Journal Title
XX(X):1–17

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

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

Ibrahim Al-Kharusi[1] and David W. Walker[1]

Abstract

Application performance on graphical processing units (GPUs), in terms of execution speed and memory usage, depends on the efficient use of hierarchical memory. It is expected that enhancing data locality in molecular dynamic simulations will lower the cost of data movement across the GPU memory hierarchy. The work presented in this paper analyses the spatial data locality and data reuse characteristics for row-major, Hilbert, and Morton orderings, and the impact these have on the performance of molecular dynamics simulations. A simple cache model is presented, and this is found to give results that are consistent with the timing results for the particle force computation obtained on an NVidia GeForce GTX960 GPU. Further analysis of the observed memory use, in terms of cache hits and the number of memory transactions, 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 architecture.

Keywords

Molecular dynamics simulation, Morton ordering, Hilbert ordering, GPU, parallel algorithms

1 Introduction

Modern computer systems are characterised by deep memory hierarchies composed of main memory, multiple layers of cache, and other specialised types of memory. In parallel and distributed systems additional memory layers are added to this hierarchy. Achieving good performance for computational science applications, in terms of execution time, depends on the efficient use of hierarchical memory. Indeed, the inefficient use of hierarchical memory can result in data movement, rather than floating-point performance, dominating execution time. This is particularly true in graphical processing units (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 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 about a factor of 50 less than the peak performance.

Data locality is a key 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. Data are typically moved between a lower and a higher level in memory in fixed-size blocks, known as cache lines. Many computations and phenomena are local in nature, so if items are stored in memory based on their location, when one item is moved into a higher level of memory the other items upon which its processing depends will also be moved, thereby exploiting *spatial data locality* and improving performance. Performance is also likely to be better if items are processed in the order in which they are stored in memory because when a new item is to be processed it is likely to already be in the higher memory level. 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.

In general, the application programmer has little direct control over the scheduling of threads or the movement of data between levels in the memory hierarchy. Instead application programmers are encouraged to follow best practices in programming style that coerce the compiler and the runtime system into running code efficiently. For example, in dense linear algebra computations better performance is achieved if the computations are performed on matrix blocks through the use of Level 3 BLAS* operations[9]. This programming style has performance benefits because it results in good data locality for this class of application.

The order in which data items are accessed has a significant impact on data locality, and hence on application performance. This paper investigates the data locality properties of three ways of ordering data in a 3-dimensional array, namely the row-major, Morton, and Hilbert orderings. These orderings are described in detail in Sec. 2, and Sec. 3

[1] School of Computer Science & Informatics, Cardiff University, Cardiff CF24 3AA, UK

Corresponding author:
Ibrahim Al-Kharusi, School of Computer Science & Informatics , Cardiff University, 5 The Parade, Cardiff CF24 3AA, UK.
Email: AlKharusiI@cardiff.ac.uk

*BLAS = Basic Linear Algebra Subprograms

discusses their data locality properties. The second part of the paper applies these three orderings to 3D molecular dynamics simulations. Section 4 describes a large class of molecular dynamics simulations, while in Sec. 5 a GPU implementation of the widely-used miniMD application is introduced. Performance results for the GPU implementation for different data orderings are presented in Sec. 6, together with information gathered from the NVidia Nsight profiling tool. These results are interpreted in terms of the data locality properties of the different orderings. Related work is discussed in Sec. 7. Finally, in Sec. 8 the conclusions from this research are presented, together with some directions for future work.

2 Data Orderings

The location of an item within a 3D array of size $M \times N \times P$ may 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, we associate the row, column, and slab directions with the x, y, and z axes, respectively, where $0 \leq i < M, 0 \leq j < N$, and $0 \leq k < P$. An *ordering* of the items in a 3D array is a mapping, $\mathcal{O}$, from (i, j, k) to a linear index, b:

$$b = \mathcal{O}(i, j, k). \tag{1}$$

where $0 \leq b < MNP$ may be interpreted as the offset in memory, measured in number of items, from the position of the first item.

