. For brevity, we write $\mathbf{f}_{ij}(r_{ij})$ as $\mathbf{f}_{ij}$, and since $r_{ij} = r_{ji}$ and $\hat{\mathbf{r}}_{ij} = -\hat{\mathbf{r}}_{ji}$, it follows that $\mathbf{f}_{ij} = -\mathbf{f}_{ji}$, in accordance with Newton's Third Law. This can be used to reduce the number of computations needed to find $\mathbf{F}_i$ for $i = 0, 1, \ldots, n-1$ by about 50%, since when $\mathbf{f}_{ij}$ is calculated and added to $\mathbf{F}_i$ we can also add $-\mathbf{f}_{ij}$ to $\mathbf{F}_j$.

The LJ potential has a number of positive points that motivate its use as part of our study compared to the other non-bonded computations, and these can be summarized as follows:

- Buckingham has limitations in terms of short-range interactions.
- Buckingham overestimates interactions at short distances.
- $r_0$ in the Morse case is larger, which may mean that more particles must be considered in the computation.
- The Morse and Buckingham cases have three parameters as part of the calculation, while LJ employs only two.
- LJ removes the need to compute square roots in finding the distance between particles due to the even powers involved in both terms of the interparticle force computation.
- LJ was found to be the most commonly used function in most of the molecular dynamics software that we have studied.

Finally, the coulombic potential energy (*electrostatic potential energy*), results from an unequal distribution of charges in a molecule. The interaction between two points of charge is defined as follows:

$$E_{coulombic} = \frac{1}{4\pi\epsilon_0\epsilon} \sum_{ij\,pairs} \frac{q_i q_j}{r_{ij}} \tag{2.26}$$

where $q_i$ and $q_j$ are the charges on particles $i$ and $j$, $r_{ij}$ is the distance between them, and $\epsilon$ is the dielectric constant. Different electrostatic force calculations can be based on assigning off-centre positions to the charges, or by considering higher moments, such as dipoles, but they increase the computation complexity.

### 2.4.3 Periodic Boundary Conditions

Periodic boundary conditions (PBC) solve the issue of the infinite number of atoms in the modelled molecular system. The boundaries are imagined as walls surrounding a fixed number of particles, and it is assumed that the physical boundaries of the real system are far enough away to not influence the bulk behaviour. PBCs should take into consideration that a particle should not interact with its own periodic image. This will be the case as the distance between a particle and its mirror image is large enough to be able to ignore their interaction (although, in some models, their electrostatic interaction may be hard to ignore). PCBs must also ensure that the pressure tensor does not contain perturbations that have a wavelength longer than the simulation cell.

A number of simulation cell shapes has been proposed according to the structure of the simulated molecular system. However, most simulations use cubic box boundaries.

### 2.4.4 Neighbour Lists

The Lennard-Jones potential energy computation (see Equation 2.23 ) involves only those particles that are closer than the cut-off distance $r_0$. Particles $i$ and $j$ are neighbours if the distance between them is less than $r_0$. Consequently, the number of particle interactions is reduced in the potential energy computation without compromising its accuracy. However, identifying and managing neighbour particles for each particle introduces new computation and memory storage requirements. In other words, to find the neighbour particles for particle $i$ requires $(N-1)$ comparisons with the remaining particles to find which particles are within the cut-off distance, where $N$ is the total number of particles. Furthermore, particle $i$ must store at least $4/3\pi r_0 \rho$ neighbour particles. In addition, the neighbour list for a particle is required to be modified as a result of particles moving out of the neighbour list of one particle and joining that of another particle.

Verlet [140] introduced a neighbour table as a data structure to store neighbour lists and to tackle the update frequency issue. The neighbour table manages all the particle pairs that are separated by a distance of less than $r_0 + r_s$, where $r_s$ is small "skin" distance. In this approach, updating the particles' neighbour tables is conducted every $h$ time steps. Therefore, between neighbour table updates, the computation uses only particles listed within the neighbour table without performing further comparisons of distance measurements. This computation is valid as long as particles do not move more than a distance $r_s$, and $(r_s - r_0) \lesssim n\bar{v}h$, where $\bar{v}$ is the mean velocity and $n$ is the window size for neighbour table evaluation.

Other methodologies that are used for managing neighbour lists will be addressed in the literature review in Chapter 3 as this topic is an important area for MDS optimization.

## 2.4.5 Thermodynamics

In molecular dynamics simulations, temperature should be held constant, however, thermal energy may be generated due to an irreversible dissipation of mechanical energy. Therefore, a thermostat is needed to remove heat at the same rate at which it is generated. A number of algorithms have been proposed to do this. Most thermostatic controls used in MDS are based on controlling the thermal fluctuation velocities. The

Nosé-Hoover thermostat is commonly used, and is based on the Nosé Hamiltonian enhanced by a coordinate transformation:

$$\dot{\mathbf{r}}_i = \frac{\mathbf{P}_i}{m_i}, \tag{2.27}$$

$$\dot{\mathbf{p}}_i = \mathbf{F}_i - \zeta \mathbf{p}_i, \tag{2.28}$$

$$\dot{\zeta} = \frac{1}{Q}\left[\sum_{i=1}^{N} \frac{\mathbf{p}_i \cdot \mathbf{p}_i}{m_i} - N_f k_B T\right] \tag{2.29}$$

where $\zeta$ is a random Gaussian value with zero mean and variance of $\langle \zeta^2 \rangle = k_B T/M$, with $M = \sum_{i=1}^{N} m_i$ being the total mass of the system, $Q$ is the Nosé mass, $N_f$ is the number of degrees of freedom in an N-body molecular system, $k_B$ is Boltzmann's constant, $T$ is the kinetic temperature, and $N$ is the number of atoms.

### 2.4.6 Time Integration

After computing the force on each particle, the equation of motion is integrated to calculate the new position and velocity of the particles. A number of algorithms have been proposed to perform this integration, such as the leapfrog algorithm [141, p. 99], which is based on calculating position at times $t_0 + n\Delta t$ and velocity at times $t_0 + (n + 1/2)\Delta t$. The leap-frog integration scheme proceeds as follows:

$$\mathbf{v}_i(t_n + \Delta t/2) = \mathbf{v}_i(t_n - \Delta t/2) + m^{-1}\mathbf{F}_i(t_n)\Delta t, \tag{2.30}$$

$$\mathbf{r}_i(t_n + \Delta t) = \mathbf{r}_i(t_n) + \mathbf{v}_i(t_n + \Delta t/2)\Delta t. \tag{2.31}$$

Another well-known integration scheme in MDS is the velocity Verlet algorithm, which is derived from a Taylor series expansion of position $r$ around time step $t$. The method subtracts $r(t-\Delta t)$ from $r(t+\Delta t)$. The velocity Verlet algorithm is a variant of the Verlet approach that does not integrate the velocity, but calculates it from the position using the following finite difference scheme:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{1}{2}a_i(t)(\Delta t)^2, \tag{2.32}$$

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \frac{1}{2}\left[a_i(t) + a_i(t + \Delta t)\right]\Delta t. \tag{2.33}$$

The velocity Verlet scheme has a discretization error of $O(\Delta t^3)$ for the velocity, whereas the discretization error for the standard Verlet scheme with a central difference calculation for the velocity is $O(\Delta t^2)$ [60, p. 38].

## 2.5 Parallel Programming

There are number of scientific phenomena and scenarios that are large in size (in some sense), and so cannot be studied, simulated or solved by using one computer, regardless of its hardware specification and architecture. As the speed of transistors increases so does their power consumption, and some of this power is dissipated as heat, which impacts on the reliability of integrated circuits [108]. This places fundamental limits on the computational power of uni-processor computers. Therefore, it is necessary to find an effective approach of aggregating computing resources to provide a unified computing and memory space, thereby overcoming the limitations of a single hardware resource. Modern processors often contain several power-efficient computing units on one chip, each of which can execute an independent thread of control, and can access the same memory concurrently. Thus, parallelism is becoming a common part of many software products. Each single computing unit is called a *core* and the term *multicore* is used to describe a processor that has several cores [137]. Flynn's taxonomy

[49] defines relations between instruction and data and classifies parallel architectures into various categories: single instruction stream-single data stream (SISD); single instruction stream-multiple-data stream (SIMD); multiple instruction stream-single-data stream (MISD); and multiple-instruction stream-multiple data stream (MIMD). In this section we will focus on the SIMD category, which the CUDA programming model belongs to [42].

## 2.5.1 Memory Access and Process Synchronization

The major challenge in parallel programming is how to exchange data between the different processors, and how to synchronize changes carried out on the data with other processors when they are working together on solving a common problem. Therefore, there are a number of methods to use different machine resources depending on the type of problem to be solved, and which obstacles must be overcome. For example, memory intensive problems may utilize a *shared memory* model and CPU intensive computations, or it may be neccessary to overcome resource limitations using a *distributed systems* model (sometimes referred to as a network model).

### Shared Memory Model

In the shared memory model of parallel computing, which is considered an extension of the basic sequential model, multiple processors have access to a single shared memory. Thus, each processor can execute its own local code and can exchange data with other processors through the shared memory (also referred to as global memory). Each processor is defined by an ID called the *processor ID*. This model can also split into two different approaches: synchronous and asynchronous mode. In synchronous mode, all the processors operate synchronously under the control of a common clock, while in asynchronous mode, each processor operates independently [71].

#### Distributed Memory Model

In the distributed memory model of parallel computing, the processors of the computer are interconnected by a communication network and each processor has its own local memory; there is no shared memory. Each processor can access its local memory easily and quickly [129]. However, if it needs to access the memory of another processor, it will need to communicate with the other processor through message passing. A number of topologies have been proposed for the communication network that connects the nodes of a distributed memory parallel computer. There is no topology that is optimal for all parallel algorithms. However, many high performance computers use a 3D *torus* topology. Synchronization between the different processors depends on the parallel methodology adopted, such as the task farm approach, orchestration by a controller processor, and divide-and-conquer and pipeline algorithms.

#### Hybrid Models

Hybrid programming models that combine shared and distributed memory techniques have also recently become an effective approach to utilizing heterogeneous parallel architectures that contain a mix of processor types. This can be achieved by using multiple programming APIs such as OpenMP or threads packages (such as Pthreads) [27, p. 11] and the Message Passing Interface (MPI), to synchronize the computation with processors within the same node (computer) or external nodes. Detailed examples of OpenMP and MPI programming can be found in references [27] and [109], respectively.

### 2.5.2 GPGPU Architecture and Programming

General purpose GPUs have emerged as the most cost-effective high performance computing platform for certain classes of application. GPUs support the shared memory model within the context of a many-core architecture. GPU architectures focus on instruction throughput and memory bandwidth to leverage the optimisation of execution throughput by adopting massive numbers and levels of threads. Thus, memory access latency can be hidden by using fine-grain parallelism that allows computation to be overlapped with memory accesses. Today, there are various GPU vendors that have their own design and architecture and some of them provide programming interfaces to support their use in general-purpose computing. Open Computing Language (OpenCL) has been proposed as a framework for GPUs, managed by the non-profit consortium Khronos Group. OpenCL is intended for use in heterogeneous GPU applications [78, p. 1]. In this research, we have used Nvidia GPU hardware with Compute Unified Device Architecture (CUDA), a parallel computing platform and application programming interface (API) for the C/C++ programming language.

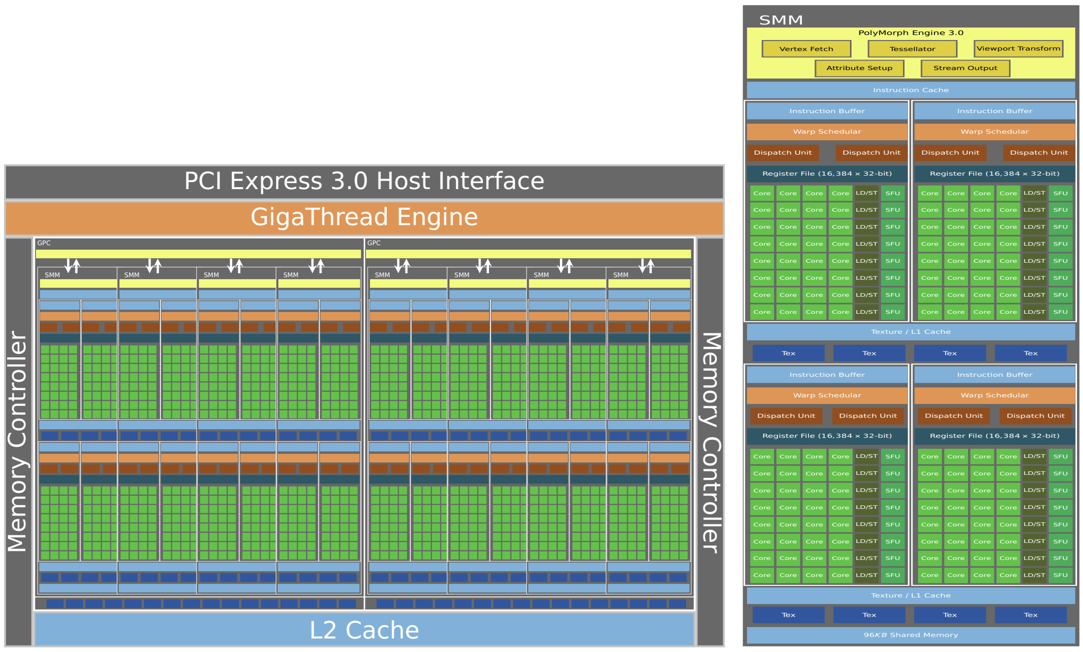
### 2.5.3 GPU General Architecture

To illustrate the typical architecture of a GPU, we are going to use the Maxwell (GM206) GPU. Maxwell is not the latest Nvidia architecture, and was introduced in 2014. Energy efficiency is one of the significant enhancements compared to earlier Nvidia architectures, so that it is nearly twice as efficient as the Kepler GPU architecture[3].

Figure 2.11 illustrates the key components of the GeForce GTX 960 GPU. At a high level, it is partitioned into multiple GPCs (*Graphics Processing Cluster*), where each is composed of several *streaming multiprocessors*, also known as SMMs, and a *Raster Engine*. Each SMM consists of a single *PolyMorph Engine*, and can be seen as a mini-processor responsible for multiple functionality, such as vertex fetching, viewport transformation, handle attribute setup, streams output, and tessellation. Each SMM is made up of multiple streaming processors (SPs), each containing multiple cores (in our case, 32 cores per SP). The SPs support a multithreading execution mechanism for SIMD or SIMT[1] implementation (with a high degree of divergent branching), and is aided by additional *Special Function Units* (SFUs) and *Load/Store Units* (LD/ST).

[1]SIMT=Single Instruction Multiple Threads

SFUs are used to execute transcendental instructions, such as sine, cosine, reciprocal, and square roots. LD/ST units are used to load and store the data required for thread execution, and can perform one load or store operation per clock cycle. Each SP contains 32 cores, 8 LD/ST units, and 8 SFUs, and each has a single *instruction buffer*, *warp scheduler*, and *register file*, and two *dispatch units*. The L1 and texture caches are combined in this architecture into a single unit, known as a *Unified Cache* of size 24 KB per SP (48 KB per SMM or 384 KB in total for all SMMs) [152]. In addition, each SMM has 96 KB of shared memory. All SMMs share a 1 MB L2 cache and a dynamic random access memory (DRAM) controller through a crossbar interconnection network.



(a) (b)

**Figure 2.11: (a) The general GPU architecture of a GeForce GTX 960 (Maxwell GM206). It shows two GPCs, 8 SMMs where there are two in each sub-unit. L2 Cache is shared by all SMMs, and one gigathread engine controls and manages the distribution of computations between SMMs. (b) Shows the components of each of the streaming processors [35, 152] .**

The global memory, also known as *device memory*, is the GPU off-chip main memory. It is the most accessed memory as the data that is copied from the host main memory are stored in this memory. The size varies from one product to another; in our case, the size is 2 GB. The performance of global memory is mainly constrained by two factors: raw memory bandwidth and coalescing degree, which is the extent to which the threads in a warp access consecutive data in a memory transaction. A warp is the smallest unit of scheduling on the GPU.

Local memory is not a dedicated physical memory space but rather a small portion of the global memory. Generally, it complements the functionality of registers, and is used when there are insufficient registers to store the required thread-local variables, especially with large structures or arrays. The compiler may decide to spill registers due to the limited number of registers that the hardware can provide to a kernel, or to spill an array when their index is not a compile-time constant [44]. Register spilling means that instead of being stored in registers data is stored in local or shared memory that is slower to access. Therefore, most GPU architectures cache local memory into L1 and L2 cache. However, for the Maxwell architecture local memory is only cached in the L2 cache.

Texture and constant memory are other memories that reside on the GPU. Constant memory (64 KB) is used to store constant data that is not modified by kernel execution. Constant memory is cached by its own specific cache, known as the constant cache (8 KB per SMM). Similarly, texture memory is part of global memory and is buffered in the unified cache.

### 2.5.4 CUDA Programming

There are a number of languages for performing general purpose computation on GPUs, and these vary depending on the GPU manufacturer, the extent to which heterogeneity is supported, and performance and functionality considerations. OpenCL supports different GPU manufacturers such as AMD, Intel, and NVIDIA. Unfortunately, OpenCL does not support GPU-to-GPU communication, and a GPU can communicate only with the processing node that hosts it and not with other processing nodes. OpenCL supports both data and task-based parallel programming models, ISO C99 with parallelism extensions and the IEEE 754 specification. It can be configured for use on handheld and embedded devices, and interoperates efficiently with OpenGL and other graphics APIs [81].

OpenCL and CUDA [44, 34] provide a low level hardware abstraction and define a framework for parallel programming in which many details of the underlying hardware are exposed. It defines the platform model, memory model, execution model, and programming model. The platform model defines the relation between the CPU and GPU (or *device)*, and how the architecture of the GPU maps to the API's hardware abstraction, such as defining how the streaming multiprocessor ("computing unit" in OpenCL terminology) is divided into one or more CUDA cores ("processing elements" in OpenCL).

The memory model describes the structure, contents, behaviour and management of the memory. The memory model takes into consideration the definition of memory regions, memory objects and their implementations, virtual shared memory and consistency rules that define access, atomic/fence operations, and synchronization relationships. The previous subsection described different memory spaces (device memory, shared memory, constant memory). Figure 2.12 shows how the CUDA thread hierarchy defines the access level per thread, block, and grid [34],[32, Chapter 5]. Thus, global, constant, and texture memory are shared among threads within a grid. Shared memory is shared among threads in the same thread block. Local memory and registers are allocated individually and privately per thread.

The execution model [44, Chapter 4] defines the execution of the kernel that takes place on a device, and the host program that executes on the host side (CPU). Additionally, it controls the execution of commands, data transfer between the different memory levels, and synchronization. The device code that is executed as a function on the GPU side (known as a *kernel*) is first translated into an intermediate device language called

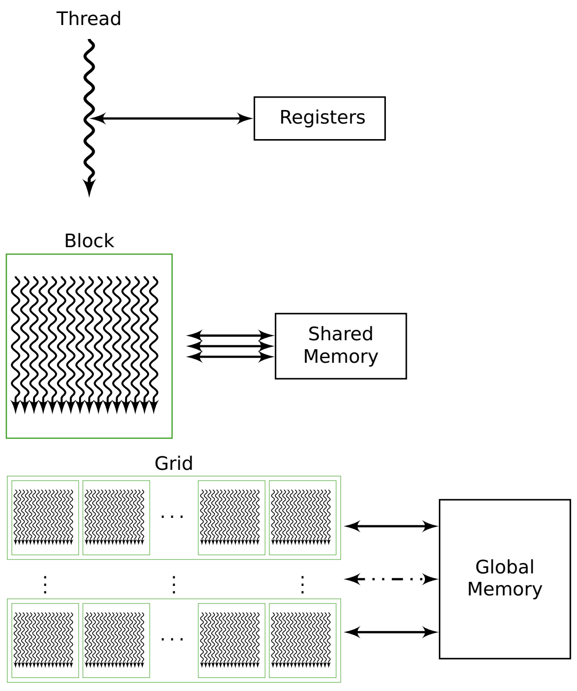


**Figure 2.12: Memory hierarchy sharing level with threads, blocks and grid, from top left to right.**

PTX (Parallel Thread Execution) and then converted into a binary code which can be executed on the processing cores. The CPU code may be compiled using a standard compiler. The Nvidia compiler is able to link both the CPU and GPU binary code, which results in the ability to access the methods or share the data between them, (for more details refer to [33]).

The programming model facilitates the optimal use of the GPU when writing a kernel function, thus maximizing the concurrency of thread execution. In order to optimize the usage of GPUs, the data are decomposed into small chunks of data (blocks) each of which is processed by a block of threads, such that each datum in a block is processed by one thread (this is generally best practice, although each thread can process more than one datum). Therefore, as highlighted in the thread hierarchy model, a grid of threads is composed of a number of blocks where each block consists of a group of threads (recommended to be a multiple of the warp size which is usually 32 threads). The total number of threads provided by the hardware specification constrains the maximum number of active warps that can be scheduled simultaneously. The scope of a thread's access to different memory levels guides developers to cause the data that are required to be shared at the block level to be stored in shared memory, while those data required to be shared at the grid level are stored in the global memory. In addition, the programming model guide the developers to be cautious in atomically accessing memory and in synchronizing the threads within a block. GPUs also support streaming execution, which is useful if the number of threads executing a kernel is small, such as in an implementation of the data reduction problem.

CUDA has been found to perform faster than OpenCL due to its high compatibility with Nvidia devices[79], and because of this OpenCL code compilation is done at runtime. CUDA offers several APIs for programming, such as the runtime API, Thrust API, and drivers API. In addition, these APIs can be called from a number of programming languages, such as Python, Java, FORTRAN, and many others. Therefore, in the research presented in this dissertation we adopted the CUDA programming language, rather than OpenCL, as performance is one of dominant factors in our study. In addition, the hardware available to us are Nvidia GPUs. Thus, CUDA guarantees high GPU compatibility as they are both provided by the same vendor.

## 2.6 Optimization and Performance Analysis

This section addresses the factors that contribute to optimal performance on GPUs, depending on the hardware and the nature of the application implemented. Then, we introduce the performance analysis methods that are adopted in this work.

### 2.6.1 GPUs Optimizations

Due to the massive parallel architecture of GPUs and the complexity of their memory hierarchy there are a number of factors that affect the utilization of GPUs, especially with an irregular application that has a complex dependency between the data and their manipulations. GPU performance-critical factors can be defined according to three strategies. The first is to maximize the parallel executions which means to maximize the utilization of the available cores to use all the hardware resources. However, each core executes one of the threads and might face a number of constraints on the availability of hardware resources, such as the number of registers, data dependencies, and bank conflicts in the shared memory. The second strategy is related to optimizing memory throughput, by reducing the data transactions with low bandwidth memory, such as copying the data between CPU and GPU or fetching data from global memory or another memory space within the device memory (for example, local, texture, or constant memories that are not cached or have high miss rates in accessing their corresponding cache). If the data required to be shared and synchronized are in the same block level, then it is highly recommended to store these data in shared memory, because this enables both write and read with high bandwidth (normally equal to that of the constant cache). However, shared memory space is limited and requires careful access, otherwise it may cause a number of bank conflicts or may be subject to thread race condition issues. As the nature of parallel programming is to deal with massive data, most data is available in the global memory, as the use of the on-chip caches is constrained by their limited size. Therefore, performing coalescent memory accesses by the threads in a warp enhances performance significantly. However, it is limited by the type and size of the data that are accessed, the transaction line size (128-byte cache line), the computation capability, and the number of memory transactions that can be executed per cycle. A third strategy is to maximize instruction throughput, which can be accomplished by a trade-off between performance and computation precision, minimizing warp divergence due to flow control instructions in condition statements (such as if or switch statements), and loop control conditions. Thread divergence, where the threads in a warp follow different execution paths, significantly impacts GPU performance by serializing instructions and increasing the number of instructions per warp. In addition, this issue has a high potential for creating scattered memory accesses among the threads within a warp.

A high level of spatial locality, whether data are stored in one of the caches or in global memory, results in highly coalesced data access. Thus, the number of transactions per access is minimized, especially if the data size and the cache line to be fetched are aligned in memory. As a result, those factors offer significant potential for improvement in the overall execution time of an application.

### 2.6.2 Performance Analysis

Evaluating the performance of an application often leads to significant outcomes in terms of enhancing application development, and may provide valuable insights related to hardware design and optimization. In order to achieve the objectives of this study, we are going to build a cache model to analyse the relation between the cache model and the various data orderings that are adopted in this work. Then, we analyse the execution time of these orderings for both CPU and GPU versions to measure the application throughput and the ratio of the execution time taken by the CPU and GPU. MiniMD collects the execution time of each module in its implementation, such as the neighbour list time, communication time (for an MPI implementation), and the force computation time. In addition, we will use the NVidia profiler, which provides very broad information related to kernel execution on GPU platforms. NVidia profiling will help us to understand the constraints on the execution of instructions and on memory accesses. In addition, it will help us to evaluate the data locality properties by studying various memory hierarchy components with our orderings. From a hardware perspective, we conduct these experiments with two different GPUs: a GeForce GTX 960 and Tesla P100. More details of the hardware and the analysis carried out are provided in Chapter 6.

The accuracy of the applications is compared with the original version and the provided reference data that is part of the MiniMD package. Early GPUs had limitations in terms of double-precision computation, however, with most of the recent GPU hardware, double-precision computation is supported.

## 2.7 Summary

This chapter has provided a background for all the components of our research: data locality, space-filling curves, the methods used to generate the indices of data orderings, molecular dynamics simulation, GPU architecture and programming, as well as a brief discussion of optimization and performance analysis. In terms of data locality, we have explained why, in general, a memory hierarchy and its access policies has a significant impact on an application's performance. We have defined space-filling curves and have shown how they can be geometrically and arithmetically represented. This chapter also introduced molecular dynamics simulations, and the various types of inter-particle forces was outlined. In the section on MDS, we focused on Lennard-Jones force computations and the reasons for adopting this force model compared to the other available models. We used one of our devices, an Nvidia GeForce GTX 960, as an example of a GPU architecture and described the specification of this hardware. In addition, we highlighted in summary, the CUDA programming language for NVidia GPU hardware, and we explained why CUDA was selected in preference to OpenCL. Finally, issues relating to optimization and performance that must be taken into consideration in the implementation and analysis phase of our research was provided. This will help in understanding the implementation and analysis chapters.

*Chapter 3*

# Literature Review

This chapter presents a literature review for all parts of our study. We shall explore previous work that has been done to optimize data locality in general, either by instruction ordering or data ordering. Then we explore how space filling curves, in particular the Hilbert and Morton curves, are generated and used to improve data locality in various applications. In the existing literature we have tried to find any work that is similar to the implementation and analysis that we have carried out in order to make a comparison and to analyse the potential for enhancing our approach. Finally, this literature review will present various relevant research studies that have been carried out in the area of molecular dynamics simulation. Different perspectives will be considered, such as general optimization for any platform, and then specific implementations that have been conducted on GPUs. Finally we check if any of the existing research addresses data locality issues.

## 3.1 Data Locality

In recent years, processor speeds have increased faster than memory speeds, so optimization for efficient memory hierarchy use has become a very important research area. There are a number of literature studies exploring data locality issues on CPU platforms that can be classified as instruction reordering, data reordering, or both. Instruction ordering can be achieved by reordering the algorithm instructions in a way that allows consecutive instructions to reuse data while they are still in cache or registers. Loop transformation is an example of instruction reordering that reduces the number of intervening iterations. Blocking [54] or tiling [149] have been used intensively in numerical algorithms as a mechanism for instruction reordering. Most of the tiling research focuses on loop transformation theory and proposes a number of approaches to maximize multiprocessor parallelism.

### 3.1.1 Loop Transformation

Loop transformation is a technique for compiler optimization and automatic program parallelization. The literature mainly explores data dependency in respect to the memory hierarchy and its architecture. Most of the literature has focused on array-based optimization of nested loops. There are a number of methods proposed which primarily rely on reordering the nested loops or modifying the loop indices in association with their initial and upper bound constraints. Examples of these methods are as follows:

- *Loop fusion*, which merges two loops into one loop as shown in Algorithms 3.1 and 3.2. This can be applied when the statements in both loops have the same order of execution, assuming there is no dependency between the statements $S_1$ and $S_2$. Examining the data dependencies and memory requirements helps to decide which loops and statements can be merged into one loop. Merging several loops should ensure that memory is not displaced from registers and/or cache. On the other hand, *loop distribution*, which is an opposite approach to loop fusion, splits the loop body into multiple loops in order to perform an additional loop transformation method on one or more of the loops. This can also expose parallelism, and reduce register use.

**Algorithm 3.1:** Before Loop Fusion

```
1 for $i$ = 1 to $N$ do
2     $S_1$
3 end
4 for $j$ = 1 to $N$ do
5     $S_2$
6 end
```

**Algorithm 3.2:** After Loop Fusion

```
1 for $i$ = 1 to $N$ do
2     $S_1$
3     $S_2$
4 end
```

- *Loop interchange (permutation)*, which is achieved by changing the order of nested loops. The approach changes the execution order of blocks of iterations, with the aim of optimizing memory access in the inner loop(s). Hence, data accessed in the innermost loop has a higher chance of being available in the cache or in a register, and are accessed in sequential order, while data accessed in the outermost loop are less likely to be found in cache or registers. For example, in matrix multiplication of size $N \times N$, Algorithm 3.3 can be transformed into Algorithm 3.4. The reason for doing this is, if the matrix data are stored in row-major order, each iteration of the $k$ loop requires data to be fetched from the memory into the cache as one row of matrix $b$ is larger than the cache size. In contrast, in Algorithm 3.4, all the matrices are accessed by row so there is no address jump as the data of the matrices ($a, b$ or $c$) is accessed along a row. The iteration over $j$ is used to advance the column index for matrices $b$ and $c$. Therefore, in the innermost loop, most data accesses have a high chance of being found in cache because the rows of the matrices are cached. This will reduce the number of accesses to main memory and reduce the cache miss rate. With respect to parallelism, the outermost loop could be handled by different processors and the innermost loops could be executed by the corresponding processor.

**Algorithm 3.3:** Before Loop Interchange

```
for $i = 1$ to $N$ do
    for $j = 1$ to $N$ do
        for $k = 1$ to $N$ do
            $c[i, j] = c[i, j] + a[i, k] * b[k, j]$
        end
    end
end
```

**Algorithm 3.4:** After Loop Interchange

```
for $i = 1$ to $N$ do
    for $k = 1$ to $N$ do
        for $j = 1$ to $N$ do
            $c[i, j] = c[i, j] + a[i, k] * b[k, j]$
        end
    end
end
```

- *Loop Skewing*, which is performed by changing the iterator bounds for at least one of the loop iteration spaces so the shape of the nested loop is changed from a rectangular to a trapezoidal access pattern. This method is normally used to allow other transformation methods to be applied directly to deal with dependencies and the distance between two nested loops. It can also be used for loop normalization by resetting the low loop bound to one (or zero).

**Algorithm 3.5:** Before Loop Skewing

```
for $i = 4$ to $N$ do
    for $j = i + 2$ to $N$ do
        $a[i, j] = a[i - 2, j] + b[i, j]$
    end
end
```

**Algorithm 3.6:** After Loop Skewing

```
for $i = 1$ to $N - 3$ do
    for $j = 1$ to $N - i - 1$ do
        $a[i+3, j+i+4] = a[i+1, j+i+4] + b[i+3, j+i+4]$
    end
end
```

- *Strip mining*, which fragments a large loop into smaller segments or strips. A similar technique is called *unrolling* in which, rather than forming a new loop, the instructions of the existing loop body are replicated. These chunks of computations should be aligned with the cache size to optimize temporal and spatial locality. Using the above example, Algorithm 3.4 can be strip mined by using a strip width *SW* to produce Algorithm 3.7. The transformation does not change the loop body instructions although it introduces two additional loops which are called control loops. Strip mining is the basis for what is known as partition space tiling or blocking.

**Algorithm 3.7:** After Strip Mining

```
for $kk=1$ to $N$ by $SW$ do
    for $jj=1$ to $N$ by $SW$ do
        for $i = 1$ to $N$ do
            for $k = kk$ to $min(kk + SW - 1, N)$ do
                for $j = jj$ to $min(jj + SW - 1, N)$ do
                    $c[i, j] = c[i, j] + a[i, k] * b[k, j]$
                end
            end
        end
    end
end
```

There are many other methods and combinations of these techniques explored in the literature. Performing a transformation is not an easy task as it is necessary to understand the relation between the iteration space and the data dependencies while accessing these data through the memory hierarchy to decide the correctness of the applied transformation. For example, in the loop skewing Algorithms 3.5 and 3.6, which are similar to an example in [148], the $i = 4$ and $j = 8$ iteration produces the value to the matrix $a[4, 8]$ which, in turn, is used in computing the value of $a[6, 8]$ when the iterator values are $i = 6$ and $j = 8$. Using the transformation Algorithm 3.6, matrix value $a[4, 8]$ is obtained when $i = 1$ and $j = 3$, and that value is used to assign a value to $a[6, 8]$ when the iterator values for $i$ and $j$ are 3 and 1, respectively. Consequently, the dependency distances before and after the transformation are $(-2, 0)$ and $(-2, 2)$ which can be presented by dependency notation vectors as $(<, 0)$ and $(<, >)$, respectively. As the direction vector is $(<, >)$, this prevents loop interchange, if it is required for any reason. Michael Wolfe has published a number of code transformation techniques for implementing parallelism and data locality optimization [88, 146, 147, 149, 151]. Most of this research has investigated the association between data dependency and parallel or pipelined execution. Therefore, a number of techniques resulting from this research, have been proposed, such as blocking or tiling, which can be used at different levels of the memory hierarchy, such as main memory, caches and registers [145]. Tiling reorders the loop executions and reduces the number of intervening nested loops which allows data to be reused while still in cache or in a register, and hence reduces memory access time. Tiling may also be used to initially reorder the data to be in consecutive order, so they are processed in a way that is aligned with the loop order to achieve optimal performance. Achieving tiling involves going through a number of primitive transformations, especially in multiple dimensional tiling. This can require applying a combination of transformation techniques, as in the research conducted in [149], which proposes transforming the iteration space by using strip mining and loop interchange.

Similarly, Irigoin [106] has focused on defining data dependencies based on the dependence cone. This work claims that a more accurate distance and direction summary can be achieved, compared to the dependency vector method, due to the representation reflected by the linear system model. This approach is very well suited to the need for global parallelization as conducted by the hyperplane or supernode partitioning method [69]. Moreover, as the loops are constructed by variables and constraints, they could be expressed as affine transformations on integer sets defined by polyhedra, and the new, transformed loops bounds can be computed using Fourier's pairwise elimination method [13].

A number of initiatives focus on translating an annotated C program into CUDA or OpenCL for execution on a GPU; for example, as carried out by the Portland Group [150]. This involves the adoption of loop transformations techniques and using directives in parallel regions for GPU codes, in order to achieve high portability between different GPU programming languages. Automatic transformation of nested loops for GPU execution has been discussed in a number of studies, such as in [40, 138]. These carry out the transformation based on uniform dependencies and balanced tiling. A limited number of papers cover the transformation reduction operations and atomic data access on GPUs.

### 3.1.2 Drawbacks of Loop Transformation

Loop restructuring techniques focus on compiler optimizations, so programmers may not be able to discover which of these techniques have been applied to their code. It would be preferable to avoid unnecessarily restructuring loops by hand, as may happen with commercial compilers that do not reveal their compiler's code. Compilers could change the instruction order and modify the implementation according to built-in optimization rules. The efficiency of loop restructuring techniques is generally lower for just a single loop, especially in a parallel environment where partition spaces are reduced by at least one loop compared to the original sequential implementation. Most loop restructuring methods can be applied with uniform data dependency. In addition, a considerable amount of research focuses on loop transformation despite its difference to the data dependency representation approach. However, critical sections, atomic modification, and reduction issues, which are part of the partition space, have been less fully explored with respect to loop transformation theories.

### 3.1.3 Data Order

A second approach to optimal use of the memory hierarchy is to order the data within an array, rather than ordering the execution of the instructions, as explained in Subsection 3.1.1. The data ordering could be applied to a fundamental data structure, such as an array, or based on more complex data structures. A third approach, used in irregular applications, uses an index array to access data on different data structures, such as an array, and has the form $data[index[i]]$.