2.1 Linear Orderings

A row-major ordering is a type of linear ordering of the form $\mathcal{O}_R(i, j, k) = (i * ldx + j) * ldy + k$, where ldx is the offset between adjacent items in the column direction, and $ldx * ldy$ is the offset between adjacent items in the slab direction. For ease of notation it will be assumed that $ldx = M$ and $ldy = N$, so the data items form a contiguous block. Furthermore, it will be assumed without loss of generality that the 3D array is cubical so $M = N = P$. The row-major mapping in this case is therefore:

$$\mathcal{O}_R(i, j, k) = (iM + j)M + k \tag{2}$$

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.

2.2 Hilbert Ordering

The Hilbert ordering, $\mathcal{O}_H$, follows the path of the space-filling Hilbert curve through the 3D array. The Hilbert ordering requires that $M = 2^n M$ for some $M \geq 1$. A Hilbert curve can be presented as a Lindenmayer system (or *L-system)* in terms of parallel rewrite rules[22,28]. For example, a 2D Hilbert curve, such as that shown in Fig. 1, can be generated by the following rewrite rules:

$$\begin{aligned} X &\rightarrow +YF - XFX - FY + \\ Y &\rightarrow -XF + YFY + FX - \end{aligned} \tag{3}$$

where F means "draw a line segment of some specified length", $+$ means "turn 90 degrees right", and $-$ means "turn 90 degrees left". Applying these rewrite rules recursively draws a Hilbert curve to the corresponding recursive depth. For example, applying the rewrite rules once yields:

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

This draws the Hilbert curve shown in the top left 4×4 block of bins in Fig. 1. Recursively applying the rewrite rules twice draws all of the Hilbert curve shown in Fig. 1. In a similar way, a 3D Hilbert curve can also be represented as an L-system with the following rewrite rule:

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

where the meanings of the symbols are given in Table 1.

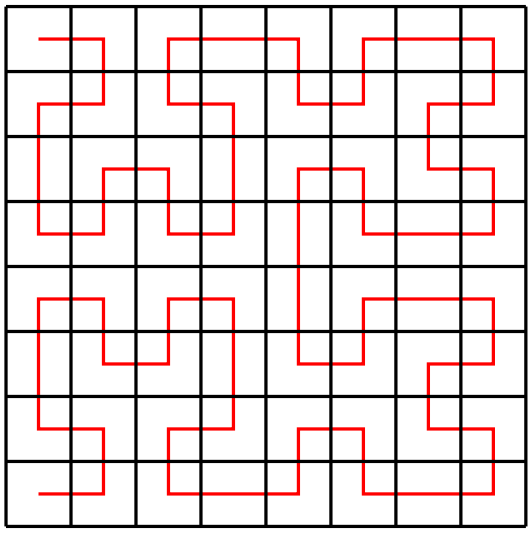


Figure 1. Two-dimensional Hilbert ordering for an 8×8 array. The index, b, increases by 1 each time the red path passes from one location to another, starting with index 0 in the top lefthand corner.

Symbol	Meaning
F	Draw line segment
$+$	Yaw 90°
$-$	Yaw -90°
$\wedge$	Pitch 90°
$\vee$	Pitch -90°
$<$	Roll 90°
$>$	Roll -90°

Table 1. Meaning of the symbols in the rewrite rule for a 3D Hilbert curve.

As we move along the 3D Hilbert curve given by the above rewrite rule it is possible to keep track of the corresponding (i, j, k) index in 3D space, in effect giving the inverse of the $\mathcal{O}_H$ mapping. Instead of the symbol F meaning "draw a line segment" it is interpreted as meaning "increment or decrement the value of i, j, or k, depending on the current orientation of the axes". In this way it is possible to initialise a 3D array giving for each bin the corresponding Hilbert index, which can subsequently be used whenever it is necessary to map between location in the 3D array and the

Hilbert index. The Hilbert ordering for a $4 \times 4 \times 4$ array is shown in Fig. 2.

The yaw, pitch and roll 90° rotations in Table 1, corresponding to the symbols $+$, $\wedge$, and $<$, can be represented by matrices:

$$Y = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix},$$
$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \tag{4}$$

The -90° rotations, corresponding to the symbols $-$, $\vee$, and $>$, are given by Y^T, P^T, and R^T. The orientation of the axes is expressed in terms of a heading, $\mathbf{H}$, and two other mutually orthogonal vectors, denoted by $\mathbf{L}$ and $\mathbf{U}$. Initially we choose $\mathbf{H} = [1\ 0\ 0]^T$, $\mathbf{L} = [0\ 0\ -1]^T$, and $\mathbf{U} = [0\ 1\ 0]^T$, and form the orientation matrix, D, that has $\mathbf{H}$, $\mathbf{L}$, and $\mathbf{U}$ as its columns. A roll corresponds to a rotation of 90° about the $\mathbf{H}$ axis; a pitch to a rotation of 90° about the $\mathbf{L}$ axis, and a yaw to a rotation of 90° about the $\mathbf{U}$ axis. Having rewritten X in Eq. 4 to the desired depth of recursion, we then process the resulting string from left to right, post-multiplying the orientation matrix by the rotation matrix corresponding to the current symbol. Thus, $D \leftarrow D\Theta$, where Θ 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 $\mathbf{H}$ (the first column of D).

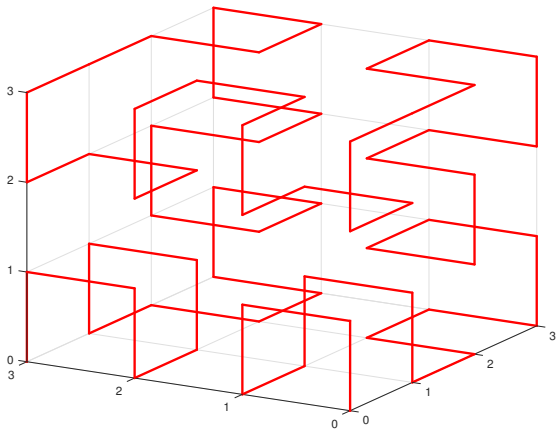


Figure 2. Three-dimensional Hilbert ordering for a $4 \times 4 \times 4$ array. The index, b, increases by 1 each time the red path passes from one location to another, starting with index 0 at $(0, 0, 0)$ and ending with index 63 at $(3, 0, 0)$.

2.3 Morton Ordering

As with Hilbert ordering, Morton ordering, $\mathcal{O}_M$, can also be viewed in terms of recursion and requires that $M = 2^m$. First, consider the 2D case of an $M \times M$ row-major array. This is re-ordered as a 2×2 array of sub-arrays each of size $M/2 \times M/2$, with each sub-array having row-major order, as shown in Fig. 3. This process is then applied recursively to each of the four sub-arrays until after $m - 1$ levels of recursion the sub-arrays are each of size 2×2. Figure 3 shows the Morton ordering obtained by applying two levels of recursive to an 8×8 array.

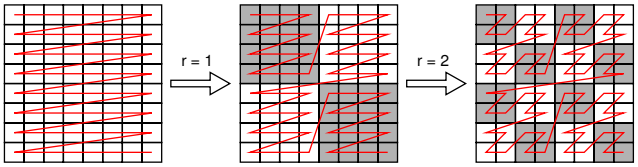


Figure 3. The lefthand part of the figure shows the original array. The middle part of the figure shows the result of Morton ordering to level $r = 1$. The righthand part of the figure shows the result of Morton ordering to level $r = 2$. Each small square represents one array item, and the continuous line between cell centres shows the order in which they are stored, starting in the top left corner. The shading highlights the division into sub-arrays.

The Morton ordering of an $M \times M \times M$ array can be defined in a similar recursive way: the array is reordered as a $2 \times 2 \times 2$ block array composed of eight $M/2 \times M/2 \times M/2$ sub-arrays, each of which has row-major order. This process is then applied recursively to each of the eight sub-arrays until after $m - 1$ levels of recursion the sub-arrays are each of size $2 \times 2 \times 2$. The Morton ordering for a $4 \times 4 \times 4$ array is shown in Fig. 4.