The canonical order, which is also referred to as a linear order, is mapped to the memory in *row-major* or *column-major* order, according to the language and compiler used. Data are sorted in this order according to the nature of the application's dominant task. For example, if searching and retrieving a datum from an array is the most significant and frequently-executed function, then the data are ordered based on their index values. Thus, there are various algorithms related to sorting which optimize data access, such as quicksort [24, 97], bitonic merge sort [113] and radix sort [67, 155]. Significant efforts have been made to implement these algorithms on GPU platforms. For example, Satish et al. [127] has implemented a radix sort algorithm on a GPU and their implementation was able to sort more than 100 million keys in a second. A speedup of about 33.4% was achieved in [67] by empirically searching for the optimal alignment between the algorithmic parameters and the architectural features of the hardware. Furthermore, Ha et al. [57, 58] have reduced the number of comparisons between the elements under evaluation and the next data elements in order to enhance coalesced data access. In addition, shared memory is used for sum reduction instead of using the global memory, which results in only one access to write the block summation.

A non-linear data structure, such as a tree, can be modified to optimize access to its data or to enhance locality. As an example, various types of trees have been proposed, such as B-trees, quad trees, k-d trees, and AVL-trees [100]. These algorithms have been extensively used in image processing, database management, data mining [133], operating systems, networking services, and other applications demanding expeditious search algorithms to allocate and access services or data. Most of these algorithms were originally designed to divide the data into partitions or blocks that can be fetched efficiently into the higher levels of memory. Normally, these partitions have a defined affinity associated with them to fulfil the objectives for which the data structures were created. In addition, they may provide other functionalities, such as insertion, deletion and tree balancing (in the case of an AVL-tree). Although these algorithms maintain data structure coherence, they increase the overhead of computations. Most of the literature assesses these algorithms based on their computational complexity and memory usage. Investigation of these algorithms from a spatial and temporal data locality perspective has not been afforded much consideration.

Most tree-based sort and search algorithms process large amounts of data, so GPU platforms could accelerate these algorithms, especially if they contain many independent calculations or tasks. However, mapping the tree structure into the memory hierarchy of a GPU is a complex task. For example, inserting an element into a tree is not well-suited for a GPU due to the difficulty of ensuring coalesced memory access [77]. Kim et al. [82] address these issues by restructuring the tree to be aligned with the hardware architecture of the GPU. They proposed the FAST algorithm which associates page size and cache line size with the arrangement of data elements in the context of a SIMD programming model. They have executed their algorithm on a GTX 280 GPU and were able to process $85M$ search queries per second, which is around 1.7 times faster than any previously reported performance.

In contrast to the row-major and column-major orderings, other orderings may make use of a combination of different sorting algorithms and various data structures. This type of non-linear ordering is mostly used when the data have a combination of variables to be sorted over, such as for a composite key (in database terminology). Also, non-linear ordering is adopted for algorithms that are limited to a single data type, such as integers. Therefore, the keys are normally sorted in a separate data structure and the associated data may be optionally sorted according to the compromise between the overhead to store the data and the latency in accessing the data. In this category, we may identify hashing, space-filling curves, and any other algorithms that take the form of $data[f(x)]$, where $f(x)$ is a mapping function applied to the data. Hashing is commonly used in image processing for image comparison and matching. It overcomes the limitation of float hashing by using its binary representation and string values by using the image's integer representation. Hashing supports injective mapping, where a distinct value can be mapped into multiple sets/buckets without a hash value collision. Hence it can be used for range search [121]. The first GPU implementation of a perfect hash algorithm [37] was conducted by Lefebvre and Hoppe [90]. A composite of the perfect hashing and cuckoo hashing approach [110, 115] has been carried out by Alcantara et al. [7]. Their results were compared with the outcomes of the GPU radix sort from Satish et al. [127] and were found to have a similar construction time while performing better in terms of lookup time. However, the composite hashing approach uses about 40% more memory. In addition, a comparison with their own implementation of a perfect hash approach [37] shows comparable lookup times, and is much faster in terms of construction time.

One of the other alternatives for this type of ordering uses *space-filling curves* (SFC) and literature pertaining to this is covered in the next section.

## 3.2 Space Filling Curve Algorithms

In the late 19th century, Peano proposed a continuous mapping of points within the unit interval $[0, 1]$ onto a subset of the unit square in $\mathbb{R}^d$. In two dimensions this can be viewed as projecting coordinate points arranged in a 2D $2^n \times 2^n$ grid onto one dimension. This is an example of a space filling curve (SFC). Sagan [18, 123] has presented an extensive study that is recommended for understanding space filling curves and their applications. The study of space filling curves is an active field of research in various application areas. For instance, the literature on Hilbert curve generation covers the following approaches:

- Using a predefined initial value or a computation of a preceding state value, a state diagram, or a predefined table of values [19, 89, 156]. The storage requirement of this approach rises exponentially with the number of dimensions. Similarly, tables of predefined values may be used to encode or map a point in two dimensions onto the Hilbert value, as in [48, 94].

- Using only computation, which generates the Hilbert curve without using any predefined values, as in [21, 22]. This approach enables the mapping of any arbitrary ordinal point to its location on the Hilbert curve. In this approach, different implementation strategies have been proposed to find the optimal performance for generating the Hilbert curve, which can be classified into algorithmic and bit-manipulation techniques. The algorithmic computation has been implemented using recursive and iterative methods, and more recently parallel implementations have been developed.

### 3.2.1 Hilbert Order

A comprehensive study of 3-dimensional Hilbert curves has been conducted by Haverkort [63, 64, 122]. Haverkort's work [63] examines a number of Hilbert curves and how their various paths can be generated. According to Haverkort, there are a number of curves that may have better data locality than the general Hilbert curve. This has also been studied in earlier literature. There is a limited amount of research focusing on multi-dimensional encoding/decoding of Hilbert curves. Lawder [89] proposed inverse mapping from n-dimensions and improved the encoding procedures suggested by Butz [21, 22]. Chen et al. [28] proposed forward and reverse mappings based on a replication process of the Hilbert square matrix rather than the bit-processing technique as proposed by Fisher [48], which is supported by a look-up table. Liu and Schark [48] used rotation matrices and vector functions in their 3-dimensional Hilbert encoding/decoding approach.

The Hilbert curve has been adopted in various applications as an approach to compress multiple keys, weights, or data in order to search for, or store, the data. For example, in database management where multi-dimensional points have to be sorted into a 1-dimensional structure [43]. In addition, the Hilbert curve has been applied widely in other application areas, based on the work conducted by Zhang [156], where it is used to perform a fast scan of an arbitrarily-sized cuboid region with high computational efficiency. The Hilbert ordering has been used in solving partial differential equations as carried out by Haase et al. [59], in which a fractal storage data structure is used to represent the sparse matrix in a matrix-vector multiplication. Their approach improves the performance by about 15-20% compared to the compressed row storage format. The implementation was conducted on a single CPU platform and compared with two different machines having L2 cache capacities of 256KB and 1MB. In image processing, McCool et al. [99] have used Hilbert ordering to divide and scan polygon fragments through the edge equation evaluation approach to determine if all the corners are outside the region, in which case it can be skipped and the next block on the Hilbert path is processed; otherwise the block is subdivided into quarters (octants in 3D). These quarters are then processed recursively, performing corner evaluations until the recursion is terminated once the desired fragment level is reached. However, this research was limited to a single polygon and implementing the method on specific hardware results in overhead from hierarchical searching. Sastry et al. [126] used Hilbert curves to order mesh vertices and elements to enhance cache utilization in unstructured meshes when solving a partial differential equation using the finite element method. This method recursively subdivides a tetrahedral element into eight octants until all the elements are covered in at least one of the octants, which are in Hilbert curve order. This approach shows better cache utilization compared to the sparse-matrix partition and computation algorithms that are normally used in such problems. A shared memory implementation using OpenMP was carried out on a 48-core multiprocessor system containing four AMD Opteron 6174 processors connected by AMD Hypertransport links. This reduced the time required to assemble the stiffness matrix and solve the resulting linear system by about 20%.

Feng et al. [46] generated a 2-dimensional Hilbert curve on a GPU using a matrix iteration method and state diagram approach. This method compares the generated Hilbert values using a built-in special function with matrix multiplications and rotations. The former outperforms the latter when implemented on an NVIDIA GeForce GTX 480 GPU. Feng et al. [45] have investigated GPU performance of 2-dimensional Hilbert curve generation algorithms that are based on a block matrix iteration method and a state diagram method. Hilbert order has been used in GPU applications, such for information visualization [52]. Unfortunately, this research does not investigate the impact of the Hilbert ordering compared with linear ordering.

### 3.2.2 Morton Order

The literature on Morton orderings has been reviewed by Sagan [123]. The Morton curve is simpler than the Hilbert curve and it changes from one dimension to another every two steps along the path. However, the nodes at the end of a segment and the first node of the following segment are not adjacent, as illustrated in Figure 2.3 (for clarity, see $m_3$), so there are long lines that link nodes between the four largest segments. Morton encoding and decoding algorithms have been studied by Raman and Wise [120] and are used to divide matrices into blocks for optimal use of the memory hierarchy. Morton order has been used to construct linear quad-trees or to restructure matrices into one-dimensional arrays [68, 131, 144].

Most of the literature on Morton ordering uses it to partition matrices into blocks or tiles for mathematical applications. Athanasaki and Koziris [16] restructured the memory layout of multi-dimensional arrays using binary mask operations in order to improve cache use in tiled algorithms. In this study, some of the applications used in their analysis were based on matrix multiplication, and others were based on data that can be split into blocks that can be processed independently of other blocks. Similarly, Lorton and Wise [95] analysed the advantages of using Morton and hybrid Morton order compared with row-major order. Matrices are transformed into Hilbert order by the OPIE compiler, [51] and the BLAS library [41] was used in executing on Opteron and Itanium hardware. The results for block matrix multiplication (defined as a 6-loop algorithm in their work) shows that Morton ordering performs twice as fast as row-major ordering for a pure C implementation due to the reduced number of TLB misses in the Morton case. The comparison also considered miss hit rates for L1 and L2 cache, as well as TLB and page faults for the different orderings on both platforms. Morton ordering exhibits better performance compared with other orderings, and a hybrid Morton order shows excellent memory bandwidth for block recursive matrix multiplication. Additional research has been carried out on Morton ordering, especially with multi-dimensional matrices in [16, 124, 142] to map the multi-dimensional iteration indices to the relevant data in linear memory.

Morton order is widely used in information indexing and sorting in big data and database research. One of the main streams of research it contributes to is in searching for near neighbours because the different dimension keys are used as factors for data classification. Therefore, the row data are initially sorted according to their keys, and then their group/block is sorted according to Morton order as in [83]. In addition, this order reduces the computation complexity to that of approximate neighbour search algorithms, especially for graph construction and geometry applications. For example, Connor and Kumar [31] extended the Shift-Shuffle-Sort method [26], which was limited only to integers, to support float values. In this work, the input points are sorted using the Morton order, which is also used to assess the partial solution validity by comparing if the box boundary of the solution found lies within the lower and upper bound of the Morton block. It was found that the cache efficiency for Morton ordering enhances the algorithm scalability to permit a larger size of point sets. The study shows a lower cache miss rate, and less memory consumption, for graph construction compared to the ANN library [104], which was modified for fair comparison. Nocentino and Rhodes [107] showed that Morton order performs better than linear order when segmenting regions for volume rendering. In addition, the Morton order reduces the number of memory transactions. The Morton indexes are generated on a GPU as in [120] and implemented as a kernel function in order to identify the block that each thread should access and process. The study did not include any analysis of the memory accesses. As the algorithm does not require any neighbouring information, it improves coalesced access for caches where cache lines are aligned with the warp size.

## 3.3 Molecular Dynamic Simulations Optimizations

Most optimizations of molecular dynamics simulations, especially for non-bonded interactions, have focused on a combination of two approaches. The first minimizes the neighbour list so it is closer to the actual interaction list. This reduces the number of distance computations between particles and lessens the number of comparisons with cut-off radius ($r_c$) as shown in Figure 3.1a. The second approach is related to how data is represented, generated and accessed in the neighbour list. Of course, in parallel computation, additional optimizations are introduced to match the execution of the application with the hardware architecture and capabilities.

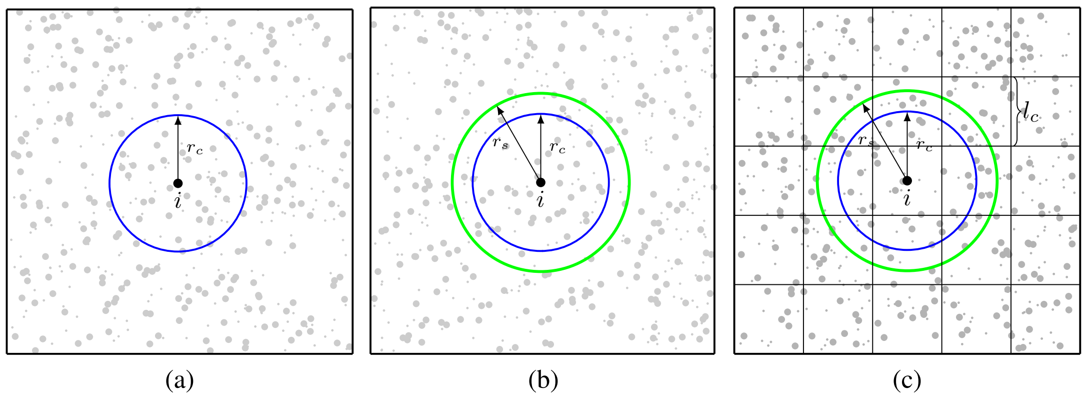


**Figure 3.1: (a) shows the neighbouring particles that are within the cut-off distance $r_c$. The neighbour list must be modified every time step as a particle might move across the perimeter of the circle; (b) shows the search radius (skin radius) as introduced by Verlet; and (c) shows the linked-cells obtained by dividing the simulation box cells with edges of length $l_c = r_c$ .**

The force computation for non-bonded interactions has complexity $O(n^2)$ using brute force computation. However, in the Lennard-Jones case, the interaction force between two particles, $\mathbf{f}_{ij}$, that are a distance larger than the cut-off length apart ($r_{ij} > r_c$) is not computed (set equal to 0), as shown in Figure 3.2. Computing the distance and comparing it with the cut-off distance still has $O(n^2)$ complexity. To exploit the LJ property, Verlet proposed in [140] a bookkeeping device known as the Verlet table and a "skin" depth ($r_s$) added to the cut-off distance. The value of $r_s$ should be set large enough to guarantee the accuracy of the neighbour table, but small enough to make the table size manageable. Sutmann and Stegailov [134] mathematically modelled the values that should be assigned to $r_s$ and the time to reconstruct the Verlet table, which is influenced by the particles' velocity and the simulation time step $\delta t$ .

The second conventional method to reduce the number of distance and force computations is known as the linked cell method, which partitions the simulation domain into cells [10], as in Figure 3.1c for a 2-dimensional domain. The cell edge length is set equal to $r_c$, and so to build the neighbour list for a particle $i$ requires all particles to be examined that are in the same cell as particle $i$ and the other 8 neighbouring cells (26 in three dimensions), subject to the application of periodic boundary conditions. The neighbour list and link cell algorithms each have their own advantages and disadvantages. While the linked cell algorithm improves the construction of the neighbour list, on the other hand, it increases the number of particle pairs that are checked to see if $|r_i - r_j| < r_c$ in the force computation. Therefore, there is an inverse relationship between the cell edge length and the time interval after which the neighbour list must be rebuilt. That is to say, if the edge length is too small, the neighbour lists will have to be rebuilt more often. In addition, reducing the cell edge size increases the number of cells involved in the force computation for a particle and consequently increases the memory requirement. The impact of the size of $l_c$ was analysed in detail in [134] and [98] in which an approach was also defined to find the neighbour cells for each cell at the beginning of the simulation, based on adding the relative cell index offsets to the cell grid index.

A comparison between the two algorithms has been conducted by Li et al. [92], which focuses on simulating the gravity-driven collisions in a granular pile consisting of multi-diameter particles. It was found that the Verlet table simulation time was less sensitive to variations in the search radius, $r_s$, than to changes in the reconstruction interval time. However, for the linked cell algorithm, using a longer reconstruction interval time results in wide variations in the simulation time. The memory access pattern shows a zigzag path, and overly long update times cause the simulation to crash as

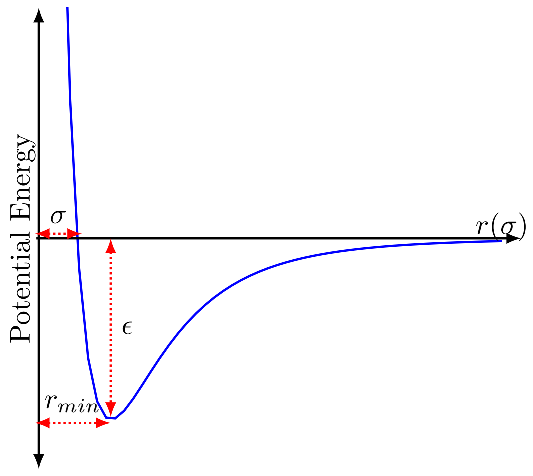


**Figure 3.2: Example of LJ curve generated with $\epsilon = 157$, $\sigma = 3.9$ and incremental time $0.124$. $r_{min}$ is the distance where the potential reaches the equilibrium position of the two particles.**

the neighbour lists become out-of-date. In general, although the Verlet table algorithm is more efficient in simulating systems that have a small number of particles with low mobility, the linked cell approach is better for large systems with high particle mobility.

### 3.3.1 MDS Optimization Based on Reducing the Neighbour List Size and Update Frequency

Yao et al. [154] improved the neighbour list reconstruction interval for the linked cell algorithm by defining a flag (called the "dirty flag") per cell. If a particle's accumulative displacement is larger than the cell edge length, $l_c$, the cell flag will set to true to indicate that the neighbour list of the corresponding cell must be updated. After the reconstruction is completed, the dirty flag is set back to false. The proposed algorithm reduces the total number of cells involved in neighbour list reconstruction. In addition, partial updates and data ordering are introduced into the improved algorithm. The results show that in both cases (single and dual-processor), the improved algorithm outperforms the conventional Verlet and linked cell algorithm, especially for large simulation sizes. Although data ordering was adopted, they did not give any performance analysis related to the cache hit rate, and there was no evidence regarding whether the boosted performance was due to the partial updates or to the data re-ordering.

Gonnet has performed a number of research studies to enhance the conventional neighbour list algorithm. In [55], the size of the neighbour lists was reduced by projecting particle positions in neighbouring cells onto the normalized vector $\vec{r}$ extending between the cell centres. This projection is carried out using the dot-product definition ($\vec{a}.\vec{b} = |\vec{a}||\vec{b}|\cos(\theta)$, where $a$ is the vector between particle $i$ and particle $j$, and $b$ is the normalized vector $\vec{r}/|\vec{r}|$). In addition, the particles in the neighbouring cell are shifted by subtracting the length of the cell edge from their position. Important issues in this research are the overhead cost of projection and the sorting of the neighbour lists. The complexity of the sorting algorithms led Welling and Germano [143] to replace the sorting approach with optimal sorting networks [20], which extended the number of particles involved in each linked cell list from 10 to 16. However, the overhead cost of the proposed algorithm compared to the overall cost was not provided. Ignoring the order of the cells in building the interaction lists affects the data locality and overall simulation performance. If the overhead cost is not an issue, then this research can be expanded to project all the neighbour lists, especially for the 3-dimensional case, onto a spherical surface centred at the middle of the cell containing particle $i$ which, in turn, will cover all the neighbouring cells and their order is based on the distance of the projection of their particles projection from the centre point.

### 3.3.2 MDS Optimization Based on Re-ordering

Meloni et al. [102] have proposed a re-ordering algorithm based on the linked cell method to improve locality of reference that re-orders particles based on transforming the sparse interaction matrix into a banded matrix. Element $(i, j)$ of the interaction matrix is 1 if particle $i$ interacts with particle $j$, and is 0 otherwise. Re-ordering may be performed by the reverse Cuthill-McKee (RCM) algorithm based on the Verlet table, but this has drawbacks related to loss of locality and excessive computation. However, the solution proposed by Meloni et al. is based on the linked-cell approach; particles in the same cell are labelled consecutively and cells are ordered in row-major order. Meloni et al. found that this ordering produced a higher degree of clustering in the elements of the interaction matrix compared with the RCM algorithm, thereby improving data locality and performance.

Luo and Liu [96] sought to improve data locality by storing, for a given cell, $I_c$, the position data of all particles in $I_c$ and its surrounding cells in a temporary array. This is similar to creating a temporary neighbour list for a cell, rather than for individual particles. Its effectiveness is determined by the trade-off between the overhead in creating the temporary list for each cell and the performance gain from the improved data locality.

Gonnet [56] addresses the large memory requirements of storing a neighbour list for each particle by means of *pseudo-Verlet lists*, which is a separate array per cell. This approach is similar to the methods presented in [98, 154]. Particles are processed by cell, with particles in one cell interacting with those in neighbouring cells. The construction of a pseudo-Verlet list involves sorting particles in each pair of neighbouring cells along a line connecting their centres. These sorted lists, based on a combination of quicksort and insertion sort, are then used to determine if two particles interact, rather than using traditional neighbour lists that store every potentially interacting pair of particles; in fact, the storage requirement for pseudo-Verlet lists in three-dimensional simulations is only 13 times the number of particles. Although the approach resolves the memory conflict issue, its efficiency is reduced as the simulation system has high density which requires additional memory bandwidth to load the pair-wise Verlet list. Furthermore, the study does not cover memory analysis, nor the probability of idle threads in relation to the increase in involved cores, where the number of idle threads is expected to increase as the pairs in the list are consumed in the computation.

Meel et al. [139] and Anderson et al. [15] were among the first researchers to fully implement molecular dynamics simulations on a GPU using CUDA. The former focused purely on the implementation itself and the generation of random numbers. Their implementation was limited to the use of the shared memory, which impacts the size of the linked cell. In addition, they did not resolve the issue of thread synchronization. Their performance analysis is limited to a comparison between the CPU and GPU implementations. Performance results are presented for Lennard-Jones fluids and polymer systems, and show that GPUs are a cost-effective alternative to the use of CPU clusters. They also use a Hilbert ordering for particles, which was found to reduce execution time in comparison with randomly ordered particles. However, this research has a number of limitations, such as the simulation size being small (up to $125,000$ particles); memory analysis and CUDA profiling are not presented; the use of synchronization and shared memory might impact the overall performance due to a trade-off in avoiding memory conflicts and race conditions; and finally, the algorithm has a number of conditional statements that may drive the threads to suffer from divergence issues. More recently, Tang and Karniadakis [135] have developed an optimized molecular dynamics simulation code based on the LAMMPS application. This hybrid parallel code uses MPI on the CPUs, and each MPI process handles a single GPU. Cells, and particles within the cell, are indexed in Morton order. Streaming is used to hide the latency of communication between CPUs and GPUs, and of kernel launch. Shared memory and a warp-centric programming model is used on the GPU to enhance performance. Analysis mainly covers the speed-up and data locality that was achieved by the proposed method based on Morton ordering. In addition, the performance results were not compared with previous work using other sorting methods.

The GPU performance of molecular dynamics simulations with Tersoff, embedded-atom model, and Lennard-Jones potentials have been compared by Minkin et al. [103]. Their implementation uses OpenCL, and computes neighbour lists and particle forces on the GPU. However, it is not clear how frequently the neighbour lists are updated, and the number of particles considered (up to 16000) is smaller than the simulations conducted in this dissertation.

The use of Hilbert and Morton orderings to enhance data locality for molecular dynamics and other irregular applications has been investigated by Mellor-Crummey et al. [101]. Their simulated results for a uni-processor workstation show that reordering both the data and computations using a Hilbert curve can significantly reduce the number of L1 cache, L2 cache, and TLB misses, thereby reducing the number of execution cycles.

Kunaseth et al. [86] have investigated how the data fragmentation ratio, $N_{frag}$, defined as the fraction of particles in a molecular dynamics simulation that have moved out of their original cell, affects the number of data translation lookaside buffer (DTLB) misses. At the start of the simulation the particles in a cubical cell of size $r_0$ are sorted, so they are contiguous in memory and $N_{frag} = 0$. However, as the simulation progresses some particles will move out of their original cell, causing $N_{frag}$ to increase. This effect is more marked at higher temperature. Kunaseth et al. show that in a low viscosity liquid the value of $N_{frag}$ and the DTLB miss rate both increase with the number of times steps, which accounts for the increase in execution time per time step. The use of Hilbert and Morton ordering for cells is also considered, but was found to make little difference to the execution time on the Intel Core i7 processor used in their experiments. Kunaseth et al. also show that there is an optimal frequency of particle reordering, which depends on temperature. The experiments have been carried on silica material and the combustion of aluminium nanoparticles, but the force field method used in the simulation is not described.

An alternative to the stencil-based approach to building neighbour lists has been proposed by Howard et al. [66]. They make use of a linear bounded volume hierarchy (LBVH) for computing neighbour lists, which partitions nearby particles into axially-aligned boxes. These boxes are then enclosed in increasingly larger boxes to form a hierarchy, which can be represented by a tree. When building a particle's neighbour list certain branches of the tree can be ignored because the corresponding boxes are too far apart. Howard et al. compare GPU implementations of the stencil and LBVH approaches to building neighbour lists, and found that the latter is significantly faster for colloidal systems characterised by large size disparities.

Another alternative to the stencil-based approach is to manage the neighbour list by using an octree, which has been adopted specifically for long-range interactions such as in free energy modelling on GPUs [25]. This is based on a force field approximation approach, as described in [29, 30]. The Octree Pairwise Approximation (OPA) algorithm has been customised with a user-defined parameter $\epsilon$ that determines if the computation should be carried out with an approximate or an exact approach, or to select which resource (CPU or GPU) to run the computation on. The octree was found to provide more data locality, outperforms the exact force computation, and is more memory efficient than the use of neighbour lists. The OPA algorithm has been implemented for multiple GPUs and allows a trade-off between accuracy and simulation time. Analysis of the memory and cache utilization has not been addressed in this work.

Jiang et al. [74] have proposed the use of asynchronous data transformations to accelerate the task of reordering the data in irregular dynamic applications. This is done by using a helper thread to analyse the interaction list (this is a list of pairwise interactions between particles). Based on this analysis, the helper thread provides a new particle ordering to the master thread, where most of the computation takes place. The helper and master thread are coordinated through a shared variable protected by a lock. To ensure program correctness, the actual reordering of particles is done by the master thread after a new ordering has been determined by the helper thread. Likewise, the helper thread will start to analyse the interaction list after a new one has been made available by the master thread. This approach hides some of the overheads associated with data ordering. Jiang et al. also demonstrated the use of a GPU in performing data transformations. They view the data locality optimization problem as a graph partitioning problem. Particles correspond to nodes in the graph, and those that interact are connected by an arc. The partitioning algorithm places nodes in clusters containing nodes that are close in the graph topology.

Glaser et al. [53] have shown good GPU scalability on up to 3375 GPUs nodes for Lennard-Jones and dissipative particle dynamics simulations of up to 108 million particles. They have reimplemented the traditional MPI decomposition (the CPU-based code of Plimpton [114]) in a multi-GPU implementation. They use GPU-aware MPI to communicate data, especially ghost particles, between the GPUs without copying the data via the CPU. The traditional MPI version requires only six communications between the neighbour nodes, typically for large amounts of particle data. In the implementation of Glaser et al. each MPI process communicates directly with all 26 neighbour processes using non-blocking messages. In addition, mapped and pinned memory have been used to provide higher bandwidth and to overlap computation and data transfer. In order to achieve efficient particle exchange, each GPU sorts the particles according to the MPI rank of the destination and the buffer boundaries are defined according to the number of particles for each rank. They have compared host-memory MPI, GPU-aware MPI, and GPUDirect RDMA communication methods. The performance results for single-precision and double-precision are best for host-memory MPI and GPUDirect RDMA, respectively.

Hardy et al. [61] have implemented a GPU-based algorithm for electrostatic potentials by using the multilevel summation method (MSM) based on the splitting and interpolation strategies introduced in the Multiple Grid Method [128]. This approach enables the CPU to control the work assigned to the GPU, and enables the computation of electrostatic potentials for one million atom systems within a few seconds. MSM can be used for short-range and long-range lattice cutoffs, and also supports boundary conditions with no cutoff. Other algorithms that are often used with electrostatic potentials are the Particle Mesh Ewald (PME) and Fast Multipole Method (FMM). The Hierarchical Charge Partitioning (HCP) approximation has been proposed in [11, 12] to speed up computation of pairwise electrostatic interactions based on an approximation to the natural partitioning of biomolecules into their constituent components. In this method, at short distances from the point of interest, HCP uses the full set of atomic charges, but for long distance interactions, approximate charge distributions are used instead. The results show that for a simulation on a single CPU, the 1-charge and 2-charge HCP are 40% and 14%, respectively, faster than the spherical cutoff method, and three times the speed for a single point computation. However, the method is very sensitive to the number of charges in the approximation, the distribution of charges per component, and the threshold distance has an inverse relationship between accuracy and speedup.

Kazachenko et al. [80] have investigated the challenges arising in simulations of complex fluids in the canonical (NVT) ensemble. This work covers the complex and varied interactions in multicomponent fluids, intramolecular and intermolecular force components, and fully atomistic and coarse-grained models. This research presents a CUDA implementation of Lennard-Jones (LJ) and Gay-Berne (GB) pairwise interactions, Ewald summation for electrostatic interactions, and adapted the algorithm of Martyna to combine a Nosé-Hoover thermostat with fixed bond lengths [14]. They examined the simulation of 3 systems: *2-(4-butyloxyphenyl)-5-octyloxypyrimidine*, SP-C/E (extended single point charge) water, and an n-hexane/2-propanol mixture. In these simulations, periodic boundary conditions were used with the minimum image convention, and a spherical cutoff radius, $r_c$. The data structure for maintaining neighbour lists was created by defining four types of pairwise interactions (LJ-LJ, LJ-GB, GB-LJ, and GB-GB) in order to guarantee coalesced access to GPU memory. The neighbour list excludes an atom from interacting with itself and other atoms in the same molecule that are deemed close.

## 3.4 Summary

This literature review has covered all aspects of our work to investigate the existing research that has been conducted in relation to data locality. Data locality has been explored from the compiler perspective and, in particular, the algorithmic perspective – specifically for data locality enhancements. Data locality and optimization in molecular dynamics simulations have been investigated to discover related research conducted in this area. Data locality and MDS implementations on GPU platforms were studied to analyse if there is any research comparable to that presented in this dissertation, or if there is any research on data locality specifically for GPUs that can be adopted for MDS implementations on GPUs. However, it was found that only a limited amount of research addresses the data locality issue for molecular dynamics simulations on GPUs. Thus, previously the data locality issue has not been addressed widely in GPU implementations.

Chapter 4

# Stencil and Data Locality Properties

In Chapter 2, previous research on data locality enhancements was highlighted, in particular methods related to data ordering. In addition, in Section 2.3, various data ordering methods that are used in this research were identified and detailed. In this chapter, the data locality properties of each of the data orderings will be investigated. In spatial applications in which performance is strongly affected by data dependency, processing an item at $(i, j, k)$ in an array usually requires access to other neighbouring items in the array. Identifying neighbouring items based on a stencil reduces the computation time compared to searching and finding the neighbouring items every time that an item is processed. It is assumed that the shape of the stencil is the same for all locations. Therefore, the stencil centre is at the item's location and then the memory offsets of the stencil locations relative to the stencil centre are stored. As an example, a simple "star" stencil consists of six adjacent items that are adjacent to the centre, and the stencil offsets are: $(0, -1, 0)$, $(-1, 0, 0)$, $(0, 0, -1)$ $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. Hence, for an $M \times M \times M$ row-major ordering, this can be formulated as item offsets $(-M^2, -M, -1, 0, 1, M, M^2)$ for all locations. The stencil may consider periodic boundary conditions, such that those items at the borders have an adjacent neighbour items on the opposite border.

For Hilbert and Morton orderings, applying the stencil to an item must consider the position within the ordering. Hence, for Hilbert and Morton orderings the offsets in memory depend on location and it is not possible to store a common stencil for all locations. For the sake of simplicity, this issue is resolved by mapping the Hilbert and Morton index to the corresponding row-major index, and then applying the stencil on the row-major ordering to find the adjacent neighbours items. These, in turn, are mapped back into their corresponding Hilbert or Morton indices. As the arrays that perform these mappings are generated only once and their values are stored in the memory, this operation has order of complexity $\mathcal{O}(1)$.

## 4.1 Spherical Stencil

The cubic stencil is one of the most common stencils that is applied in various spatial applications, such as for solution of partial differential equations. The cubic stencil consists of a $(2g + 1) \times (2g + 1) \times (2g + 1)$ block of array locations. Another stencil of interest, particularly in molecular dynamics simulations (detailed in Chapter 5) is the approximately spherical stencil. If the 3D array is composed of $M \times M \times M$ spatial bins, each of unit size, then the approximately spherical stencil consists of those bins that are fully or partially within a specified distance, $g$, of any of the vertices of the stencil centre, where $g$ is a positive integer. The number of bins in the stencil is given by:

$$M_0(g) = 1 + 6g + 12 \sum_{i=0}^{g-1} \left\lceil \sqrt{g^2 - i^2} \right\rceil + 8 \sum_{i=0}^{g-1} \sum_{j=0}^{p-1} \left\lceil \sqrt{g^2 - i^2 - j^2} \right\rceil \tag{4.1}$$

where $p = \left\lceil \sqrt{g^2 - i^2} \right\rceil$. The terms on the left-hand side of this equation correspond to:

Term 1: the central bin of the stencil;

Term 2: the bins in each of the 6 directions about the central bin;

Term 3: the bins lying along the 12 edges of the cube;

Term 4: the remaining bins in the 8 quadrants of the cube.

As $g$ increases the volume of the set of bins in the stencil progressively becomes a better approximation to that of a sphere of radius $g$, and $M_0(g)$ is shown for a few values of $g$ in Table 4.1, together with the percentage deviation from sphericity, given by:

$$1 - \frac{4\pi}{3}\frac{g^3}{M_0(g)}$$

It should be noted that the deviation from sphericity becomes less than 10% for $g > 40$.

| $g$ | Number of bins, $M_0(g)$ | Deviation (%) |
|---|---|---|
| 1 | 27 | 84.49 |
| 2 | 125 | 73.19 |
| 3 | 311 | 63.63 |
| 4 | 613 | 56.27 |
| 5 | 1015 | 48.41 |
| 6 | 1689 | 46.43 |
| 7 | 2399 | 40.11 |
| 8 | 3449 | 37.82 |
| 9 | 4675 | 34.68 |

**Table 4.1: Number of bins and the percentage deviation from sphericity.**

## 4.2 Data Locality Metrics

Data locality is impacted by various factors, such as the hardware architecture, cache size and configuration, data access pattern, data dependency level, and data ordering. Spatial and temporal locality are the major metrics used to define the efficiency of cache utilization. Normally, these factors are not revealed in profiling results, and are represented by only the cache hit or miss rates. The spatial locality principle refers to the fact that when a data item to be processed is moved into a higher level of memory other items upon which that processing depends will be brought into that memory level in the same cache block. Temporal locality refers to whether a data item to be accessed is found in high level memory, having been placed there previously. In this section, an analysis is conducted to study the data locality of different data orderings and stencils (to our best knowledge, such a study has not been conducted in the existing literature).

### 4.2.1 Memory Access Patterns

It is assumed that each array location represents a spatial bin containing a number of items that can be processed independently. This processing depends on using (and reusing) data within the nearby locations defined by the stencil, and the memory locations at which the stencil data are stored is determined by the ordering used.

We have analysed the relation between data orderings and the efficient use of memory hierarchy by examining memory access patterns associated with a given stencil. For a row-major ordering the memory access pattern for a given stencil is independent of array location, but for Hilbert and Morton orderings it is not. Therefore, we capture an overall view of the memory access pattern by making a plot of the memory offsets corresponding to a particular stencil and ordering, accumulated over all array locations. This analysis is achieved by implementing a simple model that measure the distance between a bin and other neighbour bins that are defined based on the adopted stencil. The stencil is generated based on Algorithm 5.2, which uses the stencil width as an input and generates the offsets of the neighbour bins. Then, the adopted stencil is used to find the neighbour bins for each bin. The distance between each bin and its corresponding neighbour bins is calculated. An histogram of distance is represented by an array and the index of the array represents the distance. Accordingly, neighbour bins are registered in the histogram array by incrementing the number of bins within the corresponding index as shown in Algorithm 4.1, noting that the stencil is symmetric and the number of bins can be divided by two, which represents the positive and negative values (of course, these could be represented by two arrays: one for the positive values and the second for the negative values). This algorithm can be then used with different configurations, such as the type of stencil (cubic or approximately spherical), for different sizes of the 3-dimensional array, types of data orderings, and stencil widths.