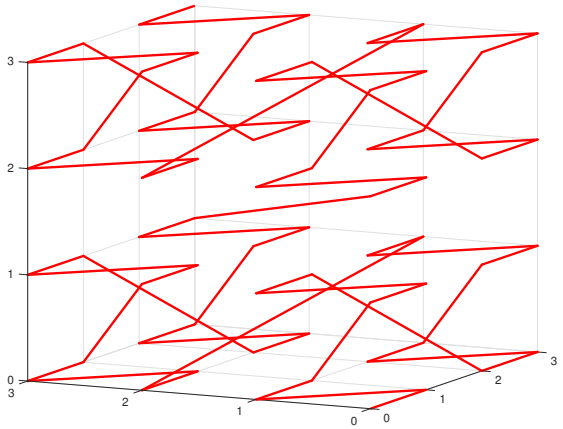


Figure 4. Three-dimensional Morton ordering for a $4 \times 4 \times 4$ array. The index, b, increases by 1 each time the red path passes from one location to another, starting with index 0 at $(0, 0, 0)$ and ending with index 63 at $(3, 3, 3)$.

Another way to represent the index, b, of Morton ordering is in terms of the bits of i, j, and k: the bits of b are obtained by interleaving the bits of i, j, and k:

$$k_{m-1}j_{m-1}i_{m-1}k_{m-2}j_{m-2}i_{m-2}\ldots k_1j_1i_1k_0j_0i_0 \tag{5}$$

2.4 Hybrid Orderings

Hybrid orderings are obtained by splitting the 3D array into sub-arrays of equal size and applying one ordering within the sub-arrays and another ordering between them. For example, a row-major ordering could be applied within each sub-array and a Hilbert or Morton ordering could be applied between them (provided the number of sub-arrays is the same power-of-two in each direction). Suppose the 3D array is of size $M \times M \times M$ and the sub-arrays are of size $T \times T \times T$,

where $M = 2^m$ and $T = 2^t$. The index, b, of the ordering consists of $3m$ bits. The lower $3t$ bits encode the ordering within each sub-array, and the upper $3(m-t)$ bits encode the ordering between sub-arrays. It should be noted that for $M = 2^m$ the row-major ordering in Eq. 2 is equivalent to concatenating the bits of i, j, and k. Thus, for a hybrid ordering applying Morton ordering between sub-arrays and a row-major ordering within them, the index b corresponding to (i, j, k) is obtained by interleaving the upper $(m-t)$ of i, j, and k, and concatenating their lower t bits.

3 Data Locality Properties

We now investigate the data locality properties of the 3D orderings defined in Sec. 2. It is assumed that processing an item at some location (i, j, k) in the array requires data from neighbouring items in the array. This dependency can be represented by a stencil, which is a list of array locations that the processing of location (i, j, k) depends on. It is assumed that the shape of the stencil is the same for all locations, so it is only necessary to store the shape of the stencil in terms of offsets from the stencil centre. For example, a simple "star" stencil consisting of an array location and the six directly adjacent locations would contain locations with the following offsets: $(0, 0, -1)$, $(0, -1, 0)$, $(-1, 0, 0)$, $(0, 0, 0))$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. For a row-major ordering this would correspond to offsets in memory of $(-M^2, -M, -1, 0, 1, M, M^2)$ for all locations. However, for Hilbert and Morton orderings the offsets in memory would depend on location.

3.1 Common Stencils

A block stencil is a common stencil consisting of a $(2g+1) \times (2g+1) \times (2g+1)$ cubical block of array locations. Another stencil of interest, particularly in molecular dynamics simulations (see Sec. 4), is the approximately spherical stencil. If the 3D array is viewed as consisting of $M \times M \times M$ spatial bins, each of unit size, then the approximately spherical template consists of all bins that are wholly of 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:

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

where $p = \left\lceil \sqrt{g^2 - i^2} \right\rceil$. 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 2, 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 2. Number of bins, and the percentage deviation from sphericity.

3.2 Data Locality Metrics

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. Some insight into the relationship between data ordering and efficient use of hierarchical memory can be gained by examining the 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. Figure 6 show the memory access patterns for a block stencil with $g = 1$ for row-major, Hilbert, and Morton orderings of a $16 \times 16 \times 16$ array. Memory offsets are accumulated over a $14 \times 14 \times 14$ array as a border of depth g bins is required by the stencil. The accumulated memory offsets for the 27 stencil bins in the row-major case all equal $14^3 = 2744$, and the correspond to the offsets shown in Fig. 5, ranging from -273 to +273. Clearly there is a greater degree of scatter in the memory access patterns for the Hilbert and Morton orderings and in both cases this extends beyond the limits of the x-axis in Fig. 6. For the Hilbert ordering the memory offsets lie between ± 3767, and 13.3% are not included in Fig. 6. The corresponding values for the Morton ordering are ± 3073 and 13.8%.

Figure 7 shows similar data to Fig. 6, but for a block stencil with $g = 3$. Here a border of depth 3 is required, so memory offsets are accumulated over a $10 \times 10 \times 10$ array. As in the $g = 1$ case the memory access patterns are more scattered for the Hilbert and Morton orderings than for the row-major ordering. For the Hilbert ordering the memory offsets lie between ± 3794, and 20.5% are not included in Fig. 7. The corresponding values for the Morton ordering are ± 3129 and 22.0%.

Another way to view the data in Figs. 6 and 7 is in terms of a histogram showing the cumulative fraction of bins within a given absolute memory offset. This is shown in Figs. 8 and 9 for $g = 1$ and $g = 3$, respectively, and $M = 16$. 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, all of the bins are

Yon slab

$-M^2+M-1$	$-M^2+M$	$-M^2+M+1$
$-M^2-1$	$-M^2$	$-M^2+1$
$-M^2-M-1$	$-M^2-M$	$-M^2-M+1$

Middle slab

M-1	M	M+1
-1	0	1
-M-1	-M	-M+1

Hither slab

M^2+M-1	M^2+M	M^2+M+1
M^2-1	M^2	M^2+1
M^2-M-1	M^2-M	M^2-M+1

Figure 5. Memory offsets for a block stencil with $g = 1$.

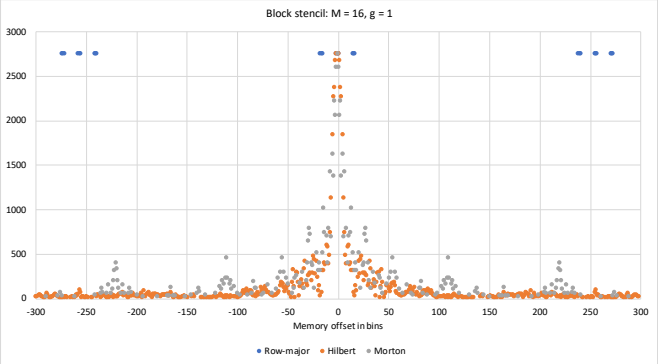


Figure 6. Accumulated memory offsets in bins for a block stencil with $g = 1$ and an array with $M = 16$. Note that the data at offsets -1, 0, and +1 for the row-major case are obscured by the Hilbert and Morton data.

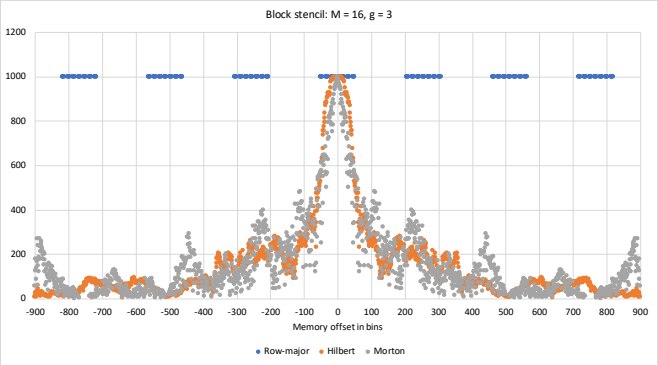


Figure 7. Accumulated memory offsets in bins for a block stencil with $g = 3$ and an array with $M = 16$.

within a memory offset of 299 for the row-major ordering, but the corresponding values for the Hilbert and Morton orderings are 0.867 and 0.862, respectively. Thus, although compared with the row-major case a higher proportion of the bins are within a small memory offset in the Hilbert and Morton cases, the reverse is true for larger memory offsets. A similar trend can be seen in Fig. 9 for the $g = 3$ case where all the bins are within a memory offset of 899 for the row-major ordering, but only 0.795 and 0.780 are within this offset for the Hilbert and Morton orderings. It is also apparent from Figs. 8 and 9 that a higher proportion of bins are within a given memory offset for the Hilbert ordering compared with the Morton ordering, up to an offset of about 899, and after that the opposite is true.