**Algorithm 4.1:** distHistogram: high level of the histogram distances computation. The functions `path2RMO` and `RMO2path` convert between a location in the ordering and the row-major index.

```
Function distHistogram(ordering,stencil,M,g)
    Input: ordering, stencil, integers M and g.
    Output: The absoulate distance value between bins i and their sencil bins.
    histogram[k]=0
    foreach ( ibin ) do
        ipath = RMO2path (ibin)
        foreach (k, stencil, in ordering) do
            jbin = ibin + stencil[sbin]
            jpath = RMO2path (jbin)
            pathdiff =|jpath − ipath|
            histogram[pathdiff]++
        end
    end
end
```

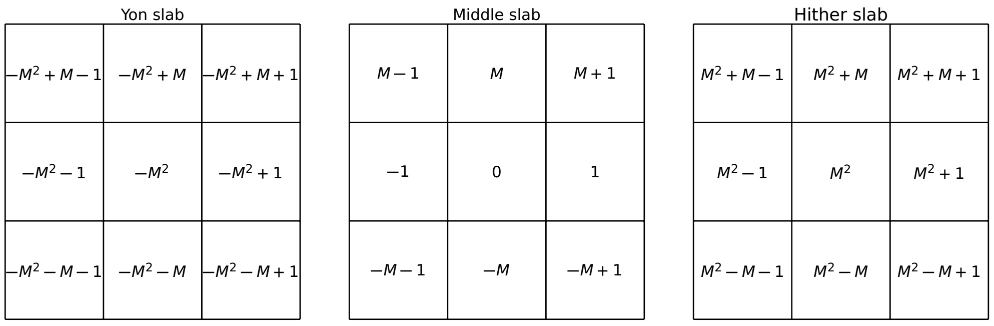


**Figure 4.1: Memory offsets for a block stencil with $g = 1$.**

Figure 4.2 gives a graphical summary of the memory access pattern for a block stencil with $g = 1$ for row-major, Hilbert and Morton orderings of a $16 \times 16 \times 16$ array. In this case, where $g = 1$, memory offsets are accumulated over a 14×14×14 array, as a border of depth $g$ is required by the stencil. The total number of the accumulated memory offsets for the 27 stencil bin in the row-major ordering case is equal to $14^3 = 2744$, and this correspond to the offsets shown in Figure 4.1, ranging from -273 to +273. It is clear that the Hilbert and Morton orderings have a greater degree of scatter in their memory access patterns, and, in both cases, this extends beyond the limit of the x-axis in Figure 4.2. In the Hilbert ordering case, the memory offsets lie between ±3767, and

13.3% are not included in Figure 4.2. For Morton ordering the corresponding values are ±3073 and 13.8%.

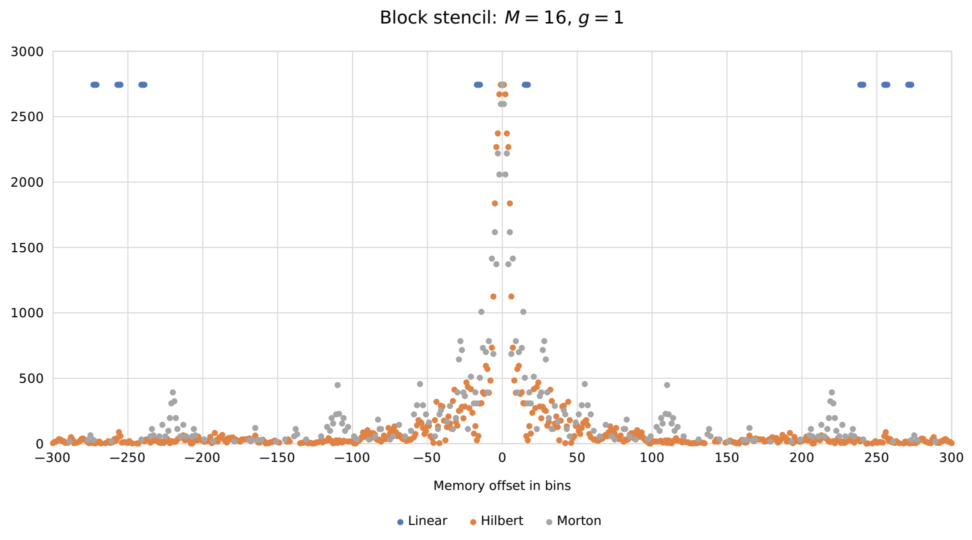


**Figure 4.2: Accumulated memory offsets in bins for a block stencil with $M = 16$ and $g = 1$.**

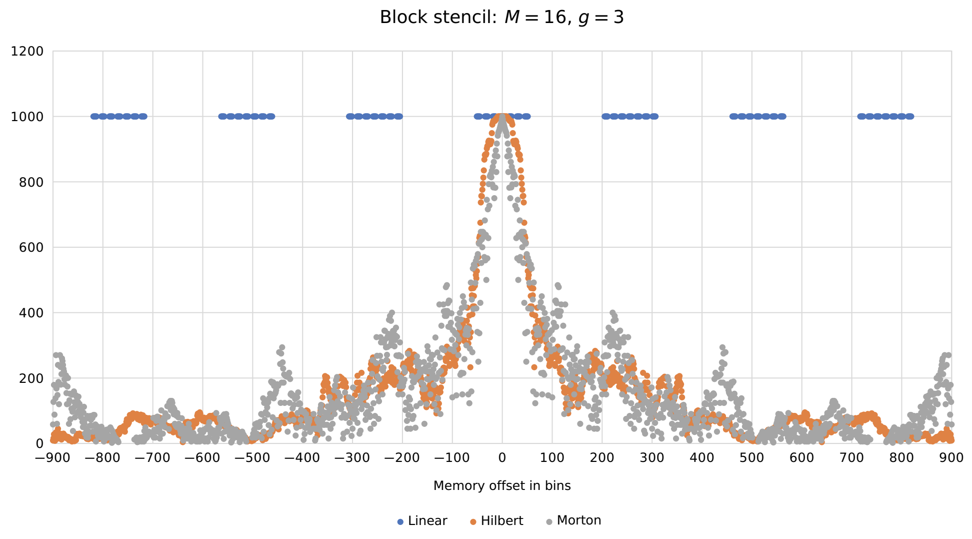


**Figure 4.3: Accumulated memory offsets in bins for a block stencil with $M = 16$ and $g = 3$.**

Figure 4.3 shows another example of similar data to Figure 4.2, but for a block stencil with $g = 3$. In this case, the border width of a block is 3, so memory offsets are gathered over a $10 \times 10 \times 10$ array. It is clear that the memory access pattern is more scattered for the Hilbert and Morton orderings than for the row-major ordering, as in the $g = 1$ case. For the Hilbert ordering, the memory offsets ranges between ±3794, and 13.3% and are not covered in Figure 4.3. Likewise, the Morton ordering values are ±3129 and 22.0%.

The data shown in Figures 4.2 and 4.3 show how the bins are scattered in the memory but do not show the ratio of bins that are allocated close to each other and at which distance all the bins are allocated. Thus, they can be presented in terms of a histogram showing the cumulative fraction of bins within a given absolute memory offset by still using the Algorithm 4.1 and a range of size 100. The cumulative fraction is shown in Figures 4.4, and 4.5 for $g = 1$ and $g = 3$, respectively, and $M = 16$. It can be seen that for $g = 1$, only one third of the bins are within a memory offset of 199 for the row-major ordering, whereas for the Hilbert and Morton orderings 0.817 and 0.787, respectively of the bins are within this memory offset. However, for the row-major ordering, all the bins are within a memory offset of 299, and the corresponding values for the Hilbert and Morton orderings are 0.867, and 0.862, respectively. As a consequence, compared with the row-major case, a higher proportion of the bins are within a small memory offset in the Hilbert and Morton cases, and the reverse is true for large memory offsets. A similar trend can be observed in Figure 4.5 for the $g = 3$ case where all the bins are within a memory offset of 899 for the row-major ordering, whereas only 0.795 and 0.780 are within this offset for the Hilbert and Morton orderings. Moreover, Figures 4.4 and 4.5 also show that a higher proportion of bins are within a given memory offset for the Hilbert ordering compared with Morton ordering, up to an offset of about 899, and after that the opposite is true.

Figures 4.4 and 4.5 indicate that for sufficiently small cache sizes, the fraction of the data needed to update the items in a bin that will fit into the cache is largest for Hilbert ordering, followed by Morton and row-major orderings; however, for a large enough cache, this is reversed. Therefore, this suggests that the performance benefits of the different orderings will depend on the size of the different levels in the memory hierarchy.

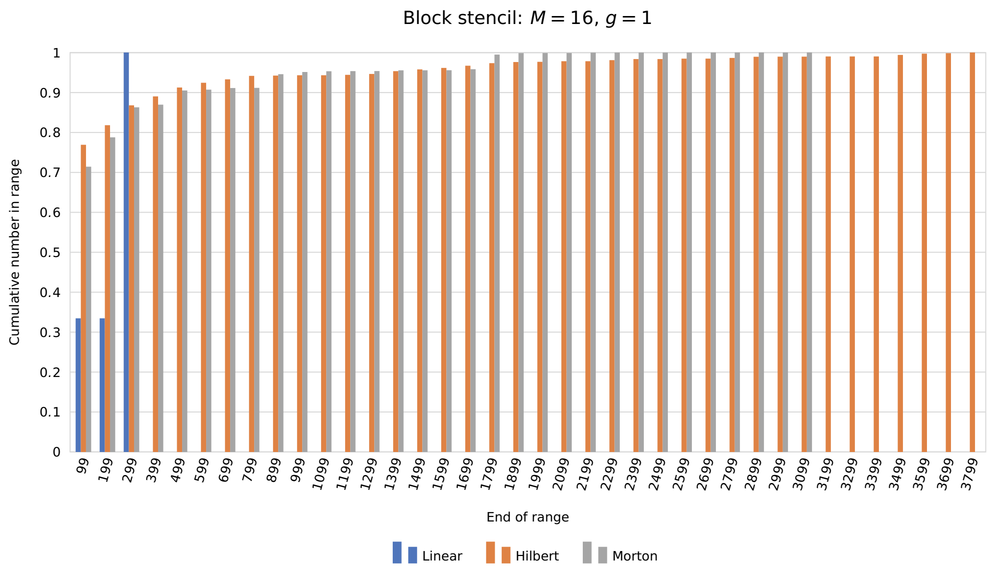


**Figure 4.4: Cumulative fraction of bins within a given memory offset for a block stencil with $g = 1$ and an array with $M = 16$. For each set of bars the range is from 0 up to the x-axis label.**

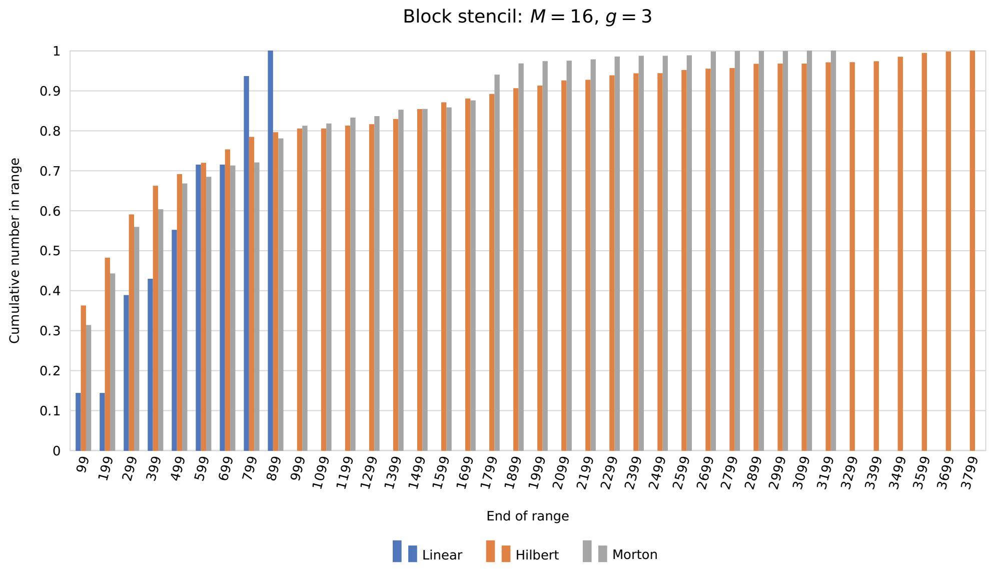


**Figure 4.5: Cumulative fraction of bins within a given memory offset for a block stencil with $g = 3$ and an array with $M = 16$. For each set of bars the range is from 0 up to the x-axis label.**

### 4.2.2 Cache Miss Rate

If the x-axis labels are replaced with cache size, then Figures 4.4 and 4.5 can also be interpreted as hit rate curves; this is a plot of the probability of a cache hit as a function of cache size. However, in order to obtain a more accurate estimate of the hit rate and the number of cache line transfers in a given time period, we developed a simple dynamic model for the cache. In this cache model, bins are stored in memory in some prescribed order (row-major, Hilbert or Morton order). The model is configured with cache line size $b$, and main memory is viewed as being divided into blocks of size $b$. Whenever a bin is not found in cache, the block in main memory containing that bin is moved into the cache. The cache has a limited size and we assume that it can hold a maximum of $c$ blocks, or $cb$ bins, and whenever the cache is full and a cache miss occurs, then the least recently used (LRU) block is ejected from the cache. For each bin in the array, the corresponding stencil bins are accessed and the number of cache misses is recorded. An outline of the simple cache module is given in Algorithm 4.2, where it should be noted that only bins not in the border zone of depth $g$ are considered. Additionally, the functions *path2RMO* and *RMO2path* are the mapping functions to and from the row-major and the memory ordering used.

The cache model has various configurable parameters: the ordering, the stencil type and size, $g$, the size of the 3D array, $M$, the number of bins per block size, $b$, and the number of blocks in the cache, $c$.The cache module consists of orderings, stencil, `cacheBlock`, and `cacheArray` classes. The `cacheArray` objective is to check if the requested bin is within any of the cache blocks or not, as shown in Algorithm 4.4, that uses Algorithm 4.3 (a method in `cacheBlock`) to check individual blocks. In addition, it is responsible for identifying in which block the bin is going to be stored, which is used by the `addToCache` function. The `addToCache` maintains the first bin index registered in each block, the least recently used bin, and a counter of number of blocks that are in the cache.

The simulation is initiated by generating the ordering indices, building the stencil ac-

**Algorithm 4.2:** cacheModel: high level view of the cache model. The functions `path2RMO` and `RMO2path` convert between a location in the ordering and the row-major index.

```
Function cacheModel(ordering,stencil,M,g)
    Input: ordering, stencil, integers M and g defining size of the array and the stencil.
    Output: The number of cache misses nmisses.
    nmisses = 0
    foreach (location, ipath, in ordering) do
        ibin = path2RMO (ipath)
        if (ibin not in border zone) then
            foreach (stencil location, sbin) do
                jbin = ibin + stencil[sbin]
                jpath = RMO2path (jbin)
                if (!inCache (jpath,ordering)) then
                    nmisses++
                    addToCache (jbin)
                end
            end
        end
    end
    return nmisses
end
```

**Algorithm 4.3:** inBlock: check if the bin exists witin the block or not.

```
Function inBlock(bin, ordering)
    if (first bin not yet initiated) then
        return 0
    end
    for i = 0 to binsPerBlock do
        ibin = firstBin + i
        if ( bin= ibin) then                                // bin is found
            return 1
        end
    end
    return 0
end
```

cording to Algorithm 5.2, setting up the cache array according to the input parameters, and finally calling the simulation of the cache model according to Algorithm 4.2. Algorithm 4.2 traces each bin within the defined ordering path. Then the neighbour bins, that are generated according to the adopted stencil, are tested by checking their availability in the cache array. If a neighbour bin is not within the cache, the number of misses

**Algorithm 4.4:** inCache: check if the bin exists in any of the blocks within the cache by calling `inBlock` method. If the bin exist, modify the recent used bin to the last accessed bin and increment number of cache misses.

```
Function inCache(bin, ordering)
    for $i = 0$ to blocksMax do
        if ( $cacheArray[i]$.inBlock (bin,ordering) ) then  // call inBlock method
            $cacheArray[i].lastused = tcount++$
            return 1;
        end
    end
    $nmiss++$
    $tcount++$
    return 0
end
```

**Algorithm 4.5:** addToCache: insert the *jbin* in the block with the *jspace* index and then modify the corresponding indices (*firstBin*, *lastUsed* and *nblocks*) to track the insertion and removal of the bins.

```
Function addToCache(jbin,jspace, ordering)
    $binsPerBlock = cachArray[0].binsPerBlock$          // copy binsPerBloc

    // [
    f]*set the first bin number
    $cacheArray[jspace].firstBin = (jbin/binsPerBlock) * binsPerBlock$
    $cacheArray[jspace].lastUsed = tcount$   // set last recently used block
    $nblocks++$          // increment the index of the number of blocks

end
```

will be incremented by one and the block will be stored into the cache according to Algorithm 4.5, after the required space is created.

A number of cache models have been run and analysed, and these show some common characteristics. For example, Figures 4.6 and 4.7 show miss rate plots for $M = 32$, a block stencil with $g = 1$, and cache block sizes of $b = 2$ and $b = 8$ bins, respectively. In both plots, it can be seen that in the row-major case the miss rate tends to stay constant for a range of cache sizes, and then decreases in steps as the cache size increases. This decrease occurs whenever the cache size is large enough to hold an additional complete row of bins. The miss rates for the Hilbert and Morton cases do not exhibit this behaviour as they are not ordered by row. Figure 4.6 shows that for small cache size, the miss rate is high for the Hilbert case, but for cache size between $c = 64$ and $c = 1024$, the miss rate is the lowest for the Hilbert ordering, closely followed by the Morton case, with row-major ordering having the highest miss rate. Figure 4.7 also shows that the ordering with the lowest miss rate depends on the cache size.

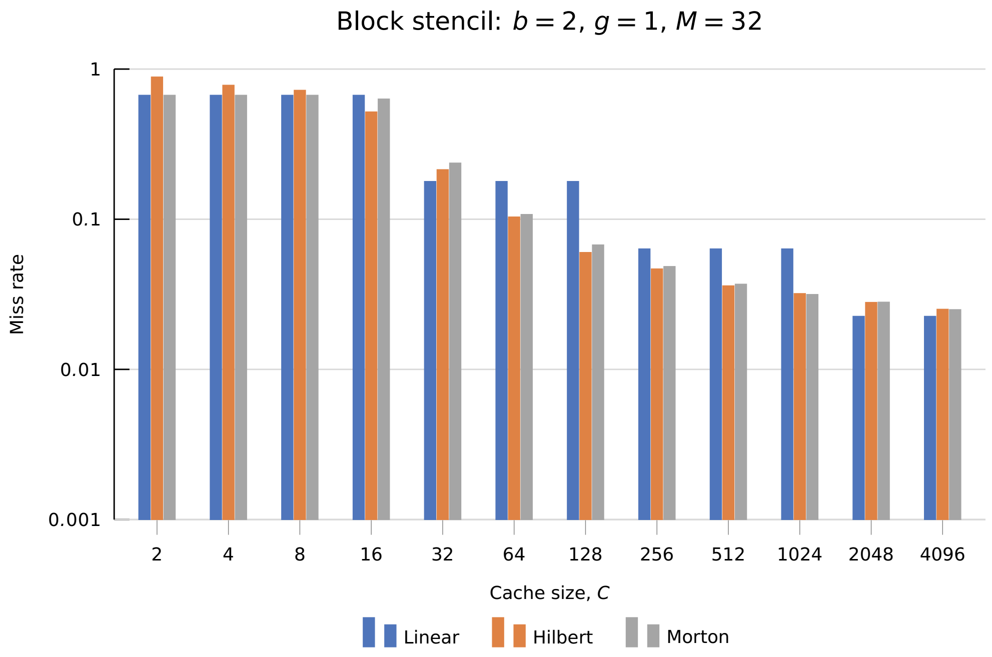


**Figure 4.6: Miss rate as a function of cache size, $c$, for a block stencil with $g = 1$, an $M = 32$ array, and a cache block size of $b = 2$ bins.**

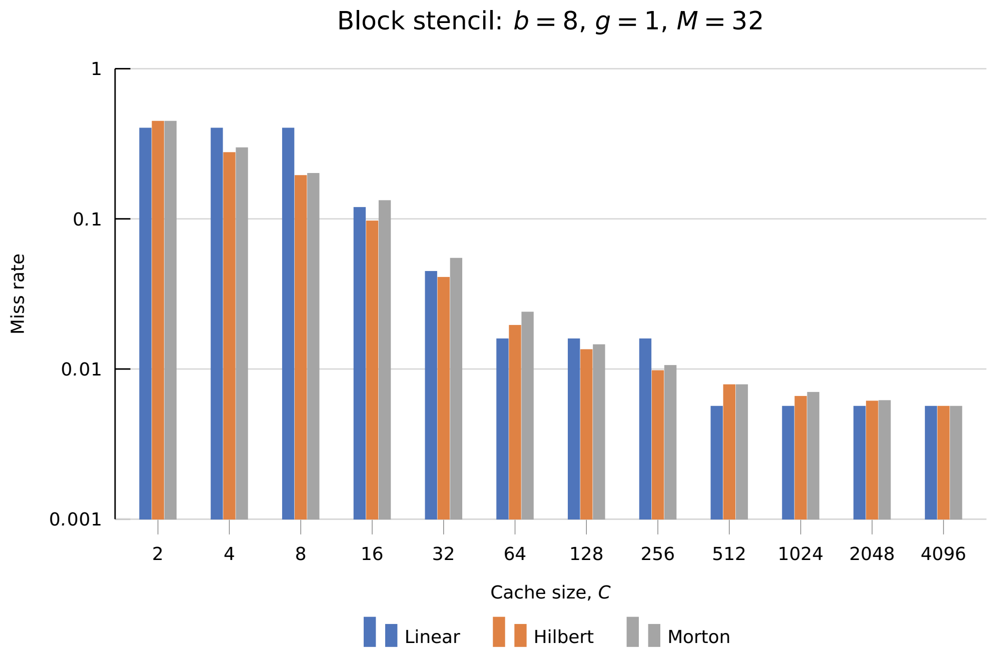


**Figure 4.7: Miss rate as a function of cache size, $c$, for a block stencil with $g = 1$, an $M = 32$ array, and a cache block size of $b = 8$ bins.**

Similar behaviour to the block stencil cases is seen for approximately spherical stencils for all the orderings. For example, Figure 4.8 shows the miss rate for $M = 32$, an approximately spherical stencil with $g = 3$, and $b = 8$. In addition, for an approximately spherical stencil with a large cache size, the miss rate is lower than for a block stencil because the number of stencil bins that are accessed by a bin is less than for the block stencil case. Figure 4.9 shows the miss rate data for $M = 64$, a block stencil with $g = 1$, and $b = 8$. The plot emphasises again that the ordering with lower miss rate depends critically on the cache block size, $b$, and the overall cache size, $c$.

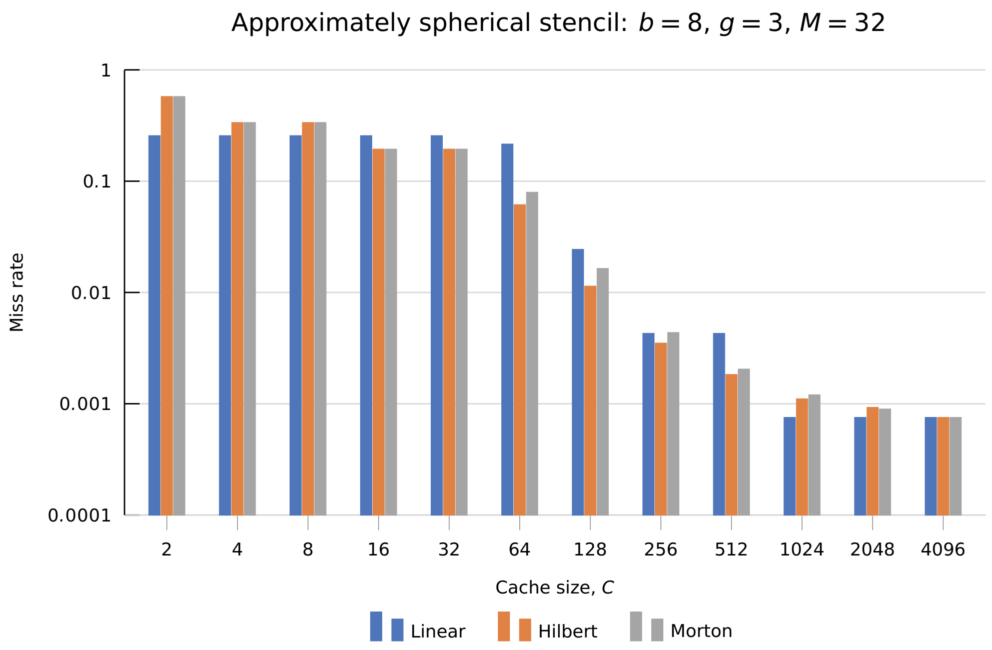


**Figure 4.8: Miss rate as a function of cache size, $c$, for an approximately spherical stencil with $g = 3$, an $M = 32$ array, and a cache block size of $b = 8$ bins.**

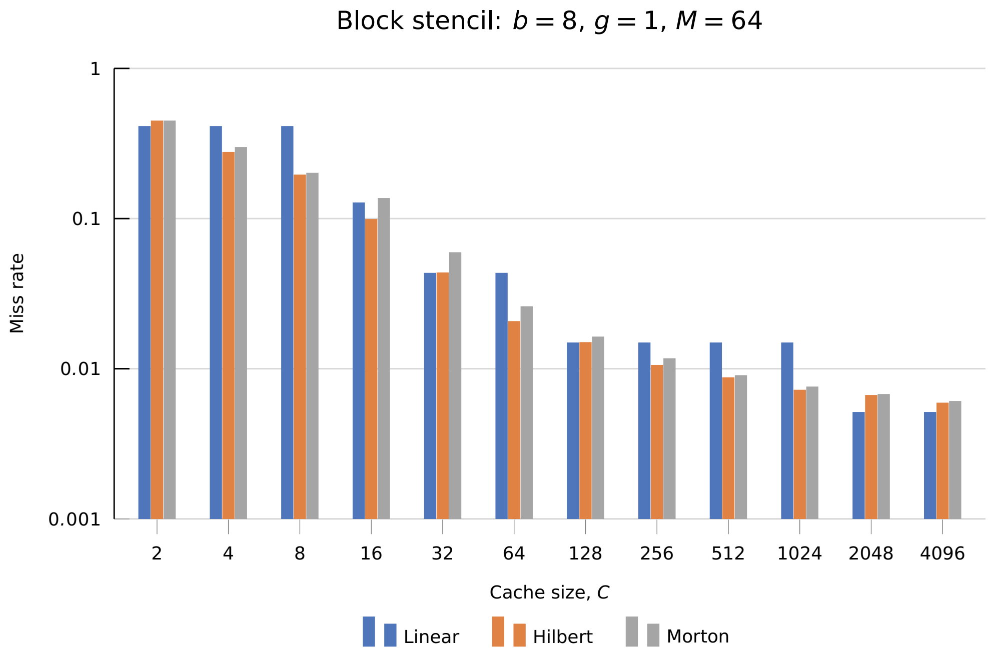


**Figure 4.9: Miss rate as a function of cache size, $c$, for a block stencil with $g = 1$, an $M = 64$ array, and a cache block size of $b = 8$ bins.**

### 4.2.3 Shared Stencil Bins

Another factor that has been studied is the number of bins that are shared between the stencils centred at consecutive bins indexed by $i$ and $i + 1$. The assumption is that processing bin $i$ requires a number of stencil bins to be loaded into cache memory; when processing for bin $i$ is completed the bins needed to process bin $i + 1$ will be required, and because the stencils for bin $i$ and $i + 1$ overlap a portion of these bins will already be in cache. We found that there is a constant maximum number of bins shared between the stencils for bins $i$ and $i + 1$ equal to $2g(2g + 1)^2$, for all orderings. Consequently, we can infer that the data ordering has little impact on the temporal data locality.

## 4.3 Summary

This chapter has presented an approximately spherical stencil that is often used in various applications, especially in molecular dynamics simulations. The relation between data ordering and an approximately spherical stencil have been explored based on memory access patterns and a cache model. We found that the memory access pattern is more scattered for the Hilbert and Morton orderings compared to the row-major order. However, the Hilbert and Morton orderings have a larger number of neighbours within a short distance of the memory offset.

This chapter presented a simple cache model to explore in detail the relationship between data orderings and the size and shape of the stencil. It was found that cache block size and the overall cache size have significant impacts on the cache behaviour of different data orderings.

*Chapter 5*

# GPU Implementation

This chapter describes the miniMD application and the modifications made to it in order to achieve our research objectives. The objectives and scope of our research will be revisited to aid the reader. The ordering module will be described in detail, as well as how it is integrated with the other modules of miniMD. The force computation module, which is used for our research, will be described in detail in order to show how it is implemented in our study.

MiniMD is a scaled-down version of the LAMMPS molecular dynamic simulation application, and was created mainly to achieve high performance as part of the Mantevo project [36, 87]. This chapter provides an overview of the implementation of miniMD in order to provide details of how it can be used and how it carries out the MDS for a Lennard-Jones potential. In addition, this will help to understand the modifications that have been performed in this research. Therefore, we will highlight miniMD's modules and components. As highlighted in Chapter 2, most molecular dynamics simulation codes are composed of modules for input and output, particle force evaluation, managing neighbour lists, potential energy and thermodynamics computations, and integration to update particle positions and velocities. In addition, there is data sharing/communication functionality specific to particular platforms that facilitates a massively parallel implementation, such as OpenMP and MPI. This is encapsulated in a communication module in the miniMD application.

## 5.1 Objectives and Scope

As mentioned in Chapter 1, the main objectives of our research are to study the data locality properties of molecular dynamics simulations, and to investigate these in the context of a GPU implementation of miniMD. It is not the focus of our study to create an optimised CUDA version of miniMD, although this would definitely improve the overall simulation time, especially if the data always resided in the GPU memory and was copied between to host and the GPU only at the beginning and end of the simulation. In addition, with the MPI Direct feature of CUDA, or some other MPI-aware CUDA implementation, the simulation could be extended to multiple GPUs. Hence, our modifications to miniMD are limited to the force computation, which is found to account for more than 80% of the simulation time. In addition, we have maintained the principle that miniMD was created as a performance comparison tool without trying to modify the data structures, especially that of the neighbour lists, in order to provide an additional optimisation that might already have been addressed in our literature review chapter. Therefore, we limit our research mainly to the study of the locality properties of different data orderings, and leave other optimisation opportunities for future work.

## 5.2 MiniMD Modules

MiniMD (version 2.0) was developed using the C/C++ programming language and its modules are implemented by classes that define variables and methods associated with each module. It supports both an MPI and a hybrid OpenMP/MPI parallel implementation. MiniMD is limited to Lennard-Jones (LJ) and embedded atom model (EAM) particle interactions. It uses neighbour lists in the force calculation, which can be of the full-neighbour or half-neighbour type. In the former case, when finding the force on a particle, the particle interacts with other particles in all the stencil cells. In the latter case, use is made of Newton's Third Law, which means that a particle interacts with only particles in the cell at the stencil centre and those in half the other stencil cells.

In version 2.0 of miniMD particle types are introduced in order to simulate multiple particle types.

### 5.2.1 Input Module

The input module provides the execution settings, the simulation system configuration, and technical settings. The run settings configure the number of MPI processors, the number of threads for OpenMP, and specifies the name of the input file, which sets the parameters for the simulation, and the data file, which contains the initial coordinates and velocities of the particles. The miniMD execution settings can be set as arguments when invoking the program, as shown in Table 5.1.

| Argument | Value (e.g.) | Description. |
|---|---|---|
| $-np$ | 1 | Number of MPI processors, as normally used with mpirun. |
| $-t$ or $--num\_threads$ | 1 | Number of threads per processor. |
| $-i$ | in.filename | Configuration parameters. If not specified the program will use file name in.lj.miniMD or in.eam.miniMD for LJ and EAM, respectively. |
| $-b$ or $--neigh\_bins$ | 0 | Number of neighbour bins. If set to zero or not used, the program will compute the value. |
| $--half\_neigh$ | 0 | Force neighbour list type; set to 0 or 1 for full or half, respectively. |
| $--ghost\_newton$ | 0 | Set usage of Newton's Third Law for ghost particles. |

**Table 5.1: Key run settings that can be used to set the program execution.**

In addition, due to the different types of compilers, especially those for hardware lacking on-chip built-in floating-point capability, there are options to choose float or double data types for floating-point values. This also enables the application to be set for single or double precision levels of computation.

Managing large dynamic arrays requires careful memory allocation and alignment, which improves memory access, especially if the adopted alignment size is compatible with the cache line size. Padding is also used when shaping the data structure from a structure of arrays into an array of structures, which has been found to yield better data locality and enhance computational performance.

The physics and technical settings can be specified in the input file, as shown in Table 5.2.

| Parameter Name | Value (e.g.) | Description |
|---|---|---|
| Force type | LJ | Defines force field of interaction: either LJ or EAM |
| Data File | None | Name of the data input file. None implies to generate particles. |
| $\epsilon$ and $\sigma$ | 1 1 | Values of $\epsilon$ and $\sigma$ in LJ potential. |
| System size | 32 32 32 | Simulation unit cell size of problem. |
| Time steps | 100 | Number of simulation time steps to run for. |
| Time step size | 0.005 | Integration time step $\Delta t$. |
| Initial temperature | 1.44 | Initial simulation temperature. |
| Density | 0.844200 | Density ($\rho$) of the particles in simulation box |
| Neighbour frequency | 20 | Number of time steps between reconstruction of neighbour lists. |
| Force cut-off and skin | 2.5 0.30 | LJ cut-off distance and the skin added to the cut-off $r_0 = r_c + r_s$. |
| Thermodynamic frequency | 100 | Number of time steps between updates to the thermodynamic system. |
| Sorting frequency | 20 | Optional, default set equal to neighbour frequency for optimal execution time. |

**Table 5.2: Sample of definition of various simulation parameters in LJ units.**

The input module uses the setup module to parse the input file and to set up the simulation box. In cases where more than one process is used in the MPI implementation, the simulation box is divided among a grid of processes and each process has its own simulation box boundaries, based on its location in the grid and the computed value of the lattice spacing, $(4/\rho)^{1/3}$. The lattice, in turn, is used to define the bin dimension.

Particles and their velocity can be randomly generated according to the method proposed by Park and Miller and as in [116, Box 1]. As the equilibrium properties of the simulation system do not rely on the initial conditions, all reasonable choices of initial conditions are, in principle, acceptable [50, p 40]. The boundary of the processor internal box, or the simulation box boundary in cases of one processor, is also taken into account. In cases where the input files are defined according to the LAMMPS format [1], they are parsed by specific functions that are defined to support LAMMPS input and data files.

### 5.2.2 Atom Module

The atom module is mainly concerned with how to store the particles' position, velocity, force and type, and manages the particle data structure. It also enforces periodic boundary conditions when particles move according to the internally defined simulation box. It also packs and unpacks the particles that are exchanged among processors through the communication module. In addition, it sorts the particles in each coordinate direction, according to their allocation in the bins constructed in the neighbour module. Algorithm 5.1 illustrates how the module ensures that particles that migrate outside the simulation box are placed back in the simulation box. In the case of multiple processes, when particles migrate from one local process box to the local box of another process, the communication module (see Section 5.2.5) enforces periodic boundary conditions through data exchanges between the processes. Another key function of the atom module is to sort the particles according to the ordering of the bins. Each bin in three dimensions has a coordinate value ($x$,$y$, and $z$) and these values are used to define the sequential label (a single index) in the ordering used, as will be described in the next chapter. Therefore, each particle in the atom array is examined to determine which bin it is located in. The ordered bins are used, in turn, to sort the particle data in the position, force, velocity and type arrays.

**Algorithm 5.1:** pbc: forces all the particles to be within the simulation box.

```
Function pbc()
    Input: box and x :stores particles coordinates.
    Output: x.
    // for each particle check their coordination.
    foreach particle in x do
        // enforce x-boundaries
        if x[i*PAD+0] < 0.0 then x[i*PAD+0]+=xprd
        if x[i*PAD+0] >xprd then x[i*PAD+0]-=xprd
        // enforce y-boundaries
        if x[i*PAD+1] < 0.0 then x[i*PAD+1]+=yprd
        if x[i*PAD+1] >xprd then x[i*PAD+1]-=yprd
        // enforce z-boundaries
        if x[i*PAD+2] < 0.0 then x[i*PAD+2]+=zprd
        if x[i*PAD+2] >xprd then x[i*PAD+2]-=zprd
    end
end
```

### 5.2.3 Neighbour Module

The neighbour module is the core component of the MDS where neighbour lists are managed. Thus, its core functionality is to assign each particle to the corresponding bin according to its position in the simulation box. However, before doing that, the active and ghost cells must be defined. Therefore, if the number of bins in each dimension is not part of the system configuration input then by default it is set to 5/6 of the simulation box size in each dimension. Then, the bin dimensions are computed by dividing the simulation box dimension by the number of bins in each direction ($x$, $y$, and $z$), which is handled by the *setup* method of this module.

The ghost bins are constructed around the simulation box by using the offset of the lower edge position (for each direction) minus $r_0$ and a very small value relative to the overall box length (in the corresponding direction), for round-off safety. The upper edge position is added to $r_0$ and a very small value relative to the overall box length. In each direction, the lower ghost bin boundary is subtracted from the upper ghost bin boundary and the result is divided by the bin length to give the number of bins. This may be adjusted to ensure full coverage of the simulation box.

In order to find the number of bins in the stencil, a stencil is constructed by finding all the bins whose closest corner to the central bin is within the cut-off ($r_0$). The stencil array stores only the offsets of the neighbours, which can then be applied for any bin to find the corresponding neighbour bin numbers. If half neighbour lists are used the stencil covers only those bins to the upper right of the central bin and does not include the central bin itself, as shown in Algorithm 5.2.

**Algorithm 5.2:** `buildStencil`: the algorithm used to build and generate the stencil array. It takes into consideration the full or half neighbour-list configuration as the number of elements can be reduced by half compared to the full neighbour-list. `binDistance` returns the distance from the central bin to the bin $(i, j, k)$ .

```
Function buildStencil(gw, mbins)
    Input: gw: number of ghost bins, and mbins : number of bins in one direction
           assuming (assuming mbinx = mbiny = mbinz)
    Output: stencil.
    $nmax = (2 * gw + 1) * (2 * gw + 1) * (2 * gw + 1)$
    $stencil \leftarrow [0, 1, \ldots, nmax]$ // create stencil array of size nmax
    $nstencil = 0$
    for $k = -gw;\ k \leq gw;\ k++$ do
        for $j = -gw;\ j \leq gw;\ j++$ do
            for $i = -gw;\ i \leq gw;\ i++$ do
                // if force type full consider all, otherwise only the
                   upper right bins
                if not(half_neight) or (k>0 or j>0 or (j=0 and i>0 )) then
                    // check if the distance is within $r_0$
                    if (binDistance (i,j,k) < $r_0$ ) then
                        $stencil[nstencil++] = i + mbins * (j + mbins * k)$
                    end
                end
            end
        end
    end
    return stencil
end
```

Particles are assigned to bins using the `binatoms` function that calculates a particle's position relative to the grid of bins. The bin array stores the index of the particle and another array stores the number of particles that have been added to each bin. These bins and the constructed stencil are used to build the neighbour list (in the *neighbors* array) for each particle. The method examines each particle *i* to find its bin index,

$bin_i$, and then uses the stencil to find all the neighbour bins. Each neighbour bin, $bin_j$, can be itself, $bin_j = bin_i$, or a different bin, $bin_i \neq bin_j$; in both cases all the particles in $bin_j$ are considered neighbours of particle, $i$, if they are within distance $r_0$. Finally, all the neighbours of each particle are stored in an array *neighbour* and their type is also stored in the *ntypes* array. Each particle has the same maximum number of neighbours and, therefore, *neighbour*[0] indicates the first neighbour in the list for particle 0. In cases where the total number of neighbour particles is greater than the initial defined maximum size of the neighbour array, then the maximum size of the neighbour lists and their types are redefined by multiplying the old maximum number of neighbour particles by two and then reconstructing the neighbour list again. This process is repeated until the defined size is sufficient to store all the neighbours.

### 5.2.4 Force Module

The force module is mainly concerned with the computation of the LJ and EAM interaction forces. In this research, we focus only on the LJ force computation. As mentioned above, the force can be computed with full or half neighbour list options. The advantage of the full neighbour force computation is that it avoids the need to check if a neighbour particle position is to the upper left of the central bin of the particle under consideration, as must be done in the half neighbour list force computation. However, the advantage of the half neighbour list option is that the number of neighbour particles involved in the computation is approximately half the number of particles that are involved in the full neighbour list.

For the full neighbour list case, the potential energy is computed by, first, initiating all the interaction force arrays of each particle and the overall potential energy to zero and then passing through all the particles in the system by using their position and their corresponding neighbour particles. The first step is to find the squared distance between the particle, $i$, and each neighbour $j$, in the neighbour list. If the square of the computed distance is less than $r_0^2$ then the interaction force between the particles can be computed by using a reduced form of Equation 2.24 as in Equation 5.1, see [50, p. 69] for details.

$$f_x = \frac{\partial E(r)}{\partial x} = -\left(\frac{x}{r}\right)\left(\frac{\partial E(r)}{\partial r}\right) = \frac{48x}{r^2}\left(\frac{1}{r^{12}} - 0.5\frac{1}{r^6}\right) \tag{5.1}$$

where $x$ represents the distance and the interaction between particles $i$ and $j$ that contributed to the potential energy, which implies that $x$ is equal to $\epsilon$. $r$ in this equation is the distance between the two particles. Algorithm 5.3, shows a simplified algorithm for both full and half force computation that will be described in more detail in Chapter 6 on the GPU implementation.

Based on the type of neighbour list used and how ghost particles are handled the appropriate force computation is executed. There is a minor difference between these different implementations, such that in the half neighbour list case the position of each neighbour particle, $j$, is checked to ensure it is to the upper right of particle, $i$. The difference in the half neighbour list case for MPI only, and MPI with OpenMP threading, is that in the latter case in internal force computation, the energy and virial pressure updates must be done atomically.

### 5.2.5 Communication Module

The communication module is concerned mainly with the spatial decomposition of the simulation domain over a grid of processes and the communication of data between adjacent processes in the grid. Each process has its own portion of the simulation box termed the internal, or local, box to distinguish it from the global simulation box. Consequently, we use a 3-dimensional process grid of size $K \times M \times N$, where the simulation dimension in bins must be divisible by the process grid size in each direction.

The particles that migrate outside their local box are flagged to be moved to a neighbour process together with the corresponding ghost particles. With this approach, the update between the neighbouring processes takes place only if the swap flag is true,

**Algorithm 5.3:** `compFullNeigh`: shows the computation for full neighbour list of the interaction forces and the potential energy. In a multithreading implementation an `Atomic` region is defined to ensure there is no race condition among the threads. This algorithm is optimized for MPI and OpenMPI (threading).

```
Function compFullNeigh(atom, neighbour)
    Input: atom: stores atoms coordinations, and neighbour : neighbour list particles
    Output: force:array of force imposed on each particle.
    $f = atom.f$
    $x = atom.x$
    $nlocal = atom.nlocal$
    $en = 0$
    for $i = 0;\ k \leq atom.nlocal;\ i++$ do                    // Initialize force array to zero
        $f[i * PAD + 0] = f[i * PAD + 1] = f[i * PAD + 2] = 0$
    end
    foreach particle i in x do
        $neigh = neighbour.neighbours[i * maxneighs]$
        $numOfNeigh = neighbour.numneigh[i]$
        $fix = fiy = fiz = 0$
        $xtmp = x[i * PAD + 0]$
        $ytmp = x[i * PAD + 1]$
        $ztmp = x[i * PAD + 2]$
        for $k = 0; k < numOfNeigh; k++$ do
            $j = neigh[k]$                                       // index of a neighbour particle
            $\Delta x = xtmp - x[j * PAD + 0]$
            $\Delta y = ytmp - x[j * PAD + 1]$
            $\Delta z = xtmp - x[j * PAD + 2]$
            $rsq = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2$
            if $rsq<cutforcesq$ then                             // check if $r_{ij}^2 < r_0^2$
                $sr2 = 1.0/rsq$
                $sr6 = sr2^3 * \sigma^6$
                $force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * \epsilon$
                $fix+ = \Delta x * force$
                $fiy+ = \Delta y * force$
                $fiz+ = \Delta z * force$
                /* EVFLAG is true if evaluation nstate % thermoState is 0          */
                if $EVFLAG$ then
                    $t\_eng\_dwl+ = sr6 * (sr6 - 1.0) * \epsilon$
                    $t\_virial+ = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2) * force$
                end
            end
        end
        $f[i * PAD + 0]+ = fix$
        $f[i * PAD + 1]+ = fiy$
        $f[i * PAD + 2]+ = fiz$
    end
    $t\_eng\_dwl+ = sr6 * (sr6 - 1.0)* = 4.0;$
    $t\_virial* = 0.5;$
    Atomic($eng\_vdwl+ = t\_eng\_vdwl$)
    Atomic($virial+ = t\_virial$)
end
```

which reduces the required amount of communication among the processes. Thus, the periodic boundary conditions are taken into consideration even at the local box level.

In our work, we do not use more than one MPI process so most of the communication module functionality will not be called during a simulation.

### 5.2.6 Thermodynamics Module

The thermodynamics modules handles the computation of the temperature, pressure and energy. The thermodynamic module sets up the parameters and scalar values required for the computation. The thermodynamics module initially performs the temperature calculation, which is proportional to the square of the particle velocity multiplied by its mass, summed over all particles:

$$T = \frac{1}{T_{scale}} \sum_{i=0}^{n-1} m_i v_i^2 \tag{5.2}$$

where $T_{scale}$ is a temperature scaling factor. Next, the energy computation is performed using the potential energy calculated by the force module. Finally, the pressure is calculated by using the pressure computed in the force module multiplied by the pressure scale factor and the number of degrees of freedom, as defined in the setup and initialization of this module.

### 5.2.7 Integration Module

The integration module is at the core of the simulation system, and uses the run method to integrate the equations of motion using the particle forces computed by the force module. The atom module is then used to advance the position and velocity of each particle. In addition to the run method, the integration module contains the initial method that sets up the initial particle positions and velocities, and the integrate method that is called at the end of each iteration to set the final velocity values based on the particle's interaction force, that is computed earlier by the force module.

At each time step, $n$, Algorithm 5.4 starts by copying the arrays for the position, $x$, and old position, *xold*, the velocity, $v$, and the interaction forces, $f$, as well as the total number of particles, *nlocal*. An initial integration then does the first step to advance the particles' velocity and position. If necessary, particles are next exchanged among pro-

**Algorithm 5.4:** run: the main function to run the MD simulation by calling appropriate functions every time step.

```
Function run(atom, neighbor, comm, thermo, timer)
    Input: atom: instance of Atom class, neighbor:instance of Neighbor class, comm instance of Comm class,
           thermo:instance of Thermo class, and timer: instance of Timer class.
    Output: updated particles coordination,forces, velocity, potential energy, thermodynamic parameters

    $nextSort = sortEvery > 0?sortEvery : ntimes + 1$
    for $n = 0;\ n \le ntimes;\ n++$ do
        $x = atom.x$
        $v = atom.v$
        $f = atom.f$
        $xold = atom.xold$
        $nlocal = atom.nlocal$
        // Initialize integration parameters
        initialIntegrate ()                                  // call function initial integrate
        if $(n + 1)$ mod $neighbor.every$ then // is it re-neighbouring time?
            comm::communicate (atom)                         // call function communicate
        else
            if $checkSafeExchange$ then
                $\Delta max = 0$
                /* for all particles,$i$.compute $\Delta$ of $x$,$y$ and $z$ to set $\Delta max$ value.   */
                $\Delta max$ =positionDelta ($\Delta max$,$\Delta x$,$\Delta y$,$\Delta z$, $atom.box$)
                $\Delta max = \sqrt{\Delta max}$
                // check particles are not moving beyond box boundry
                if $\Delta max$ beyond box boundry then print Warning
            comm::exchange (atom)                            // call function exchange
            if $n + 1 > nextSort$ then // is it time to sort?
                $nextSort += sortEvery$
                atom::sort (neighbour)                       // sort particles
            /* before neighbouring make lists atoms going to be exchanged   */
            comm::borders (atom)
            if $checkSafeExchange$ then // only for safe exchange case
                // copy computed $x$ coordinate into $oldx$
                foreach $i$ in atom do $xold[i] = x[i]$
            neighbor::build (atom)                           // binned neighbour list
        /* Set the flag to update energy and virial pressure according to nstate   */
        $force \rightarrow EVFLAG = (n + 1)$ mod $thermo.nstate == 0$
        force→compute (atom,neighbor, comm)                  // call compute forces
        if half_neigh and ghost_newton then
            comm::reverse_communicate (atom)                 // call function reverse communication
        $v = atom.v$
        $f = atom.f$
        $nlocal = atom.nlocal$
        finalIntegrate ()                                    // call fianl integration function
        if thermo.nstate then
            thermo::compute (n+1,atom,force,timer,comm)      // call function compute
```

cesses by the *communicate* method, which transfers migrating particles from the local box of a process to that of another process. If this communication is not performed, then a safe exchange check is done that issues a warning if any particle has moved outside the simulation box. The *exchange* method at each time step moves particles to the correct process boxes. Thus, any particle that has moved out of the local box of a process in any direction is placed into a corresponding communication buffer and sent to the adjacent process in that direction. At the same time, a process may also receive particles from other processes. With this approach, each process will exchange particle data with all six neighbours in the process grid. The next step is to check if the particles need to be sorted in the current time step. This ensures that the particles are sorted by bin order so that all the particles in bin $i$ are stored in the particle data arrays before those in bin $i + 1$. The *borders* method is then called to prepare a list of particles that must be communicated to other processes. This is similar to the *communicate* method, but is called every time step. If the safe exchange protocol is being used the particle positions are then copied into the *xold* array. Next, the *build* method of the `neighbour` class is invoked, in which the neighbour list for each particle is created. Then, the *force* method computes the interaction force on each particle and evaluates the potential energy and virial pressure. If half neighbour lists are being used, the *reverse_communicate* method is called which works similarly to the *communicate* method, except that the communication between the process is from a high rank process to a lower rank process. Newly-generated velocity and force values are copied into the $v$ and $f$ array that will be used by the *finalIntegrate* method to compute the new velocity values of each particle. Finally, if the thermostat modification is to be performed in the current time step, the *compute* method of the `thermo` class is called to update values of energy, pressure and temperature.

### 5.2.8 Timer Module

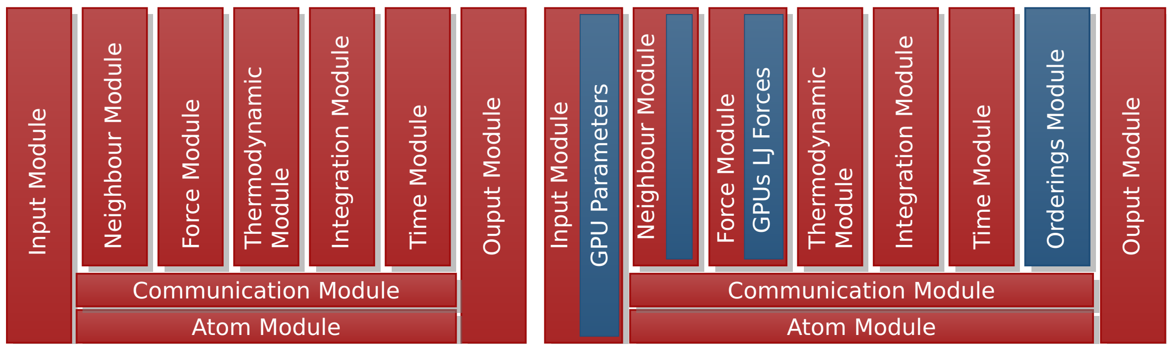
The timer module is used to record times for communication, force evaluation, building the neighbour lists, as well as the total time. Therefore, each module's *timer* method is called to update the corresponding timer variable by first initiating the timer and later stopping it at appropriate points in the code. The recorded times are then used by the output module to help to measure the performance of the simulation module and the time taken by each component of the application.

### 5.2.9 Output Module

The output module prints out the simulation parameters and the configuration parameters used during the simulation, as well as the performance results of the overall execution time and the time taken by each component of the application. If the *yaml_output* flag is enabled, then in addition to the standard output, thermodynamic details and a histogram of each component's execution time at each time step can be produced. In addition, it is possible to print out the position and the type of each particle and use a third-party application to visualise the particle movements during the simulation.

## 5.3 Implemented Modifications

Figure 5.1 illustrates the modifications introduced into the miniMD package in order to conduct this research. In this section, the new modules and the modifications carried out will be described in detail.



(a) (b)

**Figure 5.1: MiniMD before the modifications (a) and after the modifications (b).**

### 5.3.1 Ordering Module

We have incorporated an ordering module alongside the other existing modules in miniMD. The ordering module manages two arrays:

1. *ostore[i]*: this is the position in the ordering of the item at position *i* in a row-major ordering.

2. *invstore[i]*: this is the inverse mapping, i.e., the row-major index corresponding to position *i* in the ordering.

This approach reduces the number of calls to convert between a row-major ordering index to other orderings. These two arrays are evaluated once at the start of the program and are then used every time it is necessary to convert between a row-major index and a Hilbert or Morton index, rather than doing the conversion dynamically. In order to compute the *ostore* and *invstore* arrays the module needs to know the number of bins, $N \times N \times N$, where $N$ is the number of bins in each direction. Consequently, the number of bits that are used to identify the encoding level can be obtained by computing $\log_2 N$. The orderings that are supported in this module are summarised in Table 5.3, which enables users to select the desired ordering by using the flag *-o* or *--order* at execution time.

| Order number | Description |
|---|---|
| 0 | Row-major ordering |
| 1 | Hilbert ordering |
| 2 | Morton ordering |
| 3 | Hybrid Hilbert and row-major ordering |
| 4 | Hybrid Hilbert and column-major ordering |
| 5 | Hybrid Morton and row-major ordering |
| 6 | Hybrid Morton and column-major ordering |

**Table 5.3: Different orderings that are supported by the ordering module**

The row-major order is the conventional ordering of an array so the value of the entry is actually equal to the index of the array for both that *ostore* and *invstore* arrays, which are simply generated according to Algorithm 5.5.

**Algorithm 5.5:** `setRowMajor`: generates *ostore* and *invstore* arrays for a row-major ordering. $nbins = N \times N \times N$ is the total number of bins.

```
Function setRowMajor()
    Input: ostore and invstore arrays.
    Output: ostore and invstore values are generated.

    for $i = 0, p = ostore$ to $nbins$ do $*(p + i) = i$

    for $i = 0, p = invstore$ to $nbins$ do $*(p + i) = i$

end
```

A Hilbert ordering, as described in Section 2.3, is obtained by expressing the defined rules as shown in Algorithm 5.6. However, a small modification is necessary to store the computed value, $h$, in the ordering array, $ostore[ibin] = h$, and to use it as an index to store the *ibin* value in the inverse ordering array, $invstore[h] = ibin$. Despite there being various way to implement the Hilbert ordering, any adopted implementation does not affect the overall application performance because the mapping arrays are generated once before the MDS is initiated.

**Algorithm 5.6:** hilbert3D: generates Hilbert number $h$ and curve points $(px, py, pz)$ which are advanced according to Algorithm 5.7.

```
Function hilbert3D(d,h,px,py,pz,y,p,r,level)
    Input: d: direction matrix as defined in Equation 2.19; h integer number represents Hilbert index; (px, py, and pz)
           coordination of points of the Hilbert curve, p and r transformation matrices as defined in Equation 2.18; and
           level depth of the curve.
    Output: d, h and coordinates arrays ( px, py, pz )
    if level>=1 then
        $d = d * p * r$;                                   // pitch and rotate 90∘ initial matrix d
         // recursive call with $level - 1$
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position

        $d = d * p * r$;                                   // pitch and rotate 90∘.
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position

        $d = d * y'$;                                      // yaw −90∘
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position

        $d = d * p * r' * r'$                              // pitch 90∘ and than twice yaw −90∘
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position

        $d = d * p'$                                       // pitch −90∘
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position

        $d = d * y * r' * r'$                              // yaw 90∘ and than twice roll −90∘
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position

        $d = d * y'$                                       // yaw −90∘
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        cordUpdate(px,py,pz,d,h); $h++$                    // advance the point and position
        $d = d * r'$                                       // roll −90∘
        return hilbert3D(d,h,px,py,pz,y,p,r,level-1)
        $d = d * y' * r'$                                  // yaw and roll −90∘
    end
end
```

**Algorithm 5.7:** hilbert3D: updates points to move to the next location according to the orientation matrix.

```
Function cordUpdate(d,px,py,pz,h)
    Input: d: direction matrix as defined in Equation 2.19; h integer number represents Hilbert index; (px, py, and pz)
           coordination of a points.
    Output: px, py, pz are updated
    $px[h] = px[h-1] + d[0,0]$
    $py[h] = py[h-1] + d[1,0]$
    $pz[h] = py[h-1] + d[2,0]$
end
```

Similarly, a Morton ordering is constructed based on bit manipulations (as described in Section 2.3) and implemented as shown in Algorithm 5.8, which generates the Morton label and the inverse row-major ordering for each of the *nbins* bins. The algorithm first computes the 3-dimensional coordinates (the key values), which are then used as input for the dilation routine by moving the bits of each key to the correct position.

**Algorithm 5.8:** morton3D: Translates the row major index into the Morton label.

```
Function morton3D(i, n)
    Input: i is the major row index to be translated into Morton label, n length of an
           axis i.e. number of labels in one direction.
    Output: The corresponding Morton label.
    $x = i\%n$                                          // get x value.
    $y = (i\%(n*n))/n$                                  // get y value.
    $z = i/(n*n)$                                       // get z value.
    return (dilate3D($x$) + dilate3D($y$«1) + dilate3D($z$«2))
end
```

**Algorithm 5.9:** morton3D: Using bit manipulation to dilate the input value in 3D.

```
Function dilate3D(k)
    Input: k stores the input number going to be dilated by $t = 3$.
    Output: x represents the dilated number.
    // Initialization
    $x = 0, t = 0, y = k$
    while $y$ do
        int $b = y\&1$                          // copy the first bit
        $x+ = b << t$           // Shift bit value by $t$ and then
                                // added it with previous $x$ value.
        $t+ = 3$            // Advance $t$ according to the dimension size.
        $y = y >> 1$     // Remove processed bit to process the next bit.
    end
    return ($x$)
end
```

A hybrid ordering, as defined in this research, uses row-major or column-major ordering to sort the elements within each sub-block, and then these sub-blocks are organized by using either a Hilbert or Morton ordering, as explained in Section 2.3.4. In other words, a hybrid ordering can be visualised as having two levels: the first is the sub-block level, and the second level shows a sub-block containing a number of elements (bins) which are ordered internally using row-major ordering. The ordering between sub-blocks is maintained by linking the last bin in a sub-block to the first bin in the next sub-block.

Algorithm 5.10 shows how the hybrid Hilbert/row-major ordering is constructed. Similarly, Algorithm 5.11 shows the generation of a hybrid Morton/row-major ordering. The hybrid ordering with column-major order within sub-blocks is very similar, and therefore we do not present it here. In this work, we set the size of a sub-block to 4, and so $t = 2$. Therefore there are $4^3 = 64$ bins in each sub-block. This size was chosen to set the size covering the number of neighbour bins, and because most of the bins within the sub-blocks have a neighbouring bin that belongs to a neighbouring sub-block. The second reason is that this size has high potential to ensure that the bin indexes of a sub-block are fully copied into the higher cache levels.

**Algorithm 5.10:** `setHybridHilbertRM`: generates hybrid Hilbert and row-major ordering and the array inverse values.

```
Function setHybridHilbertRM()
    Input: ostore and invstore arrays.
    Output: ostore and invstore values are generated.

    $logm = logn - logt$                      // number of bits of sub-block representations
    $widthSBlocks = (1 << logm)$                   // number of sub-blocks in one dimension
    $numSBlocks = (1 << (3 * logm))$                       // total number of sub blocks
    $widthBins = (1 << logt)$               // number of bins in one dimension within sub-block
    $numBins = (1 << (3 * logt))$                  // total number of bins inside each sub-block

    $temp$ = hilbert3D(..., $level=logm$)          // generate Hilbert ordering for sub-blocks level
    for $i = 0$ to $numSBlocks$ do
        // Initial sub-blocks coordinate in relative to the bins coordination
        $startX = (i \mod widthSBlocks) * widthBins$
        $startY = ((i \mod widthSBlocks^2)/widthSBlocks) * widthBins$
        $startZ = (i/widthSBlocks^2) * widthBins$
        // Internal sub-blocks bins labeling
        for $bIdx = 0$ to $numBins$ do
            $x = bIdx \mod widthBins$
            $y = (bIdx \mod widthBins^2)/widthBins$
            $z = (bIdx/widthBins^2)$

            $rmIdx = (startZ + z) * n * n + (startY + y) * n + (startX + x)$
            $hilValue = temp[i] * numBins + z * widthBins^2 + y * widthBins + x$
            $ostore[rmIdx] = hilValue$
            $invstore[hilValue] = rmIdx$
        end
    end

end
```

## Incorporating the Ordering Module

The ordering module is initialized from the neighbour module where the ordering configuration is defined as part of the execution parameters. The number of bins generated for a Hilbert of Morton ordering must be an exact power of two, and so may differ from the number of bins that are generated in the row-major ordering case. This, in turn, increases the number of bins required but does not affect the ordering and the locations of the bins in memory. In other words, more space will be required but those

**Algorithm 5.11:** `setHybridMortonRM`: generates hybrid Morton and row-major ordering by using the `dilate3D` function that is used for Morton ordering.

```
Function setHybridMortonRM()
    Input: $n$, $ostore$ and $invstore$ arrays.
    Output: $ostore$ and $invstore$ values are generated.
    $nbins = n^3$
    $widthSBlocks = (1 << logt)$
                                        // number of bins in one dimension within sub-block
    $totalSBBins = (1 << (3 * logt))$          // total number of bins in each sub-block
    for $i = 0,\ p = ostore\ ;\ i < nbin;\ i++,\ p++$ do
        $x = i \mod n$
        $y = (i \mod n^2)/n$
        $z = i/(n^2)$
        // set the sub-block label first.
        $currSBlock$ = dilate3D( $(int)\ x/widthSBlock$ ) + dilate3D( $((int)y/widthSBlock) \ll 1$ )
                                        +dilate3D( $((int)z/widthSBlock) \ll 2$ )

        $*p = currBlock * totalSBBins + (z \mod widthSBlocks) * widthSBlocks^2$
                            $+(y \mod widthSBlocks) * widthSBlocks + (x \mod widthSBlocks)$
        $invstore[*p] = i$
    end
end
```

bins are kept in consecutive order. Therefore, in the neighbour module, particularly in `neighbour::setup`, where the number of bins in each dimension is computed, we check which dimension has the largest number of bins, even though it is assumed the simulation size is always equal in all directions. Then, this is used to compute the number of bits required to represent that number of bins in our bit manipulation module. Therefore, if the selected ordering is row-major, the number of bins generated will be according to the standard implementation, $mbinx \times mbiny \times mbinz$, otherwise it will be $2^{3*mbits}$, where $mbits$ is the number of bits computed to represent the largest number of bins in any dimension. Finally, the ordering module is initialized and the *ostore* and *invstore* arrays are generated.

In miniMD particles are allocated to bins by the *coord2bin* method of the `neighbour` class. In our implementation this is modified to convert the bins into the desired order by using the computed row-major order as an index to return the value $ostore[index]$. On the other hand, processing the neighbour list requires converting an index in the defined ordering to a row-major index in order to select the bins in the stencil without the need to construct a specific stencil function for each ordering. Therefore, the bin label is converted into a row-major index using the *invstore* array, and the stencil bin is added. This gives the row-major index of the stencil bin, which is then converted into an index in the defined ordering by: $ostore[invstore[ibin] + stencil[k]]$. As mentioned previously, this approach of using the two arrays for encoding and decoding between row-major ordering and any other ordering minimises the amount of computation in performing index conversions.

### 5.3.2 Force Computation on the GPU

Full and half neighbour force computations are specified as kernel functions that are called from the host (CPU) side of the CUDA implementation. As the GPU functions cannot directly access the host memory, or vice versa, we used paged memory, rather than pinned or unified memory, to minimize the need to modify any miniMD data structures and to allow the GPU to support large simulation sizes.

The standard force computation has been modified to check an input value in the configuration file to determine if the force computation should be performed in the standard way (on the host only) or on the GPU. If the value is less than zero, the force computation is done on the host. If the value equals zero, the force computation is done on the GPU and the program sets the thread block size, as computed internally by the CUDA API, to give the best theoretical warp occupancy. Otherwise the value is used as the thread block size for GPU execution. In addition, a CUDA compute function is added to the standard force module to set the parameters used to decide the type of force computation to call on the GPU, according to a value specified in the configuration file. If the value is 0, then the full neighbour list force computation is performed; otherwise, the half neighbour list force computation is executed. The execution on the GPU of the force computation kernel is managed by the function `cudaLJForceCompute` called on the host. This function is responsible for selecting the GPU device, computing the thread block and grid sizes, computing the maximum theoretical occupancy, allocating arrays in device memory, preparing all the required parameters, copying the data needed for the execution of the force kernel, initiating kernel execution, measuring the time for kernel execution, and summing the potential energy and virial pressure computed by the thread blocks. All these tasks are summarised in Algorithm 5.12.

**Algorithm 5.12:** `cudaLJForceCompute`: prepares the required parameters and copies the data between host and device and vice versa. The function records the time taken by `devHalfNeigh` or `devFullNeigh`

```
Function cudaLJForceCompute(...)
    Input: ostore and invstore arrays.
    Output: ostore and invstore values are generated.

    Define required parameter
    cudaSetDevice(0)                                              // select GPU device
    if neigh_half then
        cudaOccupMaxPotBlockSize(&sugGS, &sugBS, dev_half_neigh, dySMemUsage, nlocal)
    else
        cudaOccupMaxPotBlockSize(&sugGS, &sugBS, dev_full_neigh, dySMemUsage, nlocal)
    end
    $blockSize = (BSize > 0)?BSize : sugBS$
    $gridSize = (nlocal + blockSize - 1)/blockSize$

    Allocate device memory and
    copy the data from host to device ($host \rightarrow device$) for all the required parameters

    if neigh_half then
        cudaEventRecord(&start)                          // create events and record start time
        devHalfNeigh <<< gridSize, blockSize >>> (parameters)
    else
        cudaEventRecord(&start)
                                                         // create events and record start time
        devFullNeigh <<< gridSize, blockSize >>> (parameters)
    end
    cudaDeviceSynchronize()     // Synchronize the device to ensure all the block are done!.
    cudaEventRecord(&stop)                                                // record stop time

    cudaEventElapsedTime(&elapsedTime, start, stop)             // calculate execution time
    Copy back all the data from device to host $device \rightarrow host$.
    for $i = 0$ to gridSize do
        $energy += partialEnergy[i]$
        $virial += partialViria[i]$
    end
    Free all the variables created and reset device.
end
```

### Full Neighbour List Implementation of the Force Kernel

The full neighbour list force kernel is shown in Algorithm 5.13, where it can be seen that each thread is first assigned a specific particle, *Idx*, according to its location in the thread grid (based on the thread's block index and index within the block). The total number of threads could be more than the total number of particles so each thread must check that its *Idx* value does not exceed the total number of particles, *nlocal*. The thread then copies the particle position information and stores it in registers (if available). Local variables are used to store the interaction forces, potential energy and virial pressure, and these are initialised to 0.

The `for` loop iterates over the particles in the neighbour list of particle *Idx*. The distance between the particle and the neighbour list particle is found and if it is within the cut-off distance, $r_0$, then the force between the two particles is evaluated according to the Lennard-Jones potential and this force is added to the force on particle *Idx*, as in Equation 5.1. If the distance between the particles exceeds the cut-off distance then the interaction makes no contribution to the force, and subsequently the neighbour particle will be removed from the neighbour list of particle *Idx*. The potential energy and virial pressure are accumulated for each thread block using atomic operations. The atomic function `cudaAtomicAdd` has been implemented to support atomic operations on the data types of the potential energy and virial pressure. This was necessary because the built-in CUDA atomic functions were not compatible with the compute capability of the GeForce GTX 960 GPU used in the performance experiments discussed in Chapter 6.

**Algorithm 5.13:** `devFullNeigh`: shows the computation for the full neighbour list case of the interaction forces and the potential energy. A potential race condition for potential energy and virial pressure is avoided using the `cudaAtomicAdd` function.

```
Function devFullNeigh(...)
    Input: nlocal number of particles, dev_x coordinations, dev_f: forces, dev_ntypes number of particles type ,
           dev_type particles type, dev_neigh neighbours list, dev_numNeigh, dev_partEng block's potential energy,
           dev_partVirial block's virial, $\sigma_{dev}$, $\epsilon_{dev}$
    Output: force: computed interaction forces, blocks potential energy, blocks potential virial pressure.

    $Idx = blockIdx.x * blockDim.x + threadIdx.x$
    if $Idx < nlocal$ then
        $neigh = dev\_neighbours[Idx * maxneighs]$
        $numOfNeigh = dev\_numNeigh[Idx]$

        $xtmp = dev\_x[Idx * PAD + 0]$
        $ytmp = dev\_x[Idx * PAD + 1]$
        $ztmp = dev\_x[Idx * PAD + 2]$

        $fix = fiy = fiz = 0$
        $t\_eng\_vdwl = t\_virial = 0$

        for $k = 0$ to $numOfNeigh$ do
            $j = dev\_neigh[k]$                                   // index of a neighbour particle
            $\Delta x = xtmp - dev\_x[j * PAD + 0]$
            $\Delta y = ytmp - dev\_x[j * PAD + 1]$
            $\Delta z = xtmp - dev\_x[j * PAD + 2]$
            $rsq = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2$
            if $rsq<cutforcesq$ then                               // check if $r_{ij}^2 < r_0^2$
                $sr2 = 1.0/rsq$
                $sr6 = sr2^3 * \sigma_{dev}^6$
                $force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * \epsilon_{dev}$

                $fix+ = \Delta x * force$
                $fiy+ = \Delta y * force$
                $fiz+ = \Delta z * force$
                // $EVFLAG$ is true if evaluation nstate % thermoStae is 0
                if $EVFLAG$ then
                    $t\_eng\_dwl+ = sr6 * (sr6 - 1.0) * \epsilon_{dev}$
                    $t\_virial+ = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2) * force$
                end
            end
        end
        cudaAtomicAdd($dev\_partEng, t\_eng\_vdwl$)
        cudaAtomicAdd($virial+ = t\_virial$)
    end
end
```

### Half Neighbour List Implementation of the Force Kernel

In the half neighbour list case the neighbour list of a particle is constructed to contain only particles in stencil bins above and to the right of the central stencil bin, plus those particles in the central bin that are above and to the right of the particle in question. Thus, the operation count and the storage required for the neighbour lists is approximately half that of the full neighbour list case. The force computation in the half neighbour list case is carried out on the GPU as shown in Algorithm 5.13. As in the full neighbour list case, each thread is responsible for computing the interaction force between one particle and the particles in its neighbour list, and the general structure of the algorithm is the same in both cases.

When the force between particle *Idx* and a particle in its neighbour list is evaluated it is added to the total force on both particles. Since multiple threads can concurrently update these elements in the force array this results in a race condition. Thus, the atomic function `cudaAtomicAdd` is used to accumulate the force for neighbour list particles.

**Algorithm 5.14:** `devHalfNeigh`: computing the inter-particle forces in the half neighbour list case.

```
Function devHalfNeigh(...)
    Input: $nlocal$ number of particles, $dev\_x$ coordinations, $dev\_f$: forces, $dev\_ntypes$ number of particles type ,
        $dev\_type$ particles type, $dev\_neigh$ neighbours list, $dev\_numNeigh$, $dev\_partEng$ block's potential energy,
        $dev\_partVirial$ block's virial, $\sigma_{dev}$, $\epsilon_{dev}$
    Output: $force$: computed interaction forces, blocks potential energy, blocks potential virial pressure.

    $Idx = blockIdx.x * blockDim.x + threadIdx.x$
    if $Idx < nlocal$ then
        $neigh = dev\_neighbours[Idx * maxneighs]$
        $numOfNeigh = dev\_numNeigh[Idx]$

        $xtmp = dev\_x[Idx * PAD + 0]$
        $ytmp = dev\_x[Idx * PAD + 1]$
        $ztmp = dev\_x[Idx * PAD + 2]$

        $fix = fiy = fiz = 0$
        $t\_eng\_vdwl = t\_virial = 0$

        for $k = 0$ to $numOfNeigh$ do
            $j = dev\_neigh[k]$                                  // index of a neighbour particle
            $\Delta x = xtmp - dev\_x[j * PAD + 0]$
            $\Delta y = ytmp - dev\_x[j * PAD + 1]$
            $\Delta z = xtmp - dev\_x[j * PAD + 2]$
            $rsq = (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2$
            if $rsq<cutforcesq$ then                                    // check if $r_{ij}^2 < r_0^2$
                $sr2 = 1.0/rsq$
                $sr6 = sr2^3 * \sigma_{dev}^6$
                $force = 48.0 * sr6 * (sr6 - 0.5) * sr2 * \epsilon_{dev}$

                $fix+ = \Delta x * force$
                $fiy+ = \Delta y * force$
                $fiz+ = \Delta z * force$
                // Check Ghost Newton applied or j<nlocal
                if $GHOST\_NEWTON$ or $j < nlocal$ then
                    cudaAtomicAdd($\&dev\_f[j * PAD + 0], -1.0 * (\Delta x * force)$)
                    cudaAtomicAdd($\&dev\_f[j * PAD + 1], -1.0 * (\Delta y * force)$)
                    cudaAtomicAdd($\&dev\_f[j * PAD + 2], -1.0 * (\Delta z * force)$)
                end
                // $EVFLAG$ is true if evaluation nstate % thermoStae is 0
                if $EVFLAG$ then
                    $scale = (GHOST\_NEWTON$ or $(\mathbf{j} < \mathbf{nlocal}))?\mathbf{1.0} : \mathbf{0.5}$
                    $t\_eng\_dwl+ = scale * sr6 * (sr6 - 1.0) * \epsilon_{dev}$
                    $t\_virial+ = scale * (\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2) * force$
                end
            end
        end
        cudaAtomicAdd($dev\_partEng, t\_eng\_vdwl$)
        cudaAtomicAdd($virial+ = t\_virial$)
    end
end
```

## 5.3.3 Implementation Analysis

The implementation details presented in this chapter can be related to the description of molecular dynamics simulations provided in the earlier chapters of this dissertation.

If we assume a particle with array index, $i$, has a neighbour list $\{j_0, j_1, \ldots, j_{maxNeigh}\}$, and is located in $bin_i$, then:

- Consecutively indexed threads in a block process consecutively indexed particles. This is aligned with the best practice guidelines provided in Chapter 2. Moreover, long latency accesses to the particle data arrays is minimized by copying its data into registers, where appropriate.
- As a result of the particles being ordered in the particle data arrays according to their bin index, the particles in the same bin are numbered consecutively, and have similar neighbour lists. Hence, if the cache is large enough, the neighbour list will be loaded once into cache for the consecutive particles that belong to the same bin, as has been already addressed in detail in Chapter 4. Another factor in addition to cache size is the cache line size and its alignment with the maximum number of neighbours per bin. In the half neighbour list force computation, one of the main performance implications is that the neighbour list will be evicted from the cache because of the space required to load the force data for the neighbour particles.
- Experiments and analysis will be required to evaluate the trade-off between the number of neighbours that are processed, which is larger in the full neighbour list case, and the extra conditional branches, which cause warp divergence issues, in the half neighbour list case. Also the additional atomic accesses in the half neighbour list case for the interaction force computation serializes the threads within a warp, and this can also impact the execution time.

## 5.4 Summary

This chapter has covered our GPU implementation of the miniMD application, and the modifications that have been introduced to achieve this. The modifications are mainly related to the incorporation of the ordering module into the miniMD application, and the GPU implementation of the full and half neighbour list force computations. These modifications have enabled us to conduct our experiments and analysis according to our research requirements, as detailed in the next chapter.

*Chapter 6*

# Performance Experiments, Results and Analysis

This chapter describes the performance experiments that have been carried out, and provides an analysis of the research results. Two different hardware platforms have been used, and the number of time steps, the simulation size, and the size and number of thread blocks have been varied. All these parameters are used to analyse the data locality properties for each of the orderings investigated in this work.

## 6.1 Experimental Objectives

The focus of these experiments is to analyse how the data orderings (as described in Chapter 4) affect the data locality properties of the miniMD molecular dynamics application. The impact of the data orderings on the execution time, and their relation to the cache constraints and size, are studied by using different execution parameters and using the NVidia profiler to collect more detailed information about GPU metrics to understand the relationship between the parameters/configurations and the data locality properties of the application.

## 6.2 Hardware for the Experiments

In this research two hardware platforms were used: Ruthenium and Hawk. Ruthenium is a desktop computer, and Hawk is a GPU-enabled node of a cluster. They have different GPUs installed, as mentioned in Chapter 2 and summarised in Table 6.1.

| | **Ruthenium** | **Hawk** |
|---|---|---|
| Operating System | Ubuntu v14.04.5 LTS (Trusty Tahr) | Red Hat Enterprise Linux Server v7.4 (Maipo) |
| Model name | Inter(R) Xeon (R) i7-5960x CPU @ 3.00 GHz | Inter(R) Xeon (R) Gold 6148 CPU @ 2.40 GHz |
| GPU Model | Maxwell GeForce GTX 960 | Tesla P100 |
| Number of Processors | 16 | 40 |
| Cores per Sockets | 8 | 20 |
| Instruction Cache Size | 32KB | 32KB |
| L1 Data Cache Size | 32KB | 32KB |
| L1 Way Associative | 8 | 8 |
| L1 Cache Line Size | 64 Byte | 64 Byte |
| L2 Data Cache Size | 256KB | 1MB |
| L2 Way Associative | 8 | 16 |
| L2 Cache Line Size | 64 | 64 |
| L3 Data Cache Size | 20MB | 27.5MB |
| L3 Way Associative | 20 | 11 |
| L3 Cache Line Size | 64 | 64 |

**Table 6.1: Ruthenium and Hawk specifications.**

Both machines have three levels of cache, however, they are different in terms of size and latency. For example, the Hawk machine has a larger L2 cache, which could result in a lower miss hit rate compared to the Ruthenium machine.

The GPU specification, which has been described in detail in Chapter 2, is summarized in Table 6.2. In addition, we used the built-in device query routine to collect more information on the GPU specifications of both machines, and this information is provided in Appendix A.

According to Table 6.2, if the data for a particle consists of position, velocity and force vectors, then the memory required per particle is 36 and 72 bytes for single and double precision, respectively. Thus, for the Ruthenium system at single precision,

| | **Ruthenium** | **Hawk** |
|---|---|---|
| Model name | Maxwell GeForce GTX 960 | Tesla P100 |
| CUDA Compiler version | 7.2 | 9.2 |
| Compute Capability | 5.2 | 6.0 |
| Number of SMPs | 8 | 56 |
| Number of Cores per SMP | 128 | 64 |
| Maximum threads per block | 1024 | 1024 |
| Unified Cache L1/Texture | 48KB | 24KB |
| L2 Cache | 1MB | 4MB |
| Device Memory | ≈ 2GB | ≈ 16GB |

**Table 6.2: GPU specification for Ruthenium and Hawk.**

1365 particles would fit into the L1 cache, and 29127 particles would fit into the L2 cache. As an example, if we assume there is a maximum number of 12 particles per bin, then about 112 bins would fit into L1 cache and 2400 bins would fit into L2 cache. Assuming a cache line size of 128 bytes, only about 3 particles would fit within a cache line. In terms of the Hawk system, half the number of particles would fit into L1 cache, but four times as many would fit into L2 cache.

## 6.3 Profiling Tools

As mentioned earlier, the Nvidia Visual Profiler was used to profile the miniMD application on both machines. However, the Ruthenium machine has version 7.5 of the profiler, while Hawk has version 9.2. We noticed that the newer version provides more measurements about the compute capability of more recent GPUs. The Nvidia visual profiler supports GPUs with compute capability greater than 7.0.

## 6.4 Experimental Setup

In order to achieve our experimental objectives, we first specify whether the execution platform is the host CPU or the GPU. If the intended experiment is run on the CPU then the number of threads per block is set to a negative value; otherwise it will run on the GPU. The next configuration parameter to set is the number and range of the simulation time steps. Finally, the data ordering, and the parameters of the LJ force computation must be set as part of the execution parameters. In the case of GPU execution, three block sizes were used to investigate how performance varies for different block sizes.These various types of experiments are shown in Table 6.3.

| **Experiment Parameters** | **Description** |
|---|---|
| Machines | Ruthenium (GTX 960) and Hawk (P100) |
| Application version | CPU and GPU |
| Time-steps | 100 and 1100 |
| Data ordering | All seven orderings in Table 5.3 |
| Neighbour list configuration | Full and Half |
| Block size (only for GPU case) | 576, 640, and 1024 threads |

**Table 6.3: Various variables of our experiments .**

The numbers of experiments generated based on Table 6.3 are provided in Appendix B, where B.1 and B.2 are for the Ruthenium and Hawk experiments, respectively, which are executed for each data ordering for both 100 and 1100 time steps. Thus, for each simulation size, on the GPU platforms, the force computation is configured as full or half, and then the block size is defined according to the block sizes listed in Table 6.3, where for each block size the experiment is executed for 10 independent experiments. In both cases, for time steps 100 and 1100, the average of the last 100 time steps that call the force computation are considered (called from the integrate module using the run method). For each of the 10 independent executions the average of the average time is computed. It should be mentioned that the recorded standard deviation percentage for the 10 independent experiments was never greater than 3% of the average value.

In addition, we set the Lennard-Jones parameters the same for all our experiments. In LJ units the input parameters used were $\sigma = 1.0$, $\epsilon = 1.0$, time step =0.005, initial temperature =1.44, and density =0.8442. Neighbour lists were updated every 20 time steps; the cut-off distance was $r_0 = 2.5$, and the skin thickness was $r_s = 0.3$. In all simulations, it is assumed the particles are initially positioned with a face-centered cubic layout, which can be represented as a periodic lattice of repeating units, each containing four particles. The problem size, $N$, is the number of units in each dimension, and is related to n, the number of particles, by $n = 4N^3$. All the hybrid orderings are defined with an internal sub-block size of 2-bits, $2^{(3*2)} = 4 \times \times 4 \times 4 = 64$ bins. In these experiments, we will refer to the hybrid Hilbert row-major, hybrid Hilbert column-major, hybrid Morton row-major, and hybrid Morton column-major orderings as Hilbert RM, Hilbert CM, Morton RM and Morton CM, respectively.

In the Hawk case, where the experiments were executed using Slurm (Simple Linux Utility for Resource Management) a script was developed, and the type of ordering and type of force computation are provided as input parameters. The script then generates a batch job that is scheduled by the work load manager according to when the requested resources are available. The scripts and the batche jobs are provided in Appendix C.

The experimental results are compared with the original version of the miniMD package (without any modification) in order to ensure the results obtained are valid and accurate by comparing the results of individual time steps and the overall simulation summary.

## 6.5 CPU Experiments

### 6.5.1 Force Module Execution Time

The objective of this experiment was to show the ratio of the full and half neighbour list LJ force computations to the overall simulation time. In order to achieve this objective, the time for the force computation was compared to the total execution time for the miniMD application running on Ruthenium, using a row-major data ordering. The results show that the force computation, for both the full and half neighbour list cases, takes about 80% of the overall simulation time, as illustrated in Figure 6.1. These results justify the decision to conduct our investigation of the data locality properties of molecular dynamics simulations using the the LJ force computation.

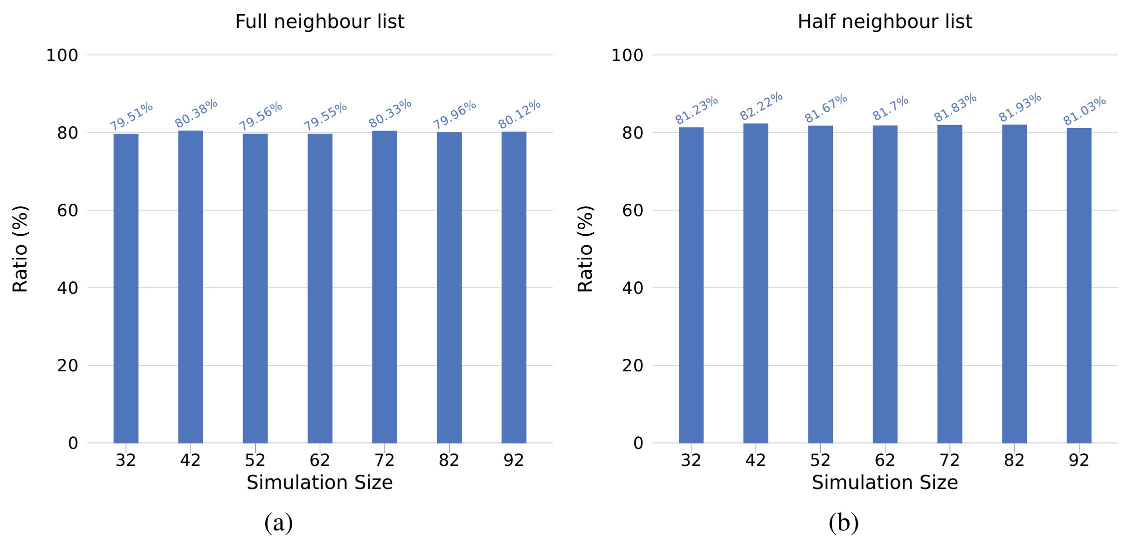


**Figure 6.1: (a) and (b) shows that the force computation for both the full and half neighbour list cases is about 80% of the overall simulation time, regardless of the simulation size.**

## 6.5.2 CPU Execution Time for Different Data Orderings

The CPU version of the application has been investigated for different data orderings using the same parameters as in the previous experiments. These experiments show which ordering has the best performance for a given simulation size, $N$. However, the smallest execution time might not guarantee the best data locality characteristics, especially for CPU platforms where a number of services and other programs share the CPU's resources. Figures 6.2 and 6.3 show that the half neighbour list algorithm is faster than the full neighbour list algorithm, particularly at larger problem sizes, $N$. In addition, the results show that the row-major ordering results in the fastest execution, especially for the half neighbour list algorithm.

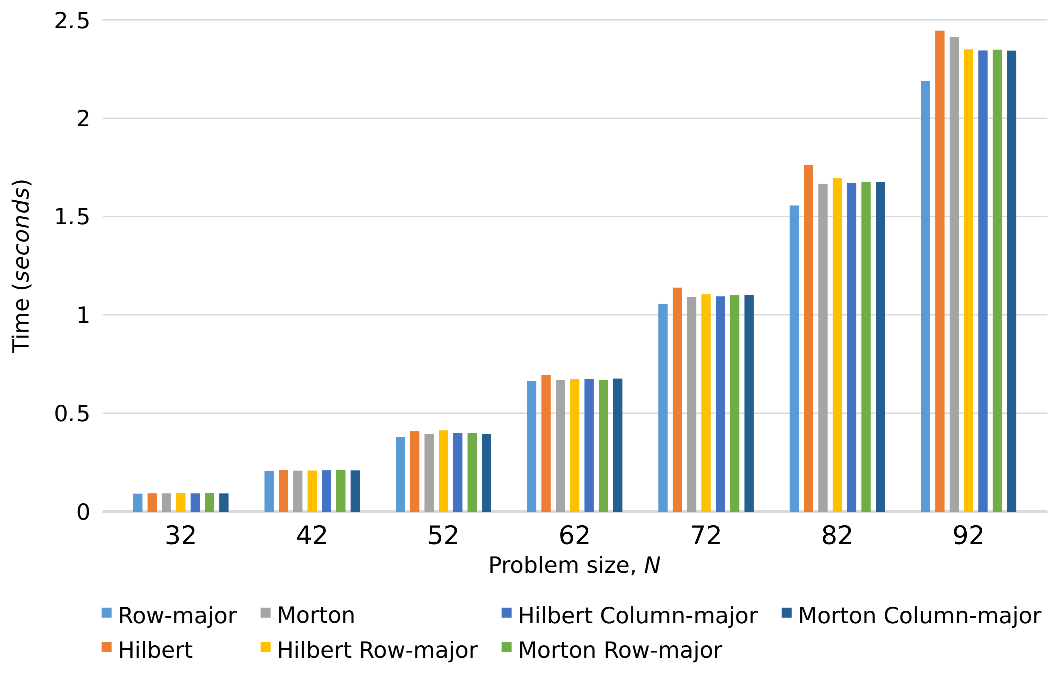


**Figure 6.2: Ruthenium,full neighbour list case: time to execute one time step of the force computation on the CPU.**

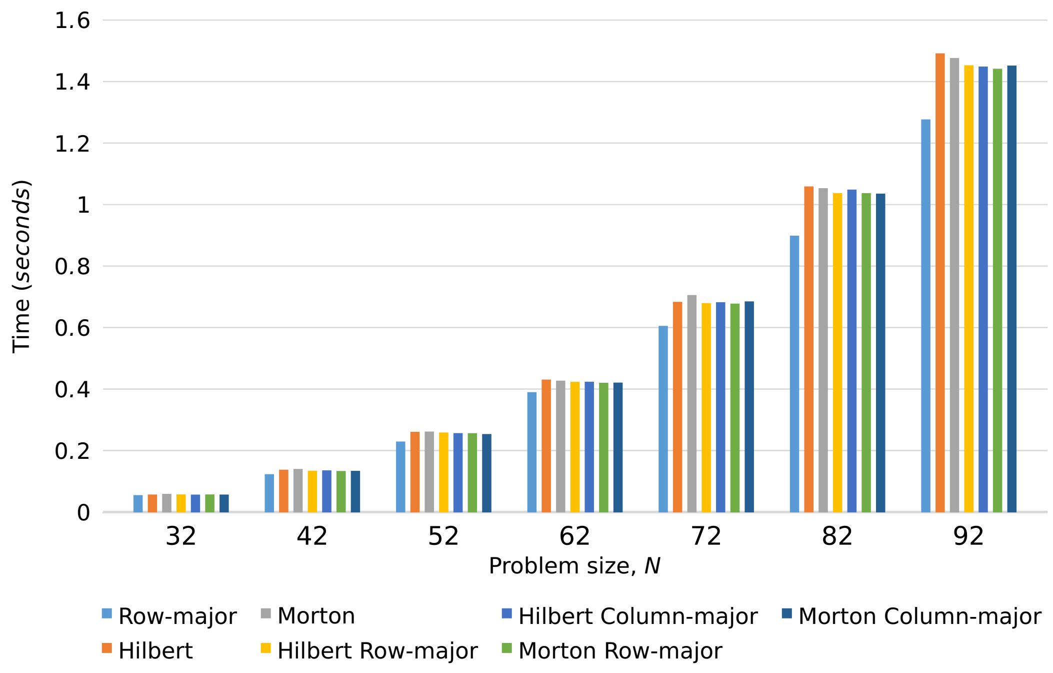


**Figure 6.3: Ruthenium, half neighbour list case: time to execute one time step of the force computation on the CPU.**

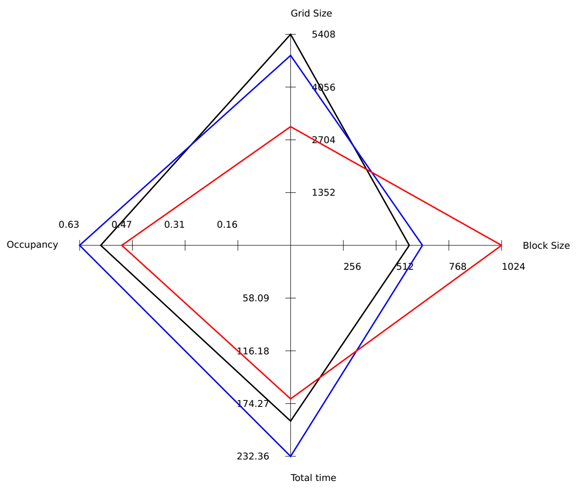
## 6.6 GPU Experiments

The data locality properties of the LJ force computation for GPU implementations are the focus of our research, therefore a number of experiments were carried out to understand the relationship between data ordering and data locality. Initially, we will compare execution on GPUs and CPUs, and then we will use the NVidia profiler to study each implementation's use of resources. This profile information will be used to account for the measured execution time in terms of data locality properties by investigating accesses and transactions for the various levels of hierarchical memory.

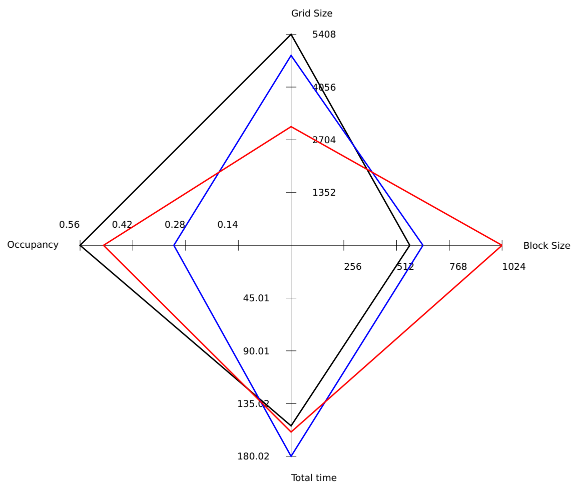
The NVidia Visual Profiler ranks the executions of each kernel by an estimation of how important it is to optimise that kernel, based on the execution time and achieved occupancy. NVidia recommends that kernel executions that rank higher (i.e., those that are expected to benefit most from optimisation), are used to conduct GPU analysis, rather than taking an average of all the executions for that kernel, as this approach is more likely to result in improved performance. We have followed this recommendation in the results and analysis presented in this chapter.

### 6.6.1 Theoretical Occupancy

The theoretical occupancy is the maximum number of warps that can execute on a streaming multiprocessor of a GPU divided by the device limit, and it is affected by factors such as the number of threads per block, the number of registers in use, and the capabilities of the GPU. Figures 6.4 and 6.5 reflect the relation between thread configuration, theoretical occupancy, and total execution time for the full neighbour and half neighbour list cases on the Ruthenium and Hawk systems, respectively. In this experiment, we set the problem size, $N = 92$, although similar results were obtained for other problem sizes. For the Ruthenium system, the half neighbour list algorithm, with a block size of 576, has an optimal theoretical occupancy of 0.563, which is less than for the full neighbour list case of 0.625, where the block size is 640. This disparity arises because the half neighbour list algorithm requires more registers compared to the full neighbour list case. However, for the full neighbour list case, higher theoretical occupancy results in slower execution times. This could be because the actual occupancy is lower for larger grids. On the other hand, for the half neighbour list case high theoretical occupancy correlates with faster execution. For the Hawk system, the general trends are similar to Ruthenium, although for the half neighbour list algorithm, the execution times are almost identical, as shown in 6.5b.



(a)

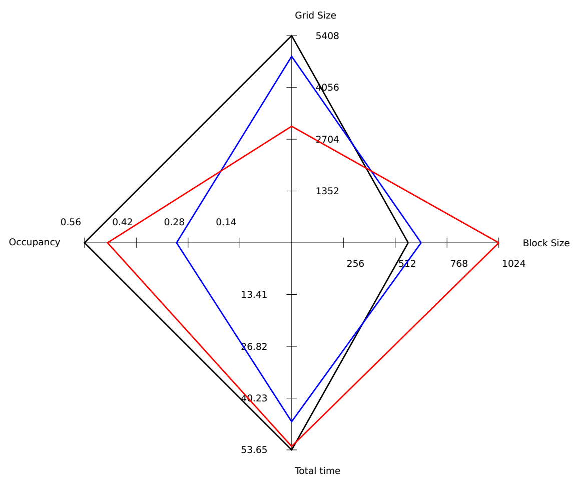


(b)

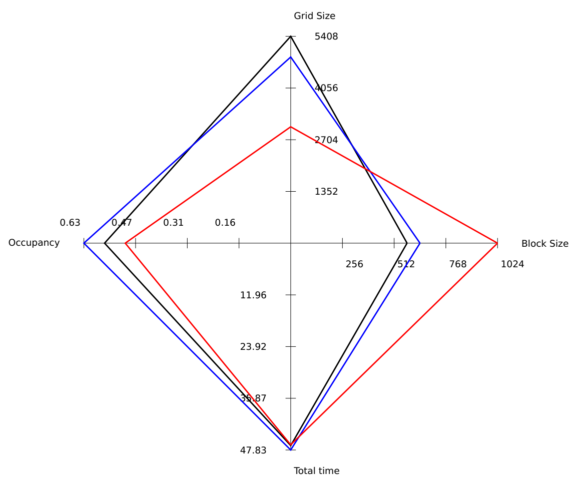
**Figure 6.4: Ruthenium: dependency of total execution time and theoretical occupancy for different thread block configurations for the full (a) and half (b) neighbour list cases. The problem size is N=92.**

### 6.6.2 GPU Force Computation Ratio

In order to show the ratio of the force computation to the total simulation time for the CPU and GPU versions, the Ruthenium machine was used to run our experiments for a row-major ordering. The number of time steps was 100, and in the GPU case the block size was set to 1024. For the CPU implementation, the ratio of the force computation to the total simulation time is consistently about 80% for both neighbour list cases, as shown in Figure 6.6. For the GPU case, in which the force time includes the time for transferring particle data between the host and the GPU in each time step, the ratio



(a)



(b)

**Figure 6.5: Hawk: dependency of total execution time and theoretical occupancy for different thread block configurations for the full (a) and half (b) neighbour list cases. The problem size is N=92.**

decreases as the problem size increases, as shown in Figure 6.7. This is because the neighbour lists are managed on the host and the processing time for this increases more rapidly than the force computation on the GPU side. The execution time for the half neighbour list case is about 60% of that of the full neighbour list case for the CPU implementation. On the other hand, for the GPU case, the difference is much less, because the lower number of neighbour particles processed is offset by the need to accumulate forces atomically.

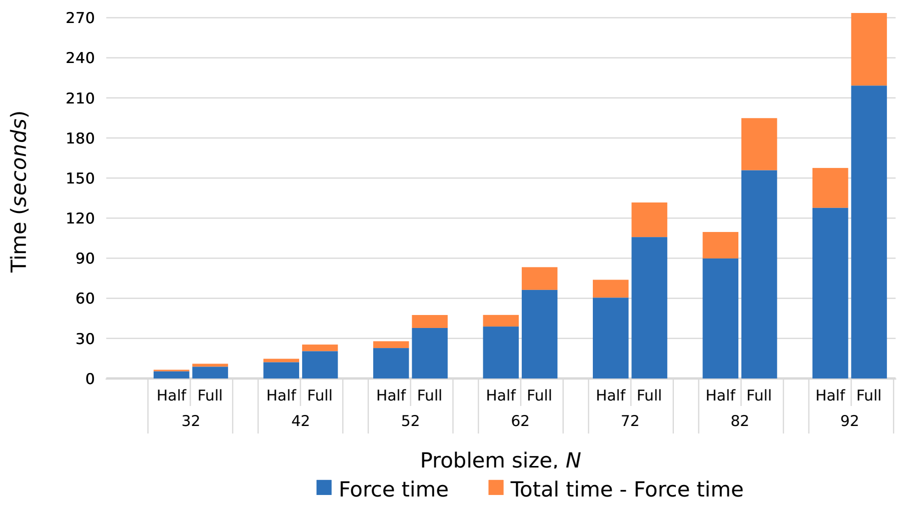


**Figure 6.6: Total simulation time and force computation time for the CPU implementation on Ruthenium, accumulated over the first 100 time steps, for different problem sizes, $N$. Times are shown for the half and full neighbour list algorithms. Note the total height of each column is the total time for the simulation.**

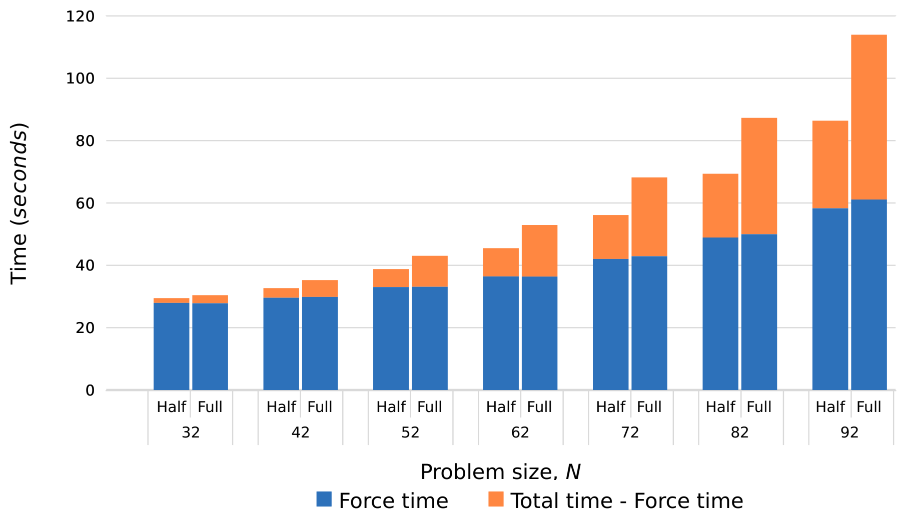


**Figure 6.7: Total simulation time and force computation time for the GPU implementation on Ruthenium with 1024 threads per block, accumulated over the first 100 time steps, for different problem sizes, $N$. Times are shown for the half and full neighbour list algorithms. Note the total height of each column is the total time for the simulation.**

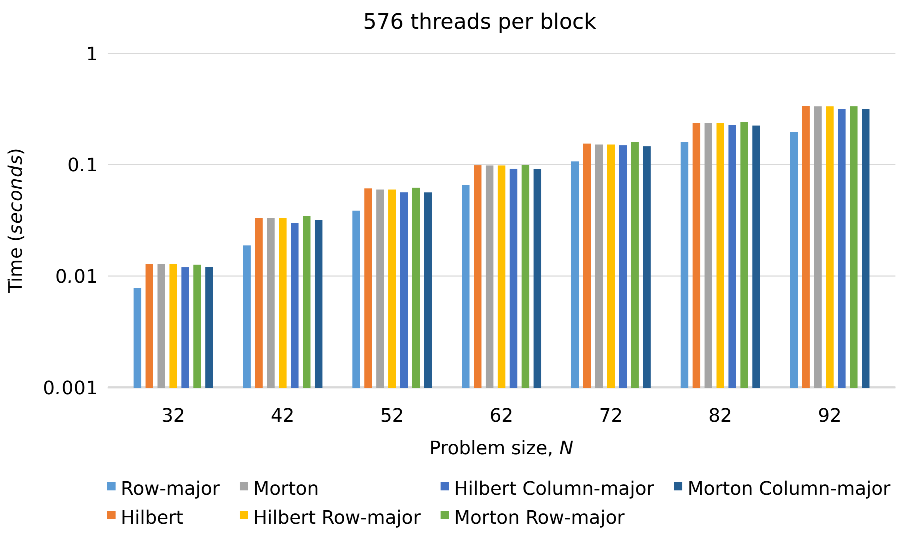
The ratio of execution time of the CPU and GPU cases for the force computation is shown in Table 6.4. In general, the ratio tends to increase slowly with problem size, $N$. For the half neighbour list algorithm, the ratio is smaller compared to the full neighbour list case, because of the atomic additions performed on the GPU side. Another reason might be the increased number of stalled threads caused by thread divergence due to additional conditional statements in the half neighbour list algorithm. This will be investigated further in the profiling analysis section.

| *N* | Row-major | | Hilbert | | Morton | | Hilbert RM | | Hilbert CM | | Morton RM | | Morton CM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Full | Half | Full | Half | Full | Half | Full | Half | Full | Half | Full | Half | Full | Half |
| 32 | 11.68 | 6.89 | 8.07 | 5.98 | 8.66 | 6.38 | 8.63 | 6.18 | 8.45 | 6.05 | 9.1 | 6.31 | 8.85 | 6.15 |
| 42 | 12.27 | 7.38 | 7.32 | 6.58 | 7.1 | 6.77 | 7.52 | 6.55 | 7.28 | 6.28 | 7.56 | 6.58 | 7.02 | 6.34 |
| 52 | 12.29 | 7.75 | 7.65 | 6.84 | 7.49 | 7.02 | 8.12 | 7.02 | 7.39 | 6.67 | 8.02 | 7.09 | 7.41 | 6.72 |
| 62 | 12.14 | 7.84 | 7.94 | 7.21 | 7.69 | 7.09 | 8.32 | 7.34 | 7.62 | 7.03 | 8.22 | 7.19 | 7.83 | 6.93 |
| 72 | 12.66 | 7.87 | 8.03 | 7.07 | 7.85 | 7.57 | 8.02 | 7.16 | 7.48 | 6.87 | 8.18 | 7.42 | 7.72 | 7.2 |
| 82 | 12.33 | 8.01 | 7.96 | 7.15 | 7.57 | 7.23 | 8.04 | 7.22 | 7.46 | 7 | 7.99 | 7.35 | 7.52 | 7.01 |
| 92 | 12.55 | 8.12 | 7.95 | 7.23 | 7.85 | 7.26 | 8.02 | 7.27 | 7.62 | 7.07 | 8.05 | 7.33 | 7.67 | 7.18 |

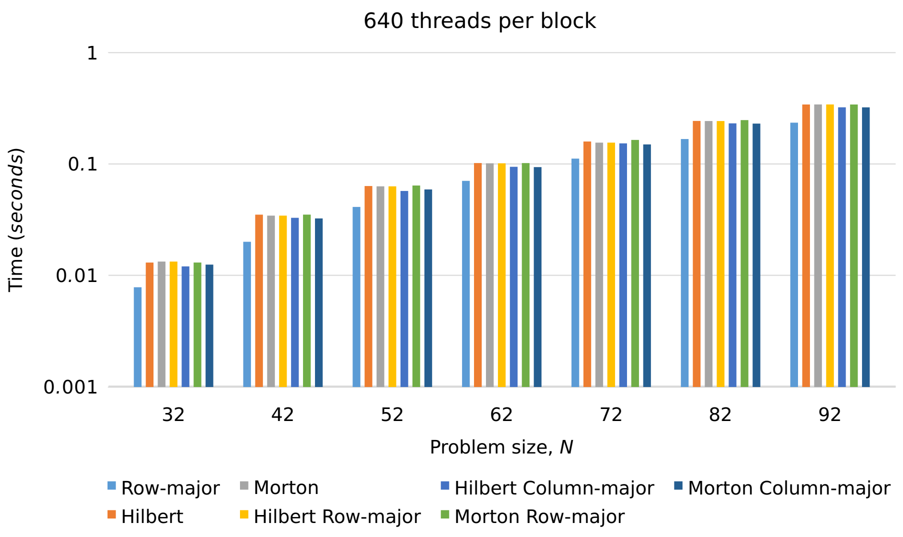
**Table 6.4: Ruthenium: ratio of CPU to GPU execution time for the force computation. In the GPU computation there are 1024 threads per block.**

## 6.7 Locality Analysis

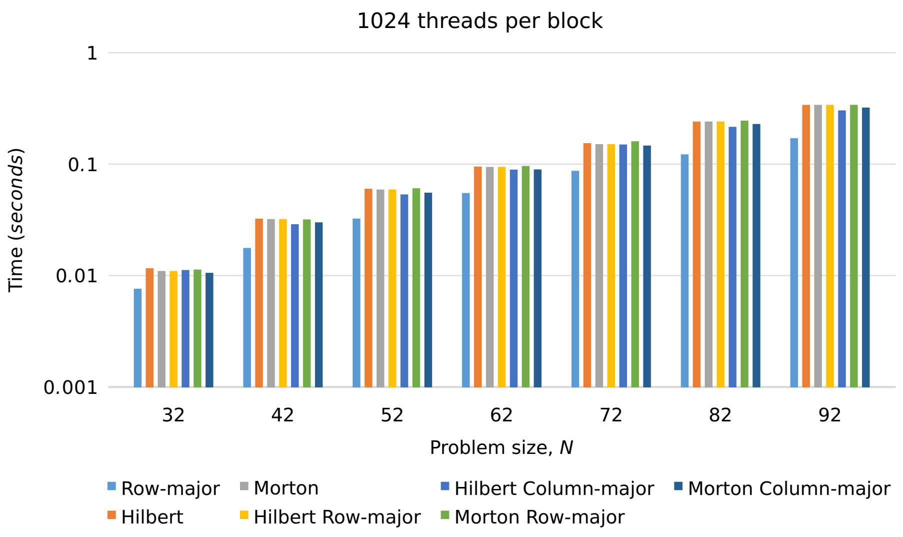
In order to study and compare just the data locality properties of the orderings considered in this research, and their impact on the performance of the force computation, all subsequent timing experiments were carried out without including the time for particle transfers between the CPU and GPU; in other words, the times reported for only for the kernel function used for the force computation. The experiments were performed for two scenarios: the first was for time steps 1 to 100, with averages being computed for this range of time steps; the second was run for 1100 time steps with avergaes being computed over time steps 1001 to 1100. By gathering timing and profiling data for these two scenarios on the Ruthenium and Hawk machines, the experimental results show that both scenarios have similar trends in terms of the ordering of execution time. Figures 6.8 and 6.9 show the time on Ruthenium for the force computation kernel per time step for differing thread block sizes for the full and half neighbour list cases, respectively. The half list algorithm achieves faster execution than the full neighbour list algorithm, especially at larger problem sizes. In the case of the full neighbour list, for all the problem sizes and the block sizes considered, the row-major ordering results in the fastest execution, as shown in Figure 6.8. In addition, for all problem sizes and orderings, a thread block size of 1024 is fastest and 640 is slowest, except for $N = 82$ and $N = 92$, where a block size of 576 results in approximately 2% faster execution. The advantage of a smaller block size is to increase the number of registers available per thread, which tends to enhance performance. However, it also decreases the number of warps per block, which may lead to inefficient GPU utilization. Hybrid orderings are aligned with other orderings in terms of the optimal performance of the adopted block size. They are still slower than row-major ordering and have slightly faster execution times compared with pure Hilbert and Morton orderings, especially for the Hilbert RM and Morton RM cases which are about 2.8% to 10.8%, and 2.9% to 7.6%, faster than the pure Hilbert and Morton cases, respectively.



(a)

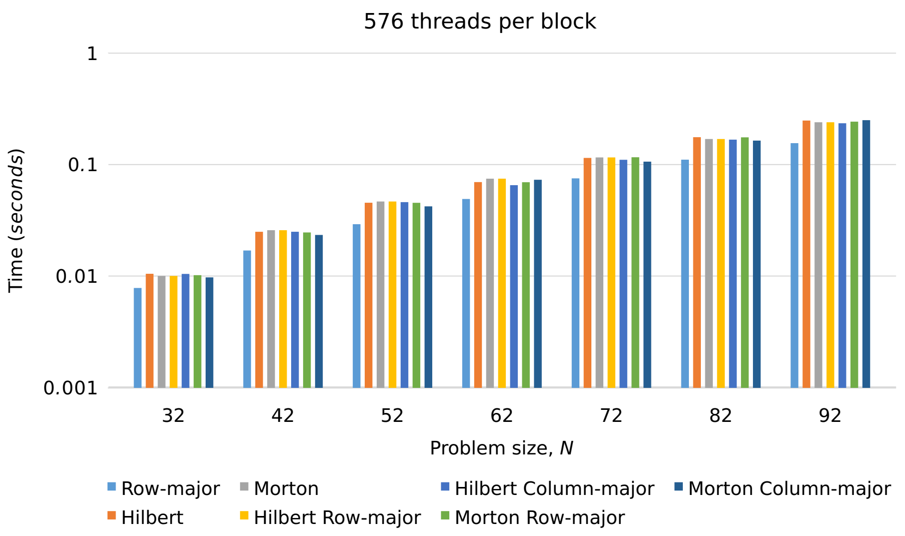


(b)

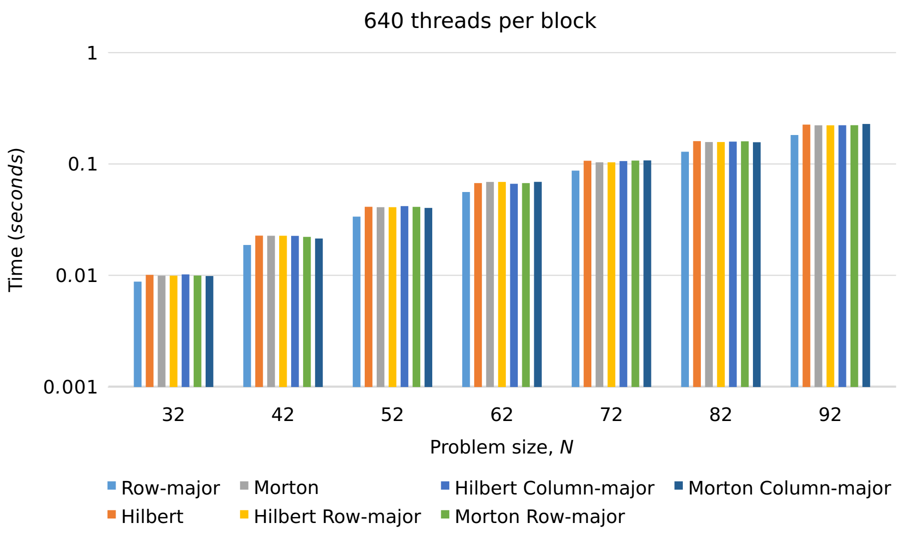


(c)

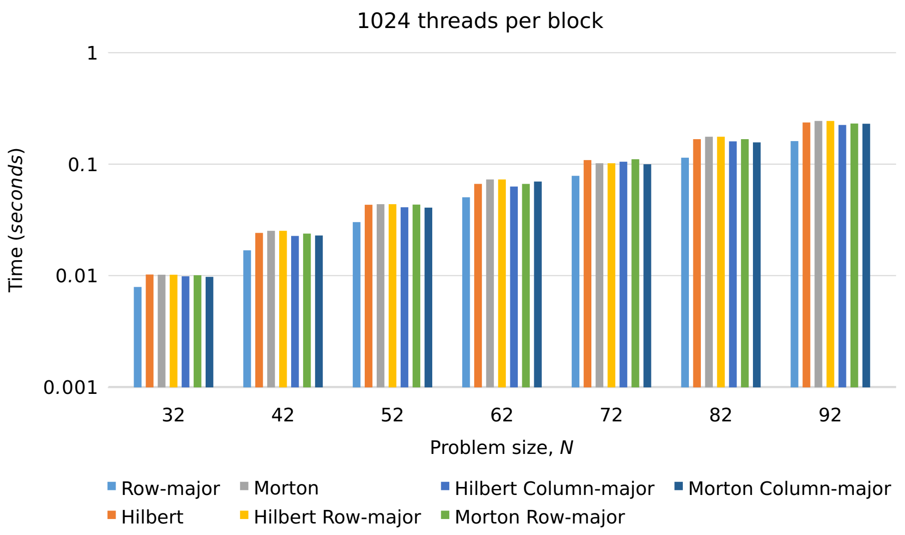
**Figure 6.8: Ruthenium, full neighbour list case. Time for execution of force computation kernel on the GPU for one time step.**



(a)



(b)

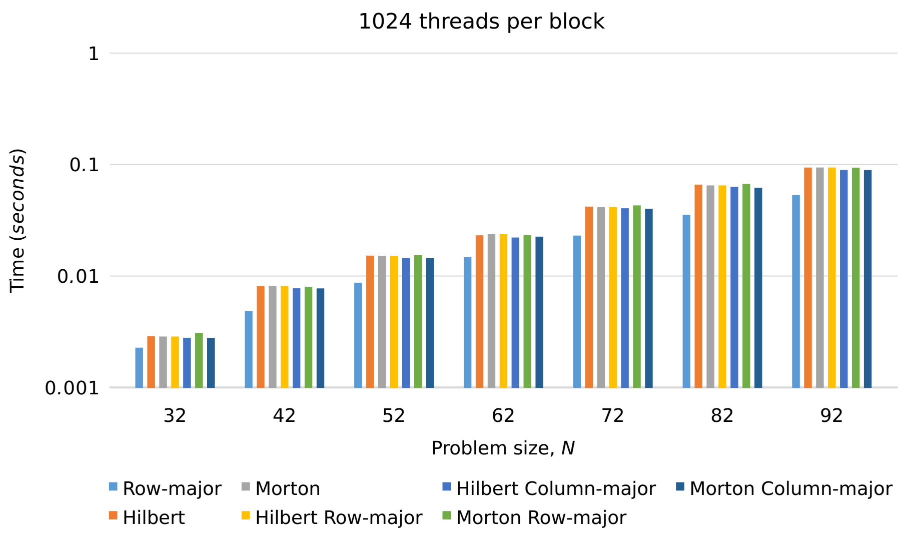


(c)

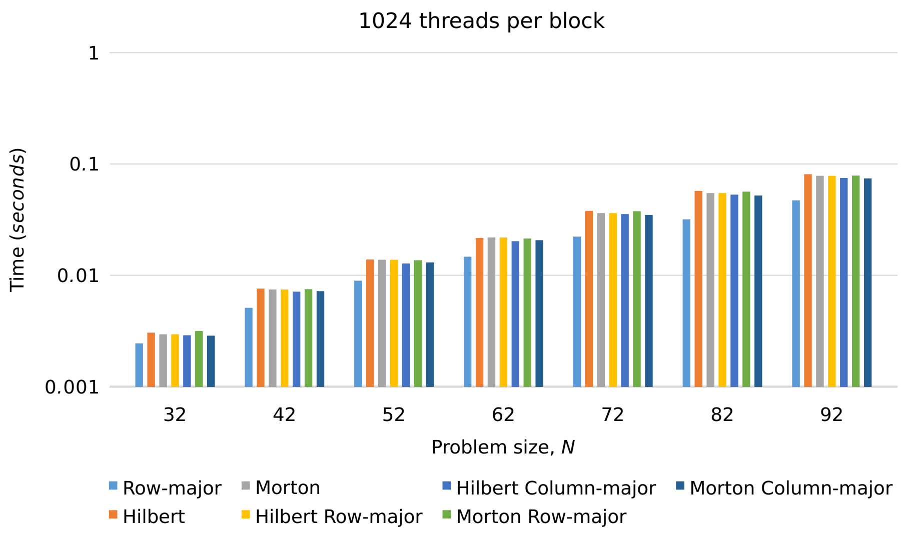
**Figure 6.9: Ruthenium, half neighbour list algorithm. Time for execution of force computation kernel on the GPU for one time step .**

The results for the half neighbour list algorithm, shown in Figure 6.9, also illustrate that a row major ordering results in the fastest execution, although not by as large an amount as in the full neighbour list case. In the half neighbour list case with row-major ordering, it was found that a thread block size of 576 was fastest, with 640 still being the slowest. On the other hand, for the Hilbert and Morton orderings, a block size of 640 was the fastest. For hybrid orderings, where the position of the neighbouring particles can favour either the inner sub-block ordering (row-major or column order) or the outer block ordering (Hilbert or Morton), it was found that a block size of 640 was the fastest in most cases, especially for hybrid Morton orderings. However, the results do not show any of the hybrid orderings being consistently faster than the others.

The corresponding execution times are faster on the Hawk system, however, similar trends are observed. We provide the corresponding figures for execution on Hawk in Appendix D, and only the results for 1024 threads per block are provided in Figure 6.10, as required for the remaining analysis of this work.



(a)



(b)

**Figure 6.10: Hawk: time for execution of force computation kernel on the GPU for one time step in the full (upper) and half (lower) neighbour list cases.**

### 6.7.1 Execution Efficiency Analysis

To analyse the locality in more detail, the *nvprof* profiler was used to collect data on the execution of the force computation kernel to gain insight into how efficiently it is being used and how the orderings affect data locality. In these experiments, profiling was conducted for a problem size of $N = 92$ (3114752 particles), a block size of 1024 threads per block, and time steps 1001-1100 for both the full and half neighbour list cases for the Ruthenium (GTX 960) and Hawk (P100) machines.

The number of eligible warps is shown in Table 6.5. An eligible warp is an active warp that is able to issue its next instruction, in contrast to a stalled warp that is not able to make progress.

Table 6.5 shows that the number of cycles to execute the force computation kernel is approximately 9-12% larger in the full neighbour list case, compared with the half neighbour list case. This is because the number of computations is higher in the full neighbour list case. For the Ruthenium system, the percentage of active warps that are eligible is about 7.4±0.1% for all orderings in the full neighbour list case, but is larger in the half neighbour list case: 10.25±0.05%. For the Hawk system the corresponding values are about 2.2% and 3.2%, which accounts for the larger number of cycles needed to execute the force computation kernel on Hawk, compared with Ruthenium. For the Hawk system, the low percentage of active warps that are eligible may be due to the memory-dependent warp stalling seen in Figures 6.13 and 6.14.

Figures 6.11 and 6.12 show further information regarding the cause of the stalled warps in the force computation kernel for the Ruthenium system. These figures show that memory dependency and execution dependency are the main reasons for the stalled warps. Memory dependent stalls occur when required resources are unavailable, fully utilized or there are too many requests for a load/store operation; in other words, when a warp must wait for a previous memory operation to release a required resource. In the full neighbour list case, memory dependencies are the dominant reason for warp stalls, accounting for 84% to 90% of all stalls, compared with a more uniform value of 90% for the half neighbour list case. On the other hand, execution dependency stalls occur when an input required by the instruction is not yet available. In the full neighbour list case, execution dependencies account for 14% of stalls for a row-major ordering, but for only 11.5% of stalls for Hilbert and Morton orderings. In the half neighbour list

| **GTX 960** | | Number of Cycles | Eligible Warps (%) |
|---|---|---|---|
| Row-major | Full | $11.85 \times 10^9$ | 7.5 |
| | Half | $10.83 \times 10^9$ | 10.3 |
| Hilbert | Full | $12.13 \times 10^9$ | 7.3 |
| | Half | $10.84 \times 10^9$ | 10.3 |
| Morton | Full | $12.08 \times 10^9$ | 7.3 |
| | Half | $10.76 \times 10^9$ | 10.2 |
| Hilbert RM | Full | $11.96 \times 10^9$ | 7.4 |
| | Half | $10.78 \times 10^9$ | 10.5 |
| Hilbert CM | Full | $12.06 \times 10^9$ | 7.4 |
| | Half | $10.85 \times 10^9$ | 10.2 |
| Morton RM | Full | $11.96 \times 10^9$ | 7.4 |
| | Half | $10.70 \times 10^9$ | 10.5 |
| Morton CM | Full | $12.02 \times 10^9$ | 7.3 |
| | Half | $10.81 \times 10^9$ | 10.2 |

| **P100** | | Number of Cycles | Eligible Warps (%) |
|---|---|---|---|
| Row-major | Full | $42.87 \times 10^9$ | 2.2 |
| | Half | $38.27 \times 10^9$ | 3.2 |
| Hilbert | Full | $43.06 \times 10^9$ | 2.2 |
| | Half | $38.25 \times 10^9$ | 3.2 |
| Morton | Full | $42.57 \times 10^9$ | 2.1 |
| | Half | $37.67 \times 10^9$ | 3.2 |
| Hilbert RM | Full | $42.88 \times 10^9$ | 2.1 |
| | Half | $37.32 \times 10^9$ | 3.3 |
| Hilbert CM | Full | $43.32 \times 10^9$ | 2.1 |
| | Half | $37.77 \times 10^9$ | 3.2 |
| Morton RM | Full | $42.89 \times 10^9$ | 2.2 |
| | Half | $37.59 \times 10^9$ | 3.3 |
| Morton CM | Full | $42.76 \times 10^9$ | 2.2 |
| | Half | $37.20 \times 10^9$ | 3.2 |

**Table 6.5: Number of active cycles and percentage of active warps that are eligible per active cycle, for $N = 92$ and 1024 threads per block. Ruthenium (upper) and Hawk (lower).**

case, execution dependencies consistently account for 8% to 9% of stalls. It should be noted that there are no synchronization stalls as the kernel code contains no explicit synchronization.

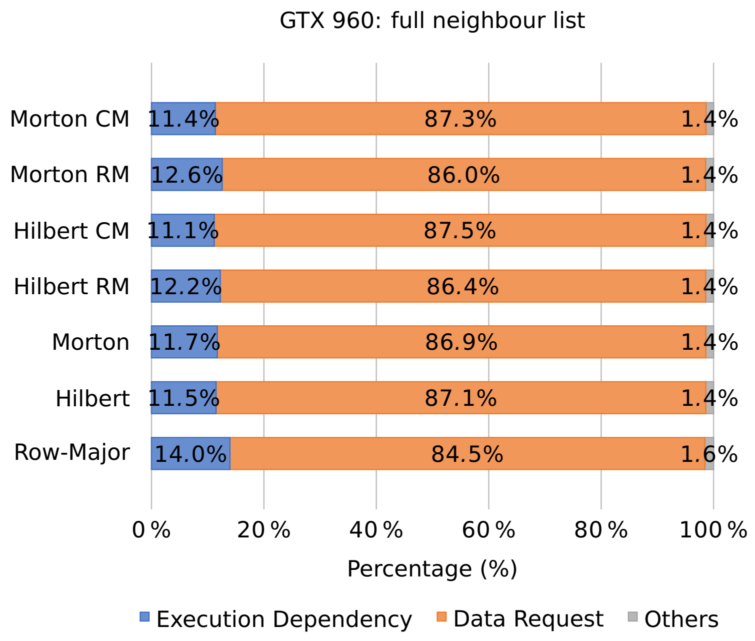


**Figure 6.11: Ruthenium, full neighbour list case. Output from *nvprof* giving the cause of warp stalling in the force computation kernel using the different orderings.**

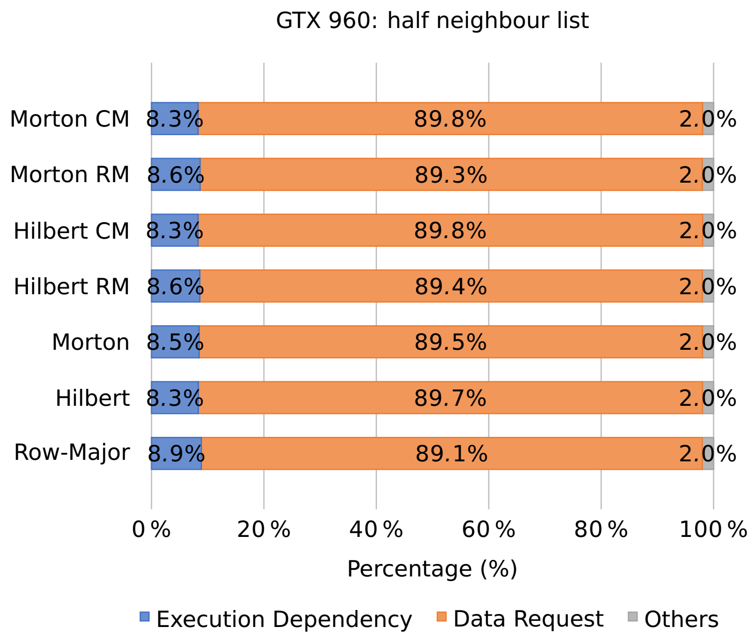


**Figure 6.12: Ruthenium, half neighbour list case. Output from *nvprof* giving the cause of warp stalling in the force computation kernel using the different orderings.**

For the Hawk system, warp stalling is caused almost entirely by memory dependency, as shown in Figures 6.13 and 6.14. In this system, the absence of any significant execution dependency is due to to the large number of SMPs and cores in the P100, compared with the GTX960, which reduces the likelihood of a thread having to wait for another thread to compute a required input. Hybrid ordering is just ±1% compared to other (standard) orderings for the full neighbour list case, but they are consistent for the half neighbour list case.

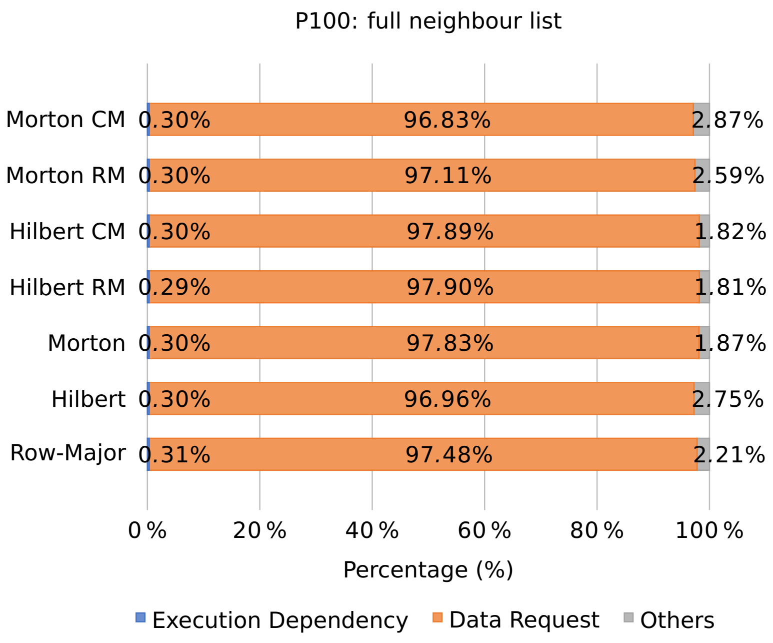


**Figure 6.13: Hawk: full neighbour list case. Output from *nvprof* giving the cause of warp stalling in the force computation kernel using the different orderings.**

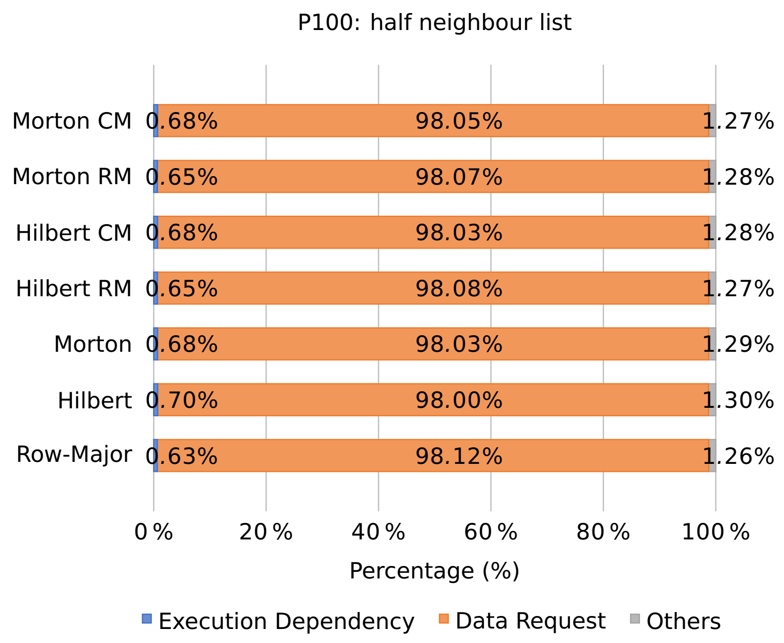
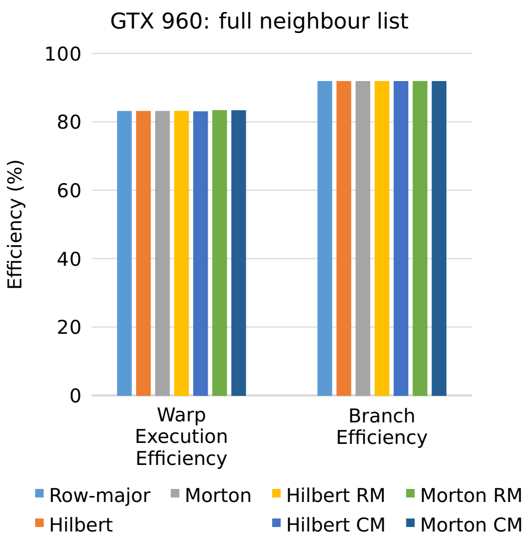
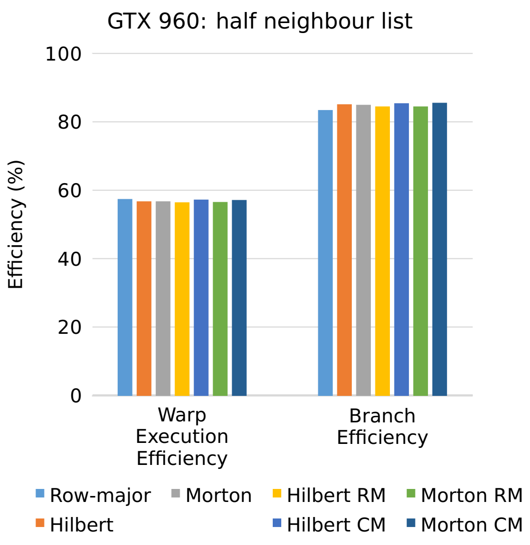


**Figure 6.14: Hawk, half neighbour list case. Output from *nvprof* giving the cause of warp stalling in the force computation kernel using the different orderings.**

Warp execution efficiency is the ratio of the average number of active threads per warp to the maximum number of threads per warp supported by the multiprocessor. In a kernel code different execution paths (arising from conditional statements) and non-coalesced memory accesses affects warp execution efficiency, which is known as intra-warp divergence. Branch efficiency is the ratio of executed uniform flow control decisions over all executed conditionals, and thus gives a measure of divergence. The warp execution and branch efficiency for Ruthenium are shown in Figure 6.15, which shows that the warp execution efficiency is about 57% and 83% for the half and full neighbour list cases, respectively. The branch efficiency is about 7-8% less in the half neighbour list case, so intra-warp divergence accounts for at least some of the lower warp execution efficiency in this case. The Hawk system shows similar results to the Ruthenium system, as shown in Figure 6.16, despite the very small decreases observed for row-major and hybrid Morton orderings, which could be due to the low spatial correlation of particles as a result of the reduced number of common neighbours. As seen in Chapter 5, the half neighbour list algorithm has additional control paths (*if* statements), compared to the full neighbour list case, which leads to a decrease in warp efficiency inside the streaming multiprocessor. In other words, although these threads belong to the same bin, and so should have a high likelihood of processing common neighbour particles in the memory, yet some threads in the same warp process some particles while others do not as those neighbour particles do not meet the neighbourhood condition of the half neighbour list algorithm.

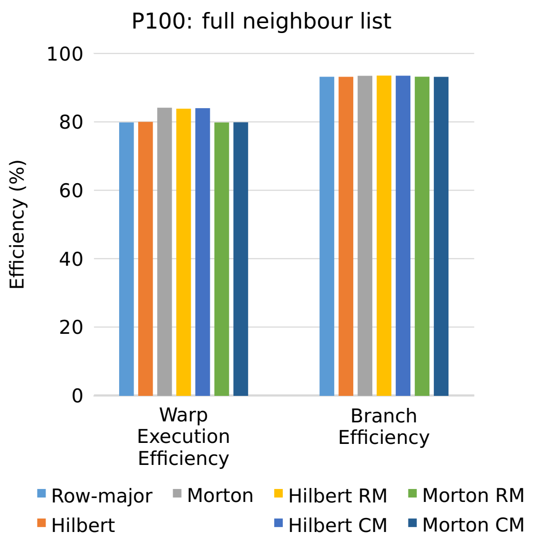


(a)

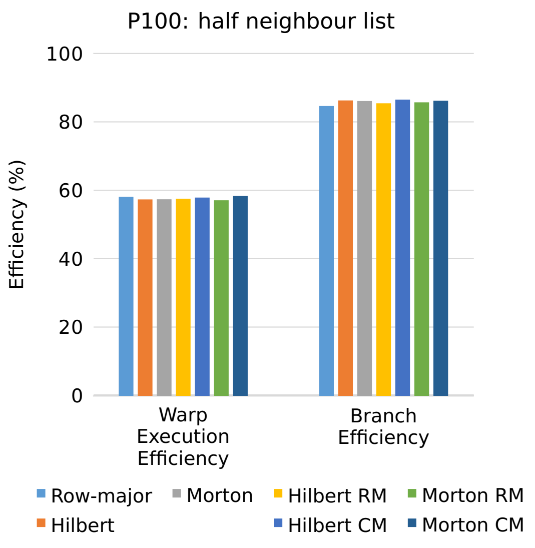


(b)

**Figure 6.15: Ruthenium: warp execution and branch efficiency for the full (a) and half (b) neighbour list algorithms.**



(a)



(b)

**Figure 6.16: Hawk: warp execution and branch efficiency for the full (a) and half (b) neighbour list algorithms.**

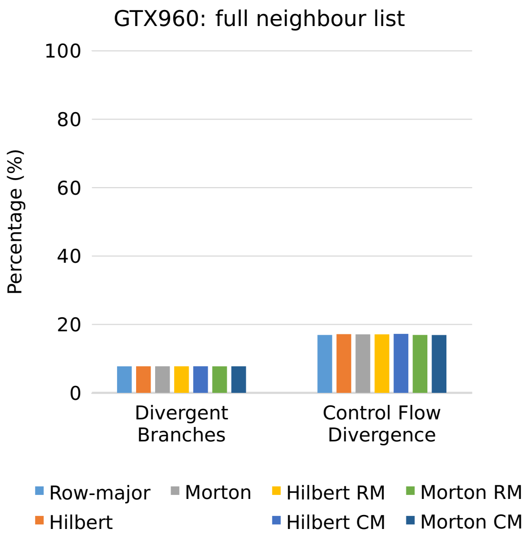
In addition, warp divergence can be studied using the percentage of divergent branches and control flow divergence percentages (see [44, p. 92]). The divergent branches percentage is the percentage of branches that are causing divergence in a warp amongst all the branches present in the kernel. This can be calculated as:

$$\frac{(100 \times \text{divergent branches})}{(\text{divergent branches} + \text{non-divergent branches})}$$

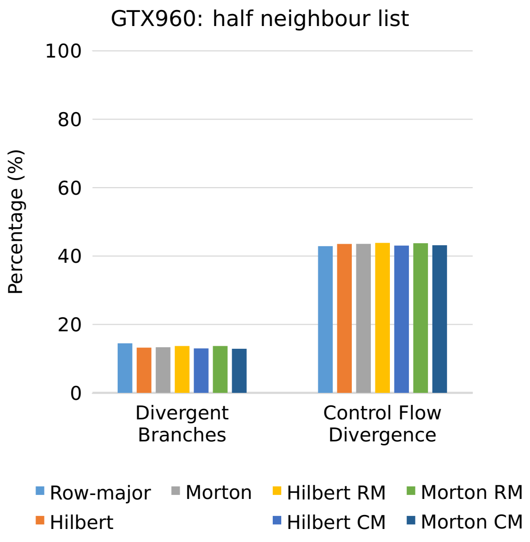
Control flow divergence is the percentage of thread instructions that were not executed by all threads in the warp. This is calculated as:

$$\frac{100 \times ((32 \times \text{instructions executed}) - \text{thread instructions executed})}{(32 \times \text{instructions executed})}$$

Despite the fact that some divergence is unavoidable, as in our case, these percentages should be as low possible to achieve higher warp execution efficiency. Both the Ruthenium and Hawk systems have similar results, except for the Morton and hybrid Hilbert orderings, which have a lower percentage. This explains why the warp execution efficiency for these orderings has a higher percentage for the full neighbour list case, as shown in Figure 6.16.

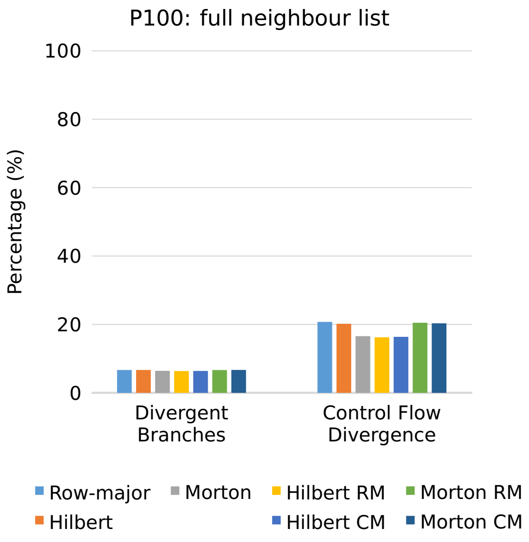


(a)

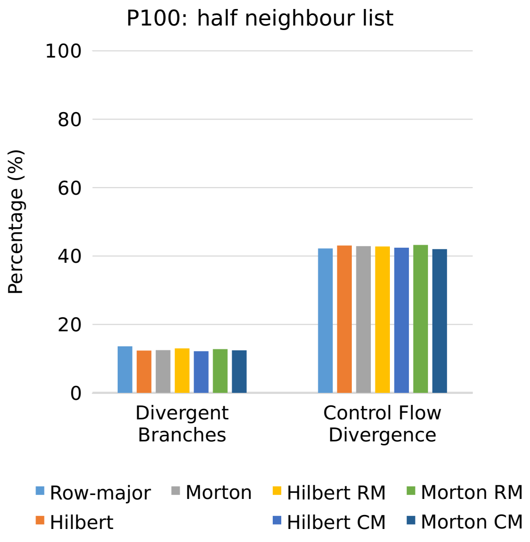


(b)

**Figure 6.17: Ruthenium: the percentages of divergent branches and control flow divergence warp execution and branch efficiency for the full (a) and half (b) neighbour list algorithms. Note, low percentage is more efficient.**



(a)



(b)

**Figure 6.18: Hawk: the percentages of divergent branches and control flow divergence warp execution and branch efficiency for the full (a) and half (b) neighbour list algorithms.Note, low percentage is more efficient.**

### 6.7.2 GPU Memory Analysis

A memory transaction is the movement of data between two areas of memory. A smaller number of transactions when accessing data results in higher efficiency. When loading a data item for all the threads in a warp from global memory to L2 cache, the number of memory transactions in the ideal case is

$$\left\lceil \frac{\text{(Number of threads in a warp)} \times \text{(Size of data item)}}{\text{Size of cache line}} \right\rceil$$

Thus, if the warp size is 32 threads, the data item size is 8 bytes, and the cache line size is 32 bytes, then ideally 8 memory transactions are needed. Figure 6.19 shows, for the Ruthenium system, the actual number of memory transactions per access for the following accesses in the force computation:

AC: Access an 8-byte coordinate value for a thread's particle.

NI: Access the 4-byte index of a particle from the neighbour list.

NC: Access an 8-byte coordinate value for a particle in the neighbour list.

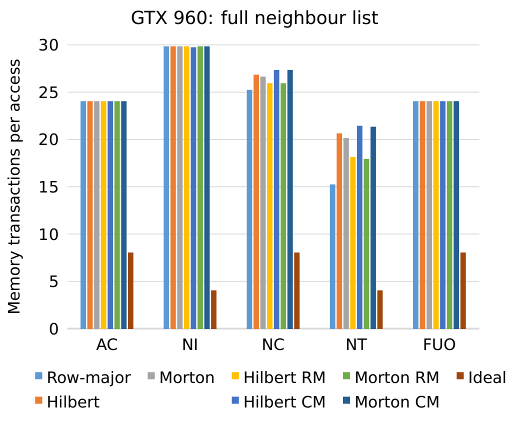
NT: Access the 4-byte type of a particle.

FUI: Access an 8-byte force component for a particle in the neighbour list (half neighbour list case only).

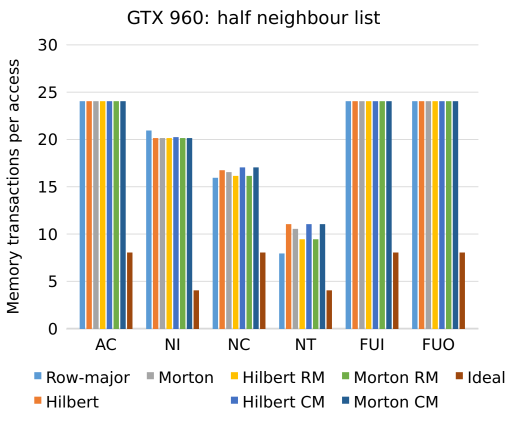
FUO: Access an 8-byte force component for a thread's particle.

Figure 6.19 shows that when accessing 8-byte doubles, the number of memory transactions per access is 2 to 3 times the ideal number, and when accessing integers, the corresponding ratio is larger. As described in Chapter 5, particle data are stored in arrays in coordinate order. For example, the x, y, and z coordinates of particle $i$ are stored at indexes $3i$, $3i + 1$, and $3i + 2$ of the position array. Consequently, the x value of consecutive particles is separated by 24 bytes so that accessing a single x value for all the threads in a warp requires 24 memory transactions (and similarly for the y and z values). This is the case for the AC, FUI, and FUO accesses in Figure 6.19, but for the NC access the value is slightly larger than 24 for the full neighbour list case, and less for the half neighbour list case. Large values occur when not all the accesses for a warp are in the same block of 32 bytes, while smaller values occur when the neighbour lists for successive particles have particles in common. Likewise, the time for the NI, NC, and NT accesses is less for the half neighbour list case, because fewer particles are processed in this case. This means particle data stays in L2 cache for a longer time before being evicted.

For the Hawk system, it was found that the number of memory transactions per access is very similar to those for Ruthenium, as shown in Figure 6.20.

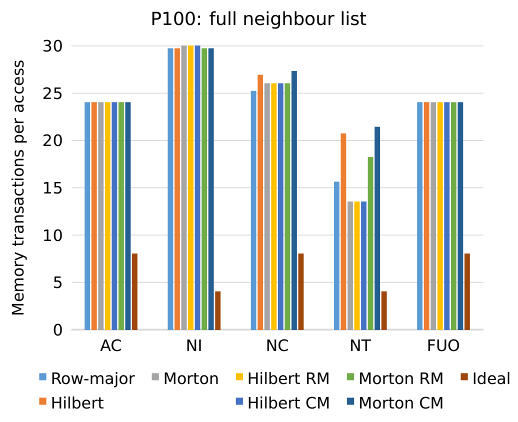


(a)

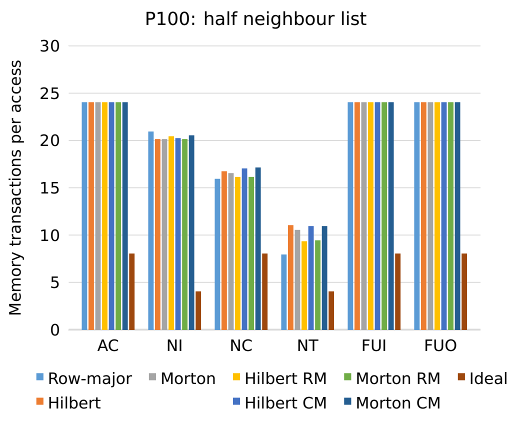


(b)

**Figure 6.19: Ruthenium: number of memory transactions per access for the full (a) and half (b) neighbour list algorithms.**



(a)



(b)

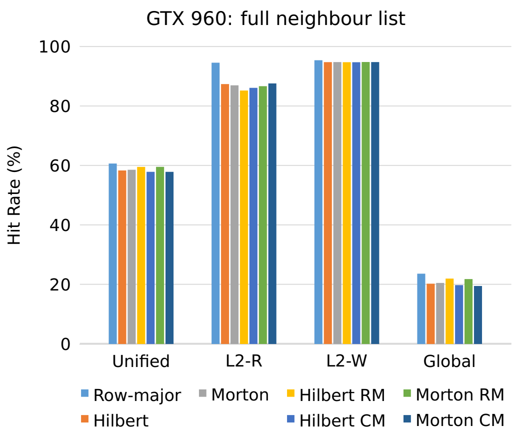
**Figure 6.20: Hawk: number of memory transactions per access for the full (a) and half (b) neighbour list algorithms.**

As indicated in Table 6.2, the L1 and texture caches of both GPUs are combined into a single unit referred to as the “unified cache”, the size of which is 48 KB and 24 KB for the Ruthenium and Hawk systems, respectively. The cache line size of both systems is 128 bytes. Since a warp consists of 32 threads, then, for the Ruthenium system, there are 1536 bytes of L1 cache for each thread. The position and force components for a particle correspond to six 8-byte values (without padding) for a total of 192 bytes, so ideally 6 particles per thread will fit into unified cache. For the Hawk system, this value is halved. However, loading neighbour list particles displaces data already in the unified cache, and hence this reduces the hit rate. Note that the cache line size for the L2 cache is 32 bytes. Figure 6.21 shows the following cache hit rates for the force computation for the full and half neighbour list cases:

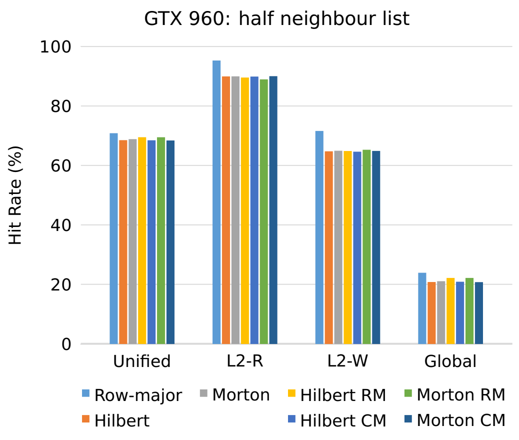
- Unified: the cache hit rate for the unified cache, i.e., the percentage of accesses that are found in the unified cache. Note that for the GPUs used here, by default read-only data are cached only in the unified cache.
- L2-R: the percentage of read requests (for data that is not read-only) that are satisfied by the L2 cache.
- L2-W: the percentage of write requests that are satisfied by the L2 cache.

- Global: the percentage of accesses to read-only data not satisfied by the unified or L2 caches that result in a direct read from global memory to unified memory without going through L2 cache.

For the Ruthenium system, the hit rate for row-major ordering is slightly larger than for any other orderings, as illustrated in Figure 6.21. The half neighbour list cases have fewer misses to unified cache, but more misses when writing to the L2 cache. In general, hybrid Hilbert CM and Morton CM orderings have a lower cache hit rate compared to all the other orderings for both the full and half neighbour list algorithms, except for the L2-R hit rate. Figure 6.22, shows that the cache hit rates for the Hawk system, and demonstrates how the hit rates for the unified cache and for global memory access are higher than for the Ruthenium system. However, the hit rate for reads and writes to/from the L2 cache is slightly lower for Hawk compared with Ruthenium, except for Morton ordering in the full neighbour list case where the hit rate for reads from L2 cache is higher for Hawk.

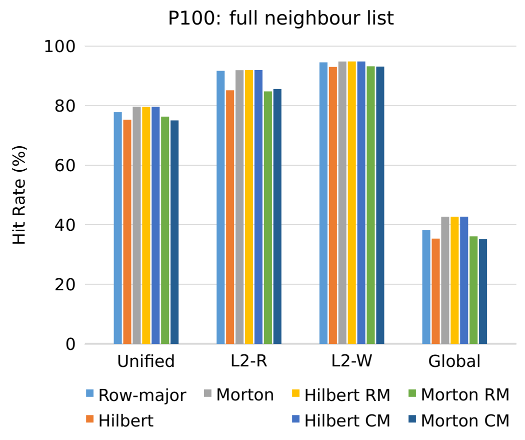


(a)

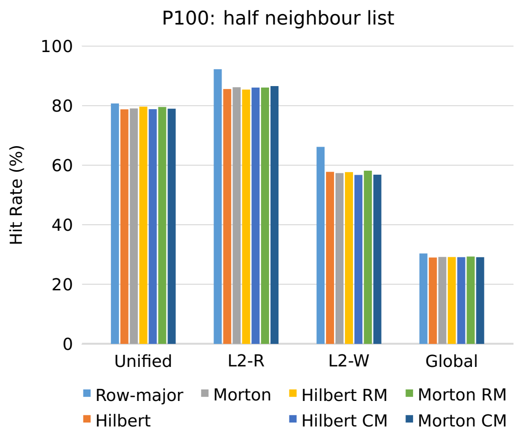


(b)

**Figure 6.21: Ruthenium: cache hit rate for the full (a) and half (b) neighbour list algorithms.**



(a)



(b)

**Figure 6.22: Hawk: cache hit rate for the full (a) and half (b) neighbour list algorithms.**

## 6.8 Summary

This chapter has described the experiments, results and analysis that have been conducted in our research. These experiments covered the execution time and the NVidia Visual Profiler results, and these data were used to compare the characteristics of the different data orderings. The profiler provides a useful tool to evaluate the data locality of the different orderings, and the results shows a significant correlation with the cache model that was introduced in Chapter 4. The next chapter will present the research findings of these results, and make recommendations regarding possible future research work.

Chapter 7

# Conclusions and Future Work

This chapter presents the findings derived from the work presented in this thesis in order to demonstrate the extent to which our research contributions, that were outlined in Chapter 1, were achieved. The overarching findings are presented, followed by a more detailed discussion of the conclusions. This chapter will also present the limitations of our research, and recommend how our research can be extended in the future.

## 7.1 Conclusion

This research has presented seven types of data ordering in order to evaluate the data locality properties of molecular dynamics simulations that use a Lennard-Jones potential in the inter-particle force computation. Both full and half neighbour list algorithms have been implemented for GPUs using the CUDA programming framework.

In Chapter 2, specifically Section 2.3, we presented our approach for generating the various data orderings, especially using space-filling curves and their related hybrid data orderings. In Chapter 4, we used these orderings in a simple cache model that investigates the relationship between the stencil shape and data locality properties for different orderings. The simple model, which evaluates memory access patterns, shows that for a row-major ordering with a given stencil the data locality properties are independent of spatial location (assuming periodic boundaries), while this is not the case for Hilbert and Morton orderings. Therefore, Hilbert and Morton orderings have a higher degree of memory scatter in their memory access pattern compared to a row-major ordering. The cache miss rate model shows that for a row-major ordering the miss rate remains constant for a range of cache sizes, and then gradually decreases as the cache size increases. On the other hand, the Hilbert and Morton orderings do not illustrate this trend as they are not ordered by row. The orderings are impacted by the cache block size and the overall size of the cache, resulting in the Hilbert and Morton orderings having a lower miss rate than a row-major ordering. The comparison between the approximate spherical stencil and the block stencil shows that the former has a lower miss rate compared to the latter. Moreover, data reuse or temporal data locality have less dependence on the ordering used as the number of bins shared in the processing of adjacent bins is $2g(2g + 1)^2$, where $g$ is is the stencil size parameter.

In Chapter 6, the execution time and the *nvprof* profiling tool were used to analyse the data locality properties of the different data orderings. An analysis of the CPU execution time showed that the force computation is the most expensive part of each time step in terms of the execution time. In addition, the row-major ordering has a faster execution time compared with the other orderings especially for the half neighbour list algorithm.

The key findings for the GPU implementation can be summarised as follows:

- The ratio of CPU to GPU execution time shows that the GPU implementation can significantly reduce the execution time for a large problem size.
- The analysis of theoretical occupancy shows that high theoretical occupancy does not guarantee the optimal performance for an application.
- Threads and data alignment and coalesced data accesses, have a significant impact on application performance.
- Branch divergence significantly impacts the execution time, especially for high data dependency applications.

The analysis of data locality that has been carried out shows that the locality properties of data orderings are affected by cache size, where row-major orderings are best for a large cache size, while Hilbert and Morton orderings are better for a small cache size, as demonstrated by the cache model. The number of particles in each bin, and the amount of padding between particles within the array, affects the properties of data orderings as the cache size might not be sufficient to store all the particles of a bin particles in the high level cache, and the padding decreases the amount of particle data that can be loaded in each cache line. As a result of each thread computing the interaction force between one specific particle and its neighbouring particles, which are also neighbours with other threads (especially those located in the same bin), the number of cycles to execute the force computation is very high due to memory dependency and execution dependency. This is particularly apparent in the Ruthenium results. The half neighbour list algorithm executes faster than the full neighbour list algorithm due to the lower number of particles involved in the computation and the lower number of neighbouring particles. However, the extra conditional statements in the half neighbour list case causes the warp execution efficiency and branch efficiency to decrease the overall execution efficiency.

Data dependency has an inverse relationship with temporal data locality, especially if particles are processed with atomic accesses. Therefore, particles being scattered in the memory might yield better performance since this reduces the cost of serializing instruction access. In addition, the cost of branch divergence has a slight impact on the execution time as the execution efficiency is decreased. Branch divergence reduces both spatial and temporal data locality as it may either not utilize the existing data or might require uploading different data that cannot be utilized by consecutive threads.

On the other hand, our memory analysis shows that the issue of the padding and alignment in accessing various arrays during the force computation increases the number of memory transactions per access. The displacement of the data during the force computation reduces data locality, especially temporal data locality.

Cache memory has been analysed with a simple cache model, and the unified cache hit rate for the Hawk system (which has a smaller cache size) is best for Hilbert and Morton orderings, in the case of the full neighbour list algorithm. This is because these orderings have better spatial locality than a row-major ordering as stencil bins are more clustered in memory around the stencil center, as demonstrated in Chapter 4. However, Hilbert and Morton orderings have a long ‘memory tail’ in the sense that 10-20% of stencil bins are further away in the memory than any of the other of stencil bins in the row-major case. In the case of the Ruthenium system (where the unified cache is larger), the row-major ordering has a slightly better cache hit rate. These results suggest that the performance benefits of the different orderings depend on the size of the different levels in the memory hierarchy, and on the cache line size.

Although the cache model shows that stencil bins are more clustered about the stencil centre in the Hilbert and Morton cases than for the row-major case, the small size of the unified cache and the large number of interactions per particle results in a lot of ‘churn’ at this level in the memory hierarchy. This reduces the impact that the different data orderings have on the data locality properties of the application.

## 7.2 Research Questions Answered

In this research, we have investigated the relation between different computation stencils and the adopted data orderings by using block (cubic) and approximately spherical stencils, as illustrated in Chapter 4. Overall this research has examined the data locality properties of different data orderings for MDS applications implemented on GPU platforms, as shown in Chapters 4, 5 and 6. The space filling curve orderings have been adopted without modifying the data structure or specific computational workflow in order to present a consistent study among the various data orderings.

This research also shows how to conduct data locality and performance analysis studies for other applications that have high data dependency. A cache model have been implemented which can be utilised as is for other applications, or with some modifications to account for different data dependency relationships, such as adopting a different type of stencil. Secondly, a performance analysis has been carried out to study the ratio of CPU to GPU execution time, the impact of theoretical occupancy and data locality by using the NVidia profiler. Finally, an analysis has been conducted using the NVidia profiler in terms of the execution efficiency, which is useful in evaluating different implementations. This type of analysis covers the evaluation of the number of eligible warps and identifying the reasons for stalled warps. The eligible warps are also studied from the perspective of the number of cycles required to complete warp execution. In addition, warp execution efficiency and branch efficiency are studied to analyse the differences between the full and half neighbour list force implementations. The full neighbour list case requires more computation, while for the half neighbour case the implemented conditional branches are costly. Therefore, additional analysis has been conducted to evaluate the percentage of divergent branches and control flow divergence, as presented in Section 6.7.1. GPU memory analysis has been performed to study the memory transactions, the type of memory accesses and the cache hit rates for both full and half force computations. All these analyses can be adopted as guidelines to use the NVidia profiler for similar types of application.

Even though this research is aligned with a number of other studies, such as [15, 53, 86, 135], in terms of using GPUs and space-filling curves to improve the MDS performance and execution time, however, to the best of our knowledge, this research is the first to address the data locality properties of MDS on GPUs. Moreover, most of the previous implementations introduced various data structures, such as interaction matrices, and incurred additional overhead in terms of the execution time. The high correlation, as explained earlier, between the findings of our cache model and the NVidia profiler is evidence of the accuracy and relevance of the obtained results and analysis.

## 7.3 Limitations

This research was limited to the study of only one application (miniMD) and considered one model of the potential energy (Lennard-Jones). In addition, the experiments and analysis were conducted on only two NVidia GPUs. Despite these limitations, the Lennard-Jones potential represents a good example of an irregular application, that has a high degree of data dependency and cannot be clustered or divided independently for the computation, to study MDS data locality properties on different GPUs.

## 7.4 Future Work

GPUs are an important platform for parallel applications, and are less expensive compared to other HPC environments. Therefore, future work could extend the timing experiments and analyse other NVidia GPUs and a broader range of molecular dynamic simulations. As an example, the embedded-atom method (EAM) [38, 39], which is one of the potentials supported by miniMD, will be implemented in our future work to investigate its data locality and performance characteristics compared to a Lennard-Jones potential. EAM is widely implemented for purely metallic systems with no directional bonding, and is used to approximate the structure, energetics, and properties of metals. This study can also be extended to include other molecular dynamics potentials.

It is possible to extend the GPU implementation to bypass the unified cache so that only the L2 cache is used. As the size of the L2 cache is large, we assume this will favour row-major orderings. However, investigating the impact on the overall execution time might yield different results compared to the full utilization of the memory hierarchy. In addition, and in order to reduce the data dependency, the work might be extended by dividing the force computation into various kernel functions such that the distance computation, neighbourhood test, and the interparticle force computation are computed by separate kernel streams. Research could then be focused on investigating the execution and the memory utilization in this implementation, and the impact of the reduction operation required in the potential energy computation could be examined.

In this study two types of space filling curve were used, namely the Hilbert and Morton curves. The research can be extended by adopting other space filling curves. In addition, the hybrid orderings used a row or column major ordering for the lower bits of the particle index (the inner ordering) and the upper bits were based on the Hilbert or Morton ordering (the outer ordering). This can be flipped around so that the inner ordering is based on the Hilbert or Morton ordering and the outer ordering is based on the row or column major ordering. In addition, hybrid orderings can be extended to use two space filling curves, such that the inner ordering is Hilbert and the outer ordering is Morton, or vice versa. Another interesting future investigation would be to base the ordering of the the bins based on filling a spherical shell, as proposed in [119]. This can also be extended to the Morton case. Therefore, that data locality properties of these cases can be compared with the results of this research.

The issues of data dependency and execution dependency might be examined by modifying the data structure of our implementation to use a two-dimensional array to store the particle position and force. In this case, one dimension represents the particle under consideration and the other dimension represents the neighbouring particles. In breaking down the computation in this way, the neighbourhood test decision for an interaction can be carried out by a separate kernel to record the results in a two-dimensional array. With this approach, a one-dimensional grid of two-dimensional thread blocks can be used to compute the interaction force between particle $i$ and its neighbour $j$ by one thread represented by the block dimension $x$ and $y$, respectively. Of course, this demands handling all the dimensions ( $x$, $y$ and $z$) of the position and force arrays. Another alternative approach is to replace the two-dimensional array by a one-dimensional array, using the Hilbert or Morton label as the index of the particle $i$ and its neighbour $j$. Thus, the thread at position $(x, y)$ in a two-dimensional block is mapped to access the space filling curve label in the one-dimensional array. Comparison of the two approaches in terms of execution time, memory utilization, and data locality for each representation can then be evaluated.

Finally, the work can be extended to support multiple nodes and multiple GPUs, with some modification needed to check the number of GPUs and to modify the communications module to support multiple nodes by using MPI with CUDA-aware and GPU-Direct support.

## 7.5 Summary

In this multidisciplinary research, where the data locality properties of GPU implementations for molecular dynamics simulations have been studied, we found the optimal data orderings are highly dependent on the cache size and the cache line size, as well as on the nature of the application in terms of the degree of data dependency. This chapter has emphasised the findings of our research as well as recommendations for future work that could potentially extend our research.

# Appendix A

# Device Query Output

This appendix shows the device query output for the Ruthenium and Hawk systems. The objective to show the detailed hardware specifications.

## A.1 Ruthenium (GeForce GTX 960)

### Device 0: “GeForce GTX 960”

| | |
|---|---|
| CUDA Driver Version / Runtime Version | 7.5 / 7.5 |
| CUDA Capability Major/Minor version number: | 5.2 |
| Total amount of global memory: | 2045 MBytes (2144141312 bytes) |
| ( 8) Multiprocessors, (128) CUDA Cores/MP: | 1024 CUDA Cores |
| GPU Max Clock rate: | 1291 MHz (1.29 GHz) |
| Memory Clock rate: | 3600 Mhz |
| Memory Bus Width: | 128-bit |
| L2 Cache Size: | 1048576 bytes |
| Maximum Texture Dimension Size (x,y,z) | 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096) |
| Maximum Layered 1D Texture Size, (num) layers | 1D=(16384), 2048 layers |
| Maximum Layered 2D Texture Size, (num) layers | 2D=(16384, 16384), 2048 layers |
| Total amount of constant memory: | 65536 bytes |
| Total amount of shared memory per block: | 49152 bytes |
| Total number of registers available per block: | 65536 |
| Warp size: | 32 |
| Maximum number of threads per multiprocessor: | 2048 |
| Maximum number of threads per block: | 1024 |
| Max dimension size of a thread block (x,y,z): | (1024, 1024, 64) |
| Max dimension size of a grid size (x,y,z): | (2147483647, 65535, 65535) |
| Maximum memory pitch: | 2147483647 bytes |
| Texture alignment: | 512 bytes |
| Concurrent copy and kernel execution: | Yes with 2 copy engine(s) |
| Run time limit on kernels: | Yes |
| Integrated GPU sharing Host Memory: | No |
| Support host page-locked memory mapping: | Yes |
| Alignment requirement for Surfaces: | Yes |
| Device has ECC support: | Disabled |
| Device supports Unified Addressing (UVA): | Yes |
| Device PCI Domain ID / Bus ID / location ID: | 0 / 1 / 0 |

## A.2 Hawk (Tesla P100)

Device 0 : "Tesla P100-PCIE-16GB"

| | |
|---|---|
| CUDA Driver Version / Runtime Version | 9.2 / 9.0 |
| CUDA Capability Major/Minor version number: | 6.0 |
| Total amount of global memory: | 16281 MBytes (17071734784 bytes) |
| (56) Multiprocessors, ( 64) CUDA Cores/MP: | 3584 CUDA Cores |
| GPU Max Clock rate: | 1329 MHz (1.33 GHz) |
| Memory Clock rate: | 715 Mhz |
| Memory Bus Width: | 4096-bit |
| L2 Cache Size: | 4194304 bytes |
| Maximum Texture Dimension Size (x,y,z) | 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384) |
| Maximum Layered 1D Texture Size, (num) layers | 1D=(32768), 2048 layers |
| Maximum Layered 2D Texture Size, (num) layers | 2D=(32768, 32768), 2048 layers |
| Total amount of constant memory: | 65536 bytes |
| Total amount of shared memory per block: | 49152 bytes |
| Total number of registers available per block: | 65536 |
| Warp size: | 32 |
| Maximum number of threads per multiprocessor: | 2048 |
| Maximum number of threads per block: | 1024 |
| Max dimension size of a thread block (x,y,z): | (1024, 1024, 64) |
| Max dimension size of a grid size (x,y,z): | (2147483647, 65535, 65535) |
| Maximum memory pitch: | 2147483647 bytes |
| Texture alignment: | 512 bytes |
| Concurrent copy and kernel execution: | Yes with 2 copy engine(s) |
| Run time limit on kernels: | No |
| Integrated GPU sharing Host Memory: | No |
| Support host page-locked memory mapping: | Yes |
| Alignment requirement for Surfaces: | Yes |
| Device has ECC support: | Enabled |
| Device supports Unified Addressing (UVA): | Yes |
| Device PCI Domain ID / Bus ID / location ID: | 0 / 59 / 0 |

## Device 1 : “Tesla P100-PCIE-16GB”

| | |
|---|---|
| CUDA Driver Version / Runtime Version | 9.2 / 9.0 |
| CUDA Capability Major/Minor version number: | 6.0 |
| Total amount of global memory: | 16281 MBytes (17071734784 bytes) |
| (56) Multiprocessors, ( 64) CUDA Cores/MP: | 3584 CUDA Cores |
| GPU Max Clock rate: | 1329 MHz (1.33 GHz) |
| Memory Clock rate: | 715 Mhz |
| Memory Bus Width: | 4096-bit |
| L2 Cache Size: | 4194304 bytes |
| Maximum Texture Dimension Size (x,y,z) | 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384) |
| Maximum Layered 1D Texture Size, (num) layers | 1D=(32768), 2048 layers |
| Maximum Layered 2D Texture Size, (num) layers | 2D=(32768, 32768), 2048 layers |
| Total amount of constant memory: | 65536 bytes |
| Total amount of shared memory per block: | 49152 bytes |
| Total number of registers available per block: | 65536 |
| Warp size: | 32 |
| Maximum number of threads per multiprocessor: | 2048 |
| Maximum number of threads per block: | 1024 |
| Max dimension size of a thread block (x,y,z): | (1024, 1024, 64) |
| Max dimension size of a grid size (x,y,z): | (2147483647, 65535, 65535) |
| Maximum memory pitch: | 2147483647 bytes |
| Texture alignment: | 512 bytes |
| Concurrent copy and kernel execution: | Yes with 2 copy engine(s) |
| Run time limit on kernels: | No |
| Integrated GPU sharing Host Memory: | No |
| Support host page-locked memory mapping: | Yes |
| Alignment requirement for Surfaces: | Yes |
| Device has ECC support: | Enabled |
| Device supports Unified Addressing (UVA): | Yes |
| Device PCI Domain ID / Bus ID / location ID: | 0 / 216 / 0 |

# Appendix B

# Generated Experiments

## B.1 Ruthenium Experiments

For each data ordering, and with 100 and 1100 time steps the experiments are configured with a specific simulation size and block size and then it is executed on Ruthenium machine.

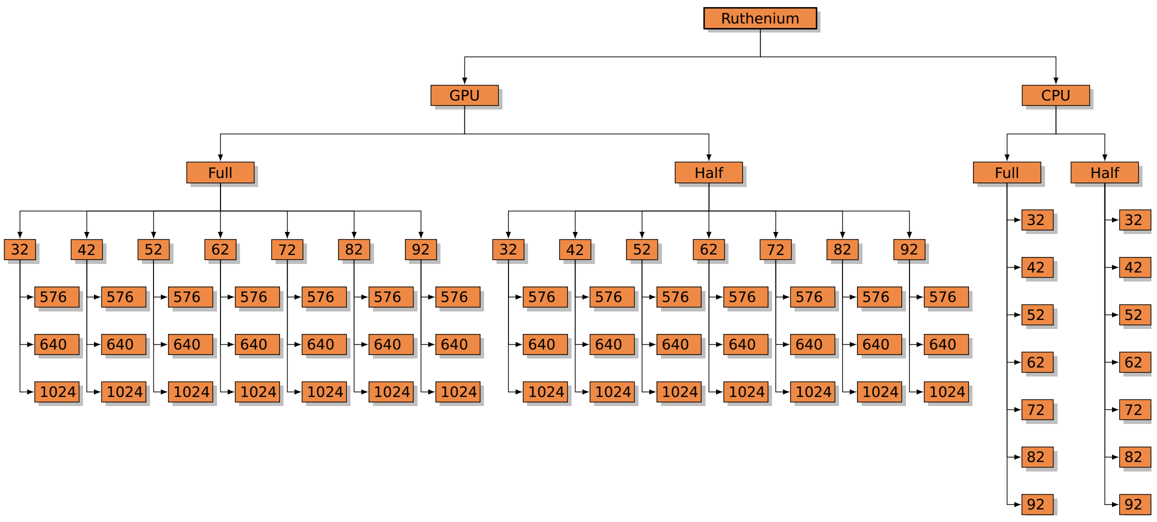


**Figure B.1: The generated experiments for each data ordering, and with 100 and 1100 time steps.**

## B.2 Hawk Experiments

Identical to Ruthenium, each data ordering, and with 100 and 1100 time steps the experiments are configured with a specific simulation size and block size and then it is executed on the Hawk machine.

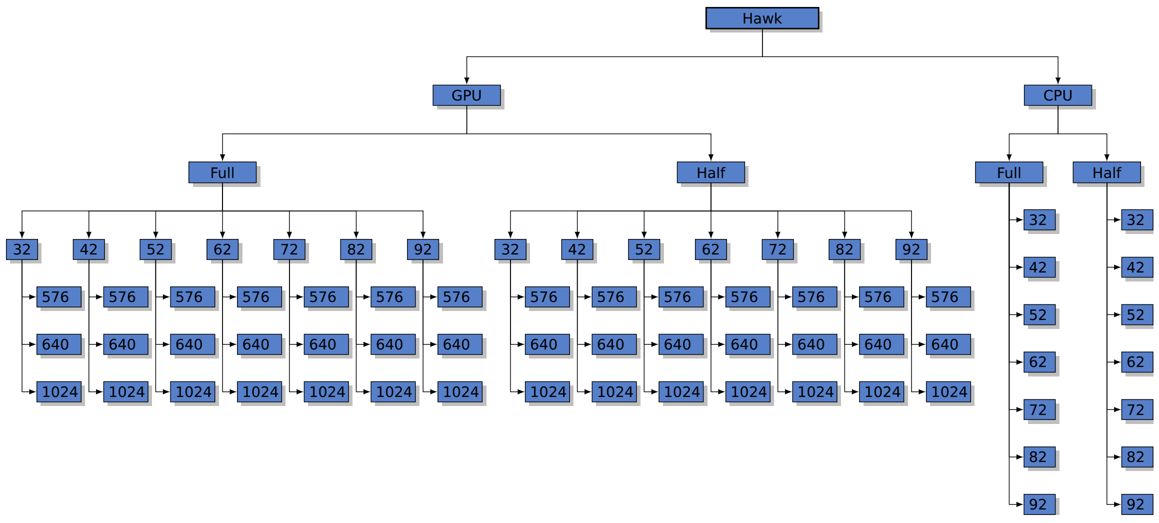


**Figure B.2: The generated experiments for each data ordering, and with 100 and 1100 time steps.**

# Appendix C

# Hawk System Scripts

## C.1 Main Execution Script

The script “runAnyOrAll.sh” can be executed by giving the required parameters which are integer number 0 or 1 to execute the script with full or half force type computation, respectively, text description “full”/“half”, and data ordering index to order the data according to the supported application ordering.

```
#!/bin/bash
#elif statements
#declare -a sizelist=(32)

declare -a sizelist=(32 42 52 62 72 82 92)
declare -a typelist=(0 1)
declare -a typeName=("full" "half")
declare -a BSizelist=(576 640 1024)
declare -a orderings=(0 1 2 3 4 5 6)
declare -a ordfolders=("00_Linear" "01_Hilbert" "02_Morton" "03
    _Hilbert4Row" "04_Hilbert4Col" "05_Morton4Row" "06_Morton4Col");

sizelength=${#sizelist[@]}
typelength=${#typelist[@]}
BSlength=${#BSizelist[@]}
filename="00_miniMD_any.job"

```

```
outfileprec="slurm-"
outfilepost=".out"
myd=$(date +"%d%b%Y")
myd="O"${myd}

nodename="ccs2012"
#==============================



funExecMiniMD(){
# ------------------------------------------------------------------------------#
# 1.) $1 Integer type represents force type 0 for full and 1 for half
# 2.) $2 String type represents force type full/half
# 3.) $3 Order , integer type
# ------------------------------------------------------------------------------#
 ordfolder=${ordfolders[$3]}
 ordName=$( echo ${ordfolder} | cut -d '_' -f 2 )

 echo "========= FUNCTION STARTED =================="
 echo "results will be written in " ${ordfolder} "."
 echo " 1) Order      = (" ${ordName} "," $3 ")"
 echo " 2) force type = ("  $2        "," $1 ")"
 echo "    1.) Prepare Job       "
 echo "---------------------------------------------------"
 (exec ./createJob.sh $1 $3 )

for (( i=1; i<${sizelength}+1; i++ ));
do
  for (( b=1; b<${BSlength}+1; b++ ));
```

```
do
  echo "== The file in.lj.miniMD preparation taks =="
  (exec ./creatCnfgfile.sh ${sizelist[$i-1]} ${BSizelist[$b-1]} )
  echo "==========> Prepreation completed          =="
                echo ""
                echo ""
                echo "====         Job SUPMISSION PHASE           ==="

                jobId=$(sbatch -w ${nodename} --parsable ${filename
})
                echo "==========> Job Id is = " ${jobId}

               # (exec squeue |grep ${JobId})
  while :
  do
                  RUNNING=$(squeue |grep $jobId )      #c.c1551517`
                          echo "=======> Runing Status " ${RUNNING}
     if [[ -z "${RUNNING// }" ]]; then
       break
     fi
     sleep 2
     echo "Still running with " ${RUNNING}
  done
  echo "===========> Runing Block Size" ${BSizelist[$b-1]}   "
COMPLETED    ==="
          (exec mv ${outfileprec}$jobId${outfilepost} ib${sizelist[
$i-1]}.${ordName}.${myd}.$2.BS${BSizelist[$b-1]}.txt )

          (exec mv ib${sizelist[$i-1]}.${ordName}.${myd}.$2.BS${
BSizelist[$b-1]}.txt analysis/${myd}/${ordfolder}/$2/size${
sizelist[$i-1]}/.)
  echo "
=====================================================================
"

```

```

            echo ""
      echo ""
                  echo ""
      sleep 1
    done
      echo "===> Work With Size " ${sizelist[$i-1]} " COMPLETED
    ==="

done
}
#
    ##########################################################################

###                     Main Program            ###
###                   ###
### 1. $1 represents the Order Number              ###
### 2. $2 represents the force type to run either full=0 half=1
    ###
#
    ##########################################################################

if [ "$1" == "All" ];
then
#     echo "Selected Order All"
#     echo "================="
      for ord in {0..6}
      do
          #if [[ ( $ord -eq 0 ) || ( $ord -eq 1 ) ]];
          #then
          #     continue
          #fi
          echo "Order Number is " ${ordfolders[$ord]} " Number " ${
    orderings[$ord]}

```

```
        if [ "$2" == "full" ];
        then
            echo "For " $1 " And Order" ${ordfolders[$ord]}
             funExecMiniMD 0 "full" ${orderings[$ord]}
        elif [ "$2" == "half" ];
        then
            echo "For " $2 " And Order" ${ordfolders[$ord]}
             funExecMiniMD 1 "half" ${orderings[$ord]}
        elif [ "$2" == "both" ];
        then
            echo "For " $2 " And Order" ${ordfolders[$ord]}
              funExecMiniMD 0 "full" ${orderings[$ord]}
              funExecMiniMD 1 "half" ${orderings[$ord]}

        else
        echo "          Error Undefined Parameter."
        fi
    done
elif [[ ("$1" -ge 0 ) && ( "$1" -le 6 ) ]];
then

        echo "Order Number is " ${ordfolders[$1]} " Number " ${
    orderings[$1]}

        if [ "$2" == "full" ];
        then
            echo "For " $2 " And Order name " ${ordfolders[$1]} " ==
     " ${orderings[$1]}
             funExecMiniMD 0 "full" ${orderings[$1]}
        elif [ "$2" == "half" ];
  then
     echo "For " $2 " And Order name " ${ordfolders[$1]} " == " ${
    orderings[$1]}
              funExecMiniMD 1 "half" ${orderings[$1]}
        elif [ "$2" == "both" ];
        then
            echo "For " $2 " And Order name " ${ordfolders[$1]} " ==
    " ${orderings[$1]}
             funExecMiniMD 0 "full" ${orderings[$1]}
             funExecMiniMD 1 "half" ${orderings[$1]}

        else
        echo "            Error Undefined Parameter."
        fi

fi
```

## C.2 Create Configuration File

The above script will call this script to modifie the experiments simulation size and block size configurations as shown below.

```
#!/bin/bash
#
filename=in.lj.miniMD
(exec cp in.lj.Reset $filename )
sed -i "s/4096/$2/g" $filename
sed -i "s/32 32 32/$1 $1 $1/g" $filename
```

## C.3 Create Batch Job

The script“runAnyOrAll.sh” will uses this batch job which will be executed on the defined node on the script and the job id will be used to check the end of the execution in order to execute the next simulation size or the next block size.

```
#!/bin/bash
#----------------------------------------------------------
```

```
# $1: force type :0 for full and 1 for half
# $2: Order type from 0 to 6
#------------------------------------------------------------
if [ $# -eq 0 ]
  then
    echo "No arguments supplied"
    exit
fi


if [[ ( "$1" -eq 0 ) || ( "$1" -eq 1) ]];
then
  if [[ ( "$2" -ge 0 ) && ( "$2" -le 6 ) ]];
  then

    filename=00_miniMD_any.job
    (exec cp 00_master.job $filename )
    sed -i "s/--half_neigh 0 --order 0/--half_neigh $1 --order $2/g"
     $filename
  fi
fi
```

As an example the following job is generated where the force computation is set to half neighbour list force computation and the selected order is hybrid Morton and column-major ordering.

```
#!/bin/bash

#SBATCH --job-name=cpu_MD

#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --gres gpu:1

```

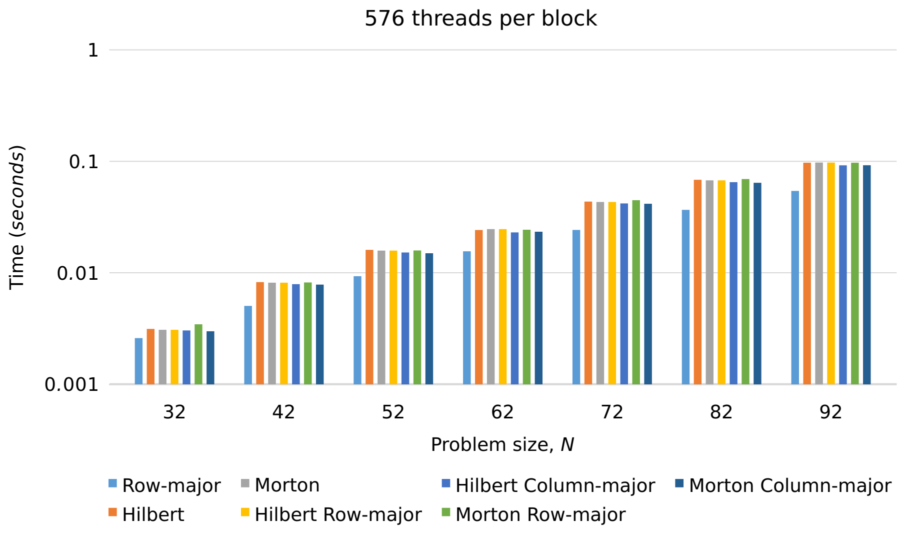
```
module load CUDA/9.1
module load mpi

MYPATH=/home/$USER/workspace/001_miniMD
EXECPATH=/home/$USER/workspace/001_miniMD
#Go to work directory
cd ${MYPATH}
echo "running "
mpirun -np 1  ./miniMD_openmpi --half_neigh 1 --order 6
echo "finish"
```

# Appendix D

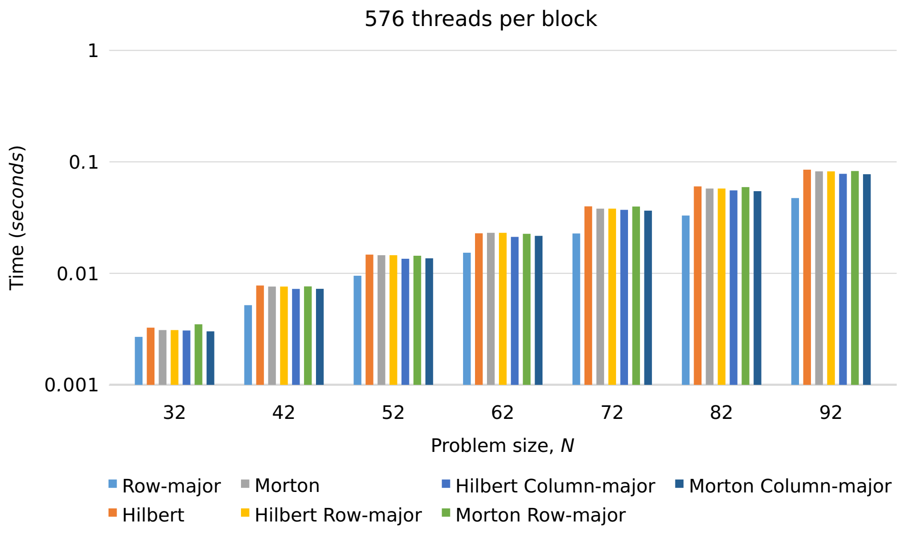
# Hawk Execution Time

Chapter 6 provided the execution time of full and half neighbour list force computation kernels on the GPU for one time step with thread block size 1024. This appendix shows the execution time for both full and half neighbour list cases for block sizes 576 and 640.

## D.1 Block Size: 576



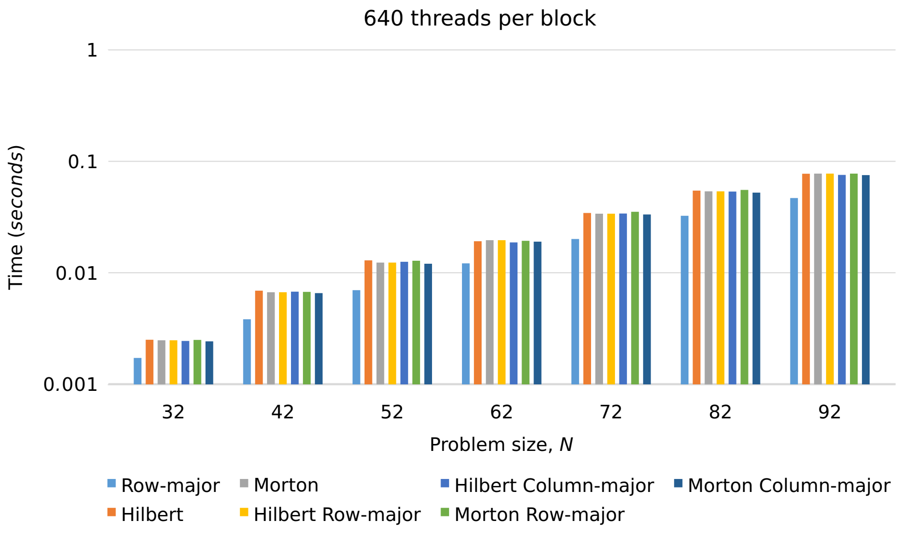
(a)



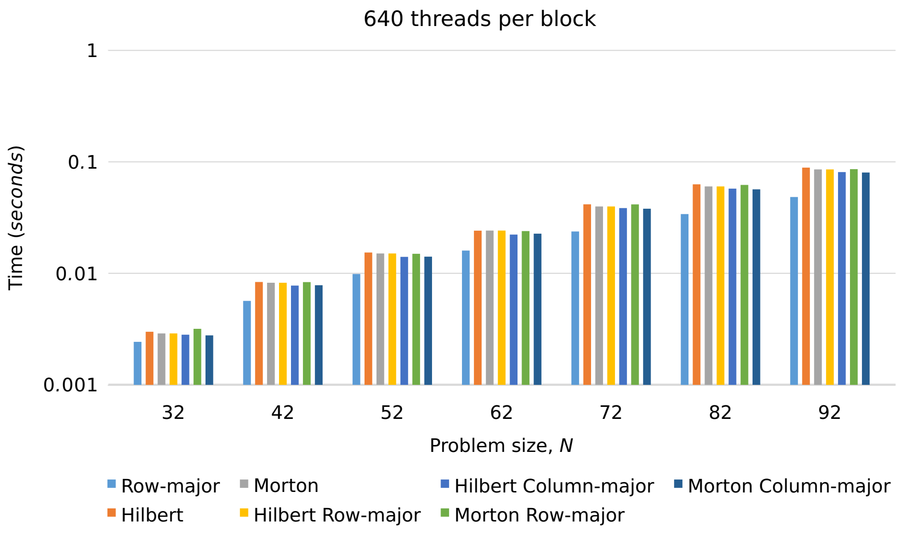
(b)

**Figure D.1: Hawk: time for execution of force computation kernel on the GPU for one time step in the full (upper) and half (lower) neighbour list cases .**

## D.2 Block Size: 640



(a)



(b)

**Figure D.2: Hawk: time for execution of force computation kernel on the GPU for one time step in the full (upper) and half (lower) neighbour list cases .**

# Bibliography

[1] 5.3. Input script structure — LAMMPS documentation. https://lammps.sandia.gov/doc/Commands_structure.html.

[2] *Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188544. URL http://doi.acm.org/10.1145/1188455.1188544.

[3] 5 Things You Should Know About the New Maxwell GPU Architecture. https://devblogs.nvidia.com/5-things-you-should-know-about-new-maxwell-gpu-architecture/, February 2014.

[4] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019. doi: 10.1109/IEEESTD.2019.8766229.

[5] Coffee lake - microarchitectures - intel, Jan 2019. URL https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake. Accessed: 2019-01-28.

[6] Ibrahim Al-Kharusi and David W Walker. Locality properties of 3D data orderings with application to parallel molecular dynamics simulations. *The International Journal of High Performance Computing Applications*, page 109434201984628, May 2019. ISSN 1094-3420, 1741-2846. doi: 10.1177/1094342019846282.

[7] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):1, December 2009. ISSN 07300301. doi: 10.1145/1618452.1618500.

[8] Paul Alcorn. Intel coffee lake vs. kaby lake: A side-by-side comparison, sep 2017. URL `https://www.tomshardware.co.uk/intel-coffee-lake-kaby-lake,news-56880.html`. Accessed:2019-01-28.

[9] Paul Alcorn. Intel's new roadmap revealed: 10nm ice lake in 2020, 14nm cooper lake 2019, aug 2018. URL `https://www.tomshardware.co.uk/intel-roadmap-cooper_lake-ice_lake,news-58945.html`.

[10] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids (Oxford Science Publications)*. Oxford science publications. Oxford University Press, reprint edition, #jun# 1989. ISBN 0-19-855645-4. Published: Paperback.

[11] Ramu Anandakrishnan and Alexey V. Onufriev. An N log N approximation based on the natural organization of biomolecules for speeding up the computation of long range interactions. *Journal of Computational Chemistry*, 31(4): 691–706, 2010. ISSN 1096-987X. doi: 10.1002/jcc.21357.

[12] Ramu Anandakrishnan, Tom R. W. Scogland, Andrew T. Fenley, John C. Gordon, Wu-chun Feng, and Alexey V. Onufriev. Accelerating electrostatic surface potential calculation with multi-scale approximation on graphics processing units. *Journal of Molecular Graphics and Modelling*, 28(8):904 – 910, 2010. ISSN 1093-3263. doi: http://dx.doi.org/10.1016/j.jmgm.2010.04.001.

[13] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Principles and Pratice of Parallel Programming, PPoPP'91*, volume Volume 26, pages Pages 39–50, Williamsburg, Virginia,, United States, April 1991. doi: 10.1145/109626.109631. 12 pages.

[14] Hans C Andersen. Rattle: A "velocity" version of the shake algorithm for molecular dynamics calculations. *Journal of Computational Physics*, 52(1): 24–34, 1983. ISSN 0021-9991. doi: 10.1016/0021-9991(83)90014-1. URL `https://doi.org/10.1016/0021-9991(83)90014-1`.

[15] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/j.jcp.2008.01.047.

[16] E. Athanasaki and N. Koziris. Fast indexing for blocked array layouts to improve multi-level cache locality. In *Eighth Workshop on Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004.*, pages 109–119, Madrid, Spain, 2004. IEEE. ISBN 978-0-7695-2061-2. doi: 10.1109/INTERA.2004.1299515.

[17] Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Texts in Computational Science and Engineering. Springer-Verlag, Berlin Heidelberg, 2013. ISBN 978-3-642-31045-4.

[18] Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Texts in Computational Science and Engineering. Springer-Verlag, Berlin Heidelberg, 2013. ISBN 978-3-642-31045-4.

[19] T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, 15(6):658–664, Nov 1969. ISSN 0018-9448. doi: 10.1109/TIT.1969.1054385.

[20] Daniel Bundala and Jakub Zavodny. Optimal Sorting Networks. *arXiv:1310.6271 [cs]*, October 2013.

[21] A. R. Butz. Alternative algorithm for hilbert's space-filling curve. *IEEE Trans. Comput.*, 20(4):424–426, #apr# 1971. ISSN 0018-9340. doi: 10.1109/T-C.1971.223258. URL `http://dx.doi.org/10.1109/T-C.1971.223258`.

[22] Arthur R. Butz. Space filling curves and mathematical programming. *Information and Control*, 12(4):314 – 330, 1968. ISSN 0019-9958. doi: http://dx.doi.org/10.1016/S0019-9958(68)90367-7. URL //www.sciencedirect.com/science/article/pii/S0019995868903677.

[23] Katharine Castle. Intel core cpus: Everything we know about intel's 8th and 9th gen coffee lake refresh processors, January 2019. URL https://www.rockpapershotgun.com/2019/01/24/intel-core-cpu-coffee-lake-price-specs-guide/. Accessed:2019-01-28.

[24] Daniel Cederman and Philippas Tsigas. GPU-Quicksort: A practical Quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14: 1.4, December 2009. ISSN 10846654. doi: 10.1145/1498698.1564500.

[25] Deukhyun Cha, Qin Zhang, Jesmin Jahan Tithi, Alexander Rand, Rezaul A. Chowdhury, and Chandrajit Bajaj. Accelerated Molecular Mechanical and Solvation Energetics on Multicore CPUs and Manycore GPUs. In *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, BCB '15, pages 222–231, Atlanta, Georgia, 2015. ACM. ISBN 978-1-4503-3853-0. doi: 10.1145/2808719.2808742.

[26] Timothy M. Chan. A Minimalist's Implementation of an Approximate Nearest Neighbor Algorithm in Fixed Dimensions, May 2006.

[27] Rohit Chandra, editor. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, San Francisco, CA, 2001. ISBN 978-1-55860-671-5.

[28] Ningtao Chen, Nengchao Wang, and Baochang Shi. A new algorithm for encoding and decoding the hilbert order. *Software: Practice and Experience*, 37(8):897–908, 2007. ISSN 0038-0644. doi: 10.1002/spe.v37:8. URL https://doi.org/10.1002/spe.v37:8.

[29] Rezaul Chowdhury and Chandrajit Bajaj. Algorithms for Faster Molecular Energetics, Forces and Interfaces. ICES REPORT 10-32, The Institute for Computational Engineering and Sciences, August 2010.

[30] Rezaul Chowdhury, Dmitri Beglov, Mohammad Moghadasi, Ioannis Ch Paschalidis, Pirooz Vakili, Sandor Vajda, Chandrajit Bajaj, and Dima Kozakov. Efficient Maintenance and Update of Nonbonded Lists in Macromolecular Simulations. *Journal of Chemical Theory and Computation*, 10(10):4449–4454, 2014. doi: 10.1021/ct400474w.

[31] Michael Connor and Piyush Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization & Computer Graphics*, 16(4):599–608, 2010. ISSN 1077-2626. doi: doi.ieeecomputersociety.org/10.1109/TVCG.2010.9.

[32] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Elsevier, MK, Amsterdam ; Boston, 2013. ISBN 978-0-12-415933-4. OCLC: ocn773025100.

[33] NVIDIA Corporation. The CUDA Compiler Driver NVCC, August 2010.

[34] Nvidia Corporation. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, version 4.2 edition, 2012. URL `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[35] NVIDIA Corporation. NVIDIA GeForce GTX 980 Whitepaper, 2014.

[36] Paul Stewart Crozier, Heidi K. Thornquist, Robert W. Numrich, Alan B. Williams, Harold Carter Edwards, Eric Richard Keiter, Mahesh Rajan, James M. Willenbring, Douglas W. Doerfler, and Michael Allen Heroux. Improving performance via mini-applications. Technical Report SAND2009-5574, 993908, September 2009.

[37] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1):1 – 143, 1997. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(96)00146-6. URL `http://www.sciencedirect.com/science/article/pii/S0304397596001466`.

[38] Murray S. Daw and M. I. Baskes. Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals. *Physical Review B*, 29(12):6443–6453, June 1984. ISSN 0163-1829. doi: 10.1103/PhysRevB.29.6443.

[39] Murray S. Daw, Stephen M. Foiles, and Michael I. Baskes. The embedded-atom method: A review of theory and applications. *Materials Science Reports*, 9(7): 251–310, March 1993. ISSN 0920-2307. doi: 10.1016/0920-2307(93)90001-U.

[40] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. In *2012 41st International Conference on Parallel Processing*, pages 350–359, September 2012. doi: 10.1109/ICPP.2012.19.

[41] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990. ISSN 00983500. doi: 10.1145/77626.79170.

[42] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu.com, 2012. ISBN 1-257-99254-6 978-1-257-99254-6.

[43] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 247–252. ACM, 1989.

[44] Rob Farber. *CUDA Application Design and Development*. Applications of GPU Computing Series. Morgan Kaufmann, Waltham, MA, 2011. ISBN 978-0-12-388426-8. OCLC: ocn731925404.

[45] Chunsheng Feng, Shi Shu, Junxian Wang, and Zheng Li. The parallel generation of 2-d hilbert space-filling curve on gpu. In *Biomedical Engineering and Informatics (BMEI), 2012 5th International Conference on*, pages 1359–1362. IEEE, 2012.

[46] Chunsheng Feng, Shi Shu, Junxian Wang, and Zheng Li. The parallel generation of 2-D Hilbert Space-filling Curve on GPU. In *Biomedical Engineering and Informatics (BMEI), 2012 5th International Conference On*, pages 1359–1362. IEEE, 2012.

[47] M.J. Field. *A Practical Introduction to the Simulation of Molecular Systems*. Cambridge University Press, 1999. ISBN 9780521581295. URL `https://books.google.co.uk/books?id=mPgpMig3tx0C`.

[48] A. J. Fisher. A new algorithm for generating hilbert curves. *Software: Practice and Experience*, 16(1):5–12, 1986. ISSN 1097-024X. doi: 10.1002/spe.4380160103. URL `http://dx.doi.org/10.1002/spe.4380160103`.

[49] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

[50] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation: From Algorithms to Applications*, volume 1. Academic Press, Inc, 2nd edition, 2001. ISBN 0-12-267351-4.

[51] Steven T. Gabriel and David S. Wise. The Opie compiler from row-major source to Morton-ordered matrices. In *Proceedings of the 3rd Workshop on Memory Performance Issues in Conjunction with the 31st International Symposium on Computer Architecture - WMPI '04*, pages 136–144, Munich, Germany, 2004. ACM Press. ISBN 978-1-59593-040-8. doi: 10.1145/1054943.1054962.

[52] Petr Gajdoš, Tomáš Ježowicz, Vojtěch Uher, and Pavel Dohnálek. A parallel Fruchterman–Reingold algorithm optimized for fast visualization of large

graphs and swarms of data. *Swarm and Evolutionary Computation*, 26:56–63, February 2016. ISSN 22106502. doi: 10.1016/j.swevo.2015.07.006.

[53] Jens Glaser, Trung Dac Nguyen, Joshua A. Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse, and Sharon C. Glotzer. Strong scaling of general-purpose molecular dynamics simulations on {GPUs}. *Computer Physics Communications*, 192:97 – 107, 2015. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2015.02.028.

[54] GHCF Golub and F CHARLES. Van loan, 1989: Matrix computations.

[55] Pedro Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry*, 28(2):570–573, 2007. ISSN 1096-987X. doi: 10.1002/jcc.20563.

[56] Pedro Gonnet. Pseudo-Verlet Lists: A new, compact neighbour list representation. *Molecular Simulation*, 39(9):721–727, August 2013. ISSN 0892-7022, 1029-0435. doi: 10.1080/08927022.2012.762097.

[57] Linh Ha, Jens Kruger, and Claudio T Silva. Implicit radix sorting on GPUs. page 15.

[58] Linh Ha, Jens Krüger, and Cláudio T. Silva. Fast Four-Way Parallel Radix Sorting on GPUs. *Computer Graphics Forum*, 28(8):2368–2378, December 2009. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2009.01542.x.

[59] Gundolf Haase, Manfred Liebmann, and Gernot Plank. A Hilbert-order multiplication scheme for unstructured sparse matrices. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):213–220, August 2007. ISSN 1744-5760. doi: 10.1080/17445760601122084.

[60] Jean-Pierre Hansen and Ian Ranald McDonald. *Theory of Simple Liquids*.

Elsevier, third edition, 2006. ISBN 978-0-12-370535-8. doi: 10.1016/B978-0-12-370535-8.X5000-9.

[61] David J. Hardy, John E. Stone, and Klaus Schulten. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Computing*, 35 (3):164 – 177, 2009. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2008.12.005.

[62] Sarah Harris and David Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition edition. ISBN 978-0-12-394424-5.

[63] Herman Haverkort. How many three-dimensional hilbert curves are there? *arXiv preprint arXiv:1610.00155*, 2016.

[64] Herman J. Haverkort. An inventory of three-dimensional hilbert space-filling curves. *CoRR*, abs/1109.2323, 2011. URL `http://arxiv.org/abs/1109.2323`.

[65] Warren.J. Hehre. *A Guide to Molecular Mechanics and Quantum Chemical Calculations*. Wavefunction, 2003. ISBN 978-1-890661-18-2.

[66] Michael P. Howard, Joshua A. Anderson, Arash Nikoubashman, Sharon C. Glotzer, and Athanassios Z. Panagiotopoulos. Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units. *Computer Physics Communications*, 203:45–52, June 2016. ISSN 00104655. doi: 10.1016/j.cpc.2016.02.003.

[67] B. Huang, J. Gao, and X. Li. An Empirically Optimized Radix Sort for GPU. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 234–241, August 2009. doi: 10.1109/ISPA.2009.89.

[68] Chi-Yen Huang and Yu-Wei Roy Chen. Linear quadtree construction in real time. *J. Inf. Sci. Eng.*, 26:1917–1930, 2010.

[69] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM. ISBN 978-0-89791-252-5. doi: 10.1145/73560.73588.

[70] Jacob N Israelachvili. *Intermolecular and Surface Forces*. Elsevier, Academic Press, Amsterdam, third edition, 2011. ISBN 978-0-12-391927-4 978-0-12-375182-9.

[71] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992. ISBN 0-201-54856-9.

[72] Frank Jensen. *Introduction to Computational Chemistry*. John Wiley & Sons, Chichester, England ; Hoboken, NJ, 2nd ed edition, 2007. ISBN 978-0-470-01186-7 978-0-470-01187-4. OCLC: ocm70707839.

[73] Jaeheon Jeong, Per Stenström, and Michel Dubois. Simple penalty-sensitive replacement policies for caches. In *Proceedings of the 3rd Conference on Computing Frontiers - CF '06*, page 341, Ischia, Italy, 2006. ACM Press. ISBN 978-1-59593-302-7. doi: 10.1145/1128022.1128068.

[74] Peng Jiang, Linchuan Chen, and Gagan Agrawal. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 16:1–16:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4361-9. doi: 10.1145/2925426.2926285.

[75] Norman P Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. page 46, .

[76] Norman P Jouppi. Reducing Compulsory and Capacity Misses. Technical report, .

[77] Krzysztof Kaczmarski. *Experimental B+-Tree for GPU*.

[78] David Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. Chapter 1 - Introduction. In David KaeliPerhaad MistryDana SchaaDong Ping Zhang, editor, *Heterogeneous Computing with OpenCL 2.0 (Third Edition)*, pages 1 – 14. Morgan Kaufmann, Boston, third edition edition, 2015. ISBN 978-0-12-801414-1. doi: 10.1016/B978-0-12-801414-1.00001-6.

[79] Kamran Karimi. A Performance Comparison of CUDA and OpenCL. page 10.

[80] Sergey Kazachenko, Mark Giovinazzo, Kyle Wm. Hall, and Natalie M. Cann. Algorithms for GPU-based molecular dynamics simulations of complex fluids: Applications to water, mixtures, and liquid crystals. *Journal of Computational Chemistry*, 36(24):1787–1804, 2015. ISSN 1096-987X. doi: 10.1002/jcc.24000.

[81] Khronos OpenCL Working Group. *The OpenCL Specification, Version 2.0,Document Revision: 48*. July 2015. URL `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`.

[82] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 International Conference on Management of Data - SIGMOD '10*, page 339, Indianapolis, Indiana, USA, 2010. ACM Press. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807206.

[83] Haridimos Kondylakis, Kostas Zoumpatianos, Niv Dayan, and Themis Palpanas. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. page 14.

[84] Markus Kowarschik and Christian Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies — Advanced Lectures, Volume 2625 of Lecture Notes in Computer Science*, pages 213–232. Springer, 2003.

[85] Jack B. Kuipers. *Quaternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality*. Princeton University Press, Princeton, N.J, 5th edition, August 2002. ISBN 978-0-691-05872-6.

[86] Manaschai Kunaseth, Ken-ichi Nomura, Hikmet Dursun, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. Memory-Access Optimization of Parallel Molecular Dynamics Simulation via Dynamic Data Reordering. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, Lecture Notes in Computer Science, pages 781–792. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32820-6.

[87] Sandia National Laboratories. Mantevo project. https://mantevo.org/, Sep 2011. URL `https://mantevo.org/`. Last update: 9, September 2019.

[88] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.*, 26(4): 63–74, #apr# 1991. ISSN 0362-1340. doi: 10.1145/106973.106981.

[89] J.K. Lawder. Calculation of mappings between one and n-dimensional values using the hilbert space-filling curve. Technical Report JL1/00, School of COmputer Science and Information Systems, Birkbeck College,University of London, UK, 8 2000. Technical Report no. JL1/00.

[90] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06*, page 579, Boston, Massachusetts, 2006. ACM Press. ISBN 978-1-59593-364-5. doi: 10.1145/1179352.1141926.

[91] Errol Lewars. *Computational Chemistry: Introduction to the Theory and Applications of Molecular and Quantum Mechanics*. Springer, Dordrecht [Netherlands] ; London ; New York, 2nd ed edition, 2011. ISBN 978-90-481-3860-9 978-90-481-3861-6. OCLC: ocn502425604.

[92] Wan-Qing Li, Tang Ying, Wan Jian, and Dong-Jin Yu. Comparison research on the neighbor list algorithms: Verlet table and linked-cell. *Computer Physics*

*Communications*, 181(10):1682 – 1686, 2010. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2010.06.005.

[93] Aristid Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3): 280–299, March 1968. ISSN 0022-5193. doi: 10.1016/0022-5193(68)90079-9.

[94] Xian Liu and Günther F Schrack. An algorithm for encoding and decoding the 3-d hilbert order. *IEEE transactions on image processing*, 6(9):1333–1337, 1997.

[95] K. Patrick Lorton and David S. Wise. Analyzing Block Locality in Morton-order and Morton-hybrid Matrices. *SIGARCH Comput. Archit. News*, 35(4): 6–12, September 2007. ISSN 0163-5964. doi: 10.1145/1327312.1327315.

[96] Jinping Luo and Lijun Liu. Optimisation of data locality in energy calculations for large-scale molecular dynamics simulations. *Molecular Simulation*, 43(4): 284–290, 2017. doi: 10.1080/08927022.2016.1267354.

[97] Emanuele Manca, Andrea Manconi, Alessandro Orro, Giuliano Armano, and Luciano Milanesi. CUDA-quicksort: An improved GPU-based implementation of quicksort: CUDA-QUICKSORT. *Concurrency and Computation: Practice and Experience*, 28(1):21–43, January 2016. ISSN 15320626. doi: 10.1002/cpe.3611.

[98] William Mattson and Betsy M. Rice. Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2–3):135 – 148, 1999. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/S0010-4655(98)00203-3.

[99] Michael D McCool, Chris Wales, and Kevin Moule. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. page 8.

[100] Muralidhar Medidi and Narsingh Deo. Parallel Dictionaries Using AVL Trees. *Journal of Parallel and Distributed Computing*, 49(1):146–155, February 1998. ISSN 07437315. doi: 10.1006/jpdc.1998.1432.

[101] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001. ISSN 1573-7640. doi: 10.1023/A:1011119519789.

[102] Simone Meloni, Mario Rosati, and Luciano Colombo. Efficient particle labeling in atomistic simulations. *The Journal of Chemical Physics*, 126(12):121102, 2007. doi: 10.1063/1.2719690.

[103] A. S. Minkin, A. B. Teslyuk, A. A. Knizhnik, and B. V. Potapkin. GPGPU performance evaluation of some basic molecular dynamics algorithms. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 629–634, July 2015. doi: 10.1109/HPCSim.2015.7237104.

[104] David M. Mount and Sunil Arya. ANN - Approximate Nearest Neighbor Library. http://www.cs.umd.edu/~mount/ANN/, January 2010. version 1.1.2.

[105] C. Muelder and K. Ma. Rapid Graph Layout Using Space Filling Curves. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1301–1308, November 2008. ISSN 1077-2626. doi: 10.1109/TVCG.2008.158.

[106] École nationale supérieure des mines de Paris. Centre d'Automatique et Informatique, F. Irigoin, and R. Triolet. *Computing Dependence Direction Vectors and Dependence Cones with Linear Systems*. ENSMP-CAI. Ecole Nationale Supérieure des Mines de Paris, Centre d'Automatique et Informatique, 1987.

[107] Anthony E. Nocentino and Philip J. Rhodes. Optimizing memory access on GPUs using morton order indexing. In *Proceedings of the 48th Annual Southeast Regional Conference on - ACM SE '10*, page 1, Oxford, Mississippi, 2010. ACM Press. ISBN 978-1-4503-0064-3. doi: 10.1145/1900008.1900035.

[108] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 978-0-12-374260-5.

[109] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, San Francisco, Calif, 1997. ISBN 978-1-55860-339-4.

[110] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004. ISSN 01966774. doi: 10.1016/j.jalgor.2003.12.002.

[111] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988. ISSN 00010782. doi: 10.1145/63039.63042.

[112] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann series in computer architecture and design. Elsevier/Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, Amsterdam ; Boston, fifth edition edition, 2014. ISBN 978-0-12-407726-3. OCLC: ocn859555917.

[113] Kipfer Peter and Westermann Rüdiger. GPU Gems 2. `https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter46.html`.

[114] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1 – 19, 1995. ISSN 0021-9991. doi: http://dx.doi.org/10.1006/jcph.1995.1039. URL `http://www.sciencedirect.com/science/article/pii/S002199918571039X`.

[115] S. Pontarelli, P. Reviriego, and J. A. Maestro. Parallel d-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput. *IEEE Transactions on Computers*, 65(1):326–331, January 2016. ISSN 0018-9340. doi: 10.1109/TC.2015.2417524.

[116] William H. Press and Saul A. Teukolsky. Portable Random Number Generators. *Computers in Physics*, 6(5):522, 1992. ISSN 08941866. doi: 10.1063/1.4823101.

[117] Milcho Prisagjanec and Pece Mitrevski. Reducing Competitive Cache Misses in Modern Processor Architectures. *International Journal of Computer Science and Information Technology*, 8(6):49–57, December 2016. ISSN 09754660, 09753826. doi: 10.5121/ijcsit.2016.8605.

[118] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1990. ISBN 0-387-97297-8.

[119] R Purser. *Hilbert Curves Isometrically Filling a Spherical Shell, and Their Application to the Estimation of Spatial Data Density.* May 2018. doi: 10.7289/V5/ON-NCEP-494.

[120] Rajeev Raman and David Stephen Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, 2008.

[121] Rachel N. Robey, David Nicholaeff, and Robert W. Robey. Hash-Based Algorithms for Discretized Data. *SIAM Journal on Scientific Computing*, 35 (4):C346–C368, January 2013. ISSN 1064-8275, 1095-7197. doi: 10.1137/120873686.

[122] Hans Sagan. A three-dimensional hilbert curve. *International Journal of Mathematical Education in Science and Technology*, 24(4):541–545, 1993. doi: 10.1080/0020739930240405. URL `http://dx.doi.org/10.1080/0020739930240405`.

[123] Hans Sagan. *Space-filling curves*. Universitext Series. Springer-Verlag, 1994. ISBN 9780387942650. doi: 10.1007/978-1-4612-08716. URL `https://books.google.co.uk/books?id=gUfvAAAAMAAJ`.

[124] Fatima K. Abu Salem and Mira Al Arab. Comparative study of space filling

curves for cache oblivious TU decomposition. *CoRR*, abs/1612.06069, 2016. URL `http://arxiv.org/abs/1612.06069`.

[125] Jesús Sánchez. *SMART MEMORY MANAGEMENT THROUGH LOCALITY ANALYSIS*. doctoralThesis, UNIVERSITAT POLITÈCNICA DE CATALUNYA, Barcelona (SPAIN), 2001.

[126] Shankar P. Sastry, Emre Kultursay, Suzanne M. Shontz, and Mahmut T. Kandemir. Improved cache utilization and preconditioner efficiency through use of a space-filling curve mesh element- and vertex-reordering technique. *Engineering with Computers*, 30(4):535–547, October 2014. ISSN 0177-0667, 1435-5663. doi: 10.1007/s00366-014-0363-0.

[127] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10, Rome, Italy, May 2009. IEEE. ISBN 978-1-4244-3751-1. doi: 10.1109/IPDPS.2009.5161005.

[128] Robert D. Skeel, Ismail Tezcan, and David J. Hardy. Multiple grid methods for classical molecular dynamics. *Journal of Computational Chemistry*, 23(6): 673–684, 2002. ISSN 1096-987X. doi: 10.1002/jcc.10072.

[129] Justin R. Smith. *The Design and Analysis of Parallel Algorithms*. Oxford University Press, Inc., New York, NY, USA, 1993. ISBN 0-19-507881-0.

[130] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, September 2004. ISSN 00045411. doi: 10.1145/1017460.1017464.

[131] Leo Stocco and Günther Schrack. Integer dilation and contraction for quadtrees and octrees. In *Communications, Computers, and Signal Processing, 1995. Proceedings., IEEE Pacific Rim Conference on*, pages 426–428. IEEE, 1995.

[132] Anthony J. Stone. *The Theory of Intermolecular Forces*. Oxford University Press, Great Clarendon Street;Oxford; ox2 6DP;United Kingdom, 2nd edition edition, 2013. ISBN 978-0-19-967239-4,.

[133] D. Strnad and A. Nerat. Parallel construction of classification trees on a GPU: PARALLEL CONSTRUCTION OF CLASSIFICATION TREES ON A GPU. *Concurrency and Computation: Practice and Experience*, 28(5):1417–1436, April 2016. ISSN 15320626. doi: 10.1002/cpe.3660.

[134] G. Sutmann and V. Stegailov. Optimization of neighbor list techniques in liquid matter simulations. *Journal of Molecular Liquids*, 125(2):197–203, April 2006. ISSN 0167-7322. doi: 10.1016/j.molliq.2005.11.029.

[135] Yu-Hang Tang and George Em Karniadakis. Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications. *Computer Physics Communications*, 185(11):2809 – 2822, 2014. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2014.06.015.

[136] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 48(2):150–158, February 1999. ISSN 0018-9340. doi: 10.1109/12.752656.

[137] Gudula Rünger Thomas Rauber. *Parallel Programming: For Multicore and Cluster Systems*. Springer-Verlag Berlin, "Berlin", 2nd edition edition, 2010. ISBN 978-3-642-04817-3 978-3-642-04818-0. doi: 10.1007/978-3-642-04818-0. URL `"http://opac.inria.fr/record=b1133063"`.

[138] Xiaonan Tian, Rengan Xu, Yonghong Yan, Sunita Chandrasekaran, Deepak Eachempati, and Barbara Chapman. Compiler transformation of nested loops for general purpose GPUs. *Concurrency and Computation: Practice and Experience*, 28(2):537–556, February 2016. ISSN 1532-0634. doi: 10.1002/cpe.3648.

[139] J. A. van Meel, A. Arnold, D. Frenkel, S. F. Portegies Zwart, and R. G.

Belleman. Harvesting graphics power for MD simulations. *Molecular Simulation*, 34(3):259–266, March 2008. ISSN 0892-7022, 1029-0435. doi: 10.1080/08927020701744295.

[140] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103, July 1967. ISSN 0031-899X. doi: 10.1103/PhysRev.159.98.

[141] Thijs J. H Vlugt, Jan P. J. M. van der Eerden, M Dijkstra, Berend Smit, and Daan Frenkel. *Introduction to Molecular Simulation and Statistical Thermodynamics*. Delft, Netherlands, 2009. OCLC: 934654365.

[142] David W Walker. Morton ordering of 2D arrays for efficient access to hierarchical memory. *The International Journal of High Performance Computing Applications*, 32(1):189–203, January 2018. ISSN 1094-3420, 1741-2846. doi: 10.1177/1094342017725568.

[143] Ulrich Welling and Guido Germano. Efficiency of linked cell algorithms. *Computer Physics Communications*, 182(3):611 – 615, 2011. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2010.11.002.

[144] D.S. Wise. Ahnentafel indexing into morton-ordered arrays, or matrix locality for free. volume 1900 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 771–773. Springer Verlag, 2000. ISBN 978-3-540-67956-1.

[145] E Wolf and S Lam. A Data Locality Optimizing Algorithm. page 15.

[146] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991. ISSN 1045-9219. doi: 10.1109/71.97902. URL `https://doi.org/10.1109/71.97902`.

[147] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, Toronto, Ontario, Canada, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113449. URL http://doi.acm.org/10.1145/113445.113449.

[148] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, August 1986. ISSN 0885-7458, 1573-7640. doi: 10.1007/BF01407876.

[149] Michael Wolfe. More iteration space tiling. In *In Proceedings of the Supercomputing 89*, pages 655–664, 1989.

[150] Michael Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 43–50, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-935-0. doi: 10.1145/1735688.1735697.

[151] Michael J. Wolfe. T echniques for improving the inherent parallelism in programs. Technical Report UIUCDCS-R-78-929, University of Illinois, 1978.

[152] Don Woligroski and Igor Wallossek. Nvidia GeForce GTX 960: Maxwell In The Middle. https://www.tomshardware.co.uk/nvidia-geforce-gtx-960,review-33113.html, January 2015.

[153] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995. ISSN 01635964. doi: 10.1145/216585.216588.

[154] Zhenhua Yao, Jian-Sheng Wang, Gui-Rong Liu, and Min Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Computer Physics Communications*, 161(1–2):27 – 35, 2004. ISSN 0010-4655. doi: http://dx.doi.org/10.1016/j.cpc.2004.04.004.

[155] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing - Supercomputing '91*, pages 712–721, Albuquerque, New Mexico, United States, 1991. ACM Press. ISBN 978-0-89791-459-8. doi: 10.1145/125826.126164.

[156] Jian Zhang and Sei ichiro Kamata. A generalized 3-d hilbert scan using look-up tables. *Journal of Visual Communication and Image Representation*, 23(3):418 – 425, 2012. ISSN 1047-3203. doi: http://dx.doi.org/10.1016/j.jvcir.2011.12.005. URL `//www.sciencedirect.com/science/article/pii/S1047320311001672`.

[157] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796, August 2000. ISSN 1051-8215. doi: 10.1109/76.856455.