Figures 8 and 9 indicate that for sufficiently small cache sizes the largest fraction of the data needed to update the items in a bin will fit into the cache for a Hilbert ordering, followed by the Morton and row-major orderings; however, for a large enough cache this is reversed. This suggests that the performance benefits of the different orderings will depend on the sizes of the different levels in the memory hierarchy.

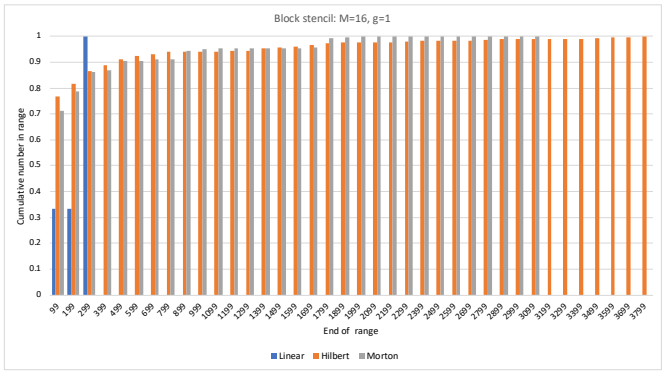


Figure 8. Cumulative fraction of bins within a given memory offset for a block stencil with $g = 1$ and an array with $M = 16$. For each set of bars the range is from 0 up to the x-axis label.

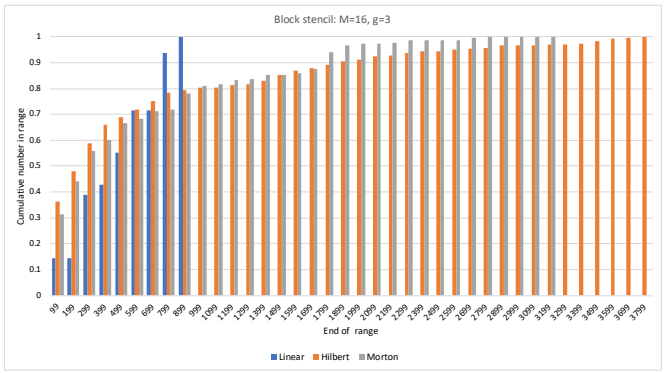


Figure 9. Cumulative fraction of bins within a given memory offset for a block stencil with $g = 3$ and an array with $M = 16$. For each set of bars the range is from 0 up to the x-axis label.

If the x-axis labels are replaced with cache sizes, then Figures 8 and 9 can also be interpreted as hit rate curves; that is, a plot of the probability of a cache hit as a function of cache size. However, a more accurate estimate of the hit rate, and the number of cache lines transferred into cache within a given time period, requires more dynamic modelling. In our simple cache model bins are stored in memory in some prescribed order (row-major, Hilbert or Morton order). The size of each cache line is b bins, 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. It is assumed that the cache can contain 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 model is given in Alg. 1, where it should be noted that only bins not in the border zone of depth g are considered.

The parameters of the cache model are the ordering, the stencil type and size, g, the size of the 3D array, M, the cache block size, b, and the number of blocks in the cache, c. A number of cache models have been run and these show some common characteristics. For example, Figs. 10 and 11 show miss rate plots for $M = 32$, a block stencil with $g = 1$,

ALGORITHM 1: 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*)) **then**
          nmisses++
          `addBlock2Cache` (*jbin*)
        end
      end
    end
  end
  return *nmisses*
end

and cache block sizes of $b = 2$ and $b = 8$ bins, respectively. In both plots in 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 is large enough to hold an additional complete row of bins. The miss rate for the Hilbert and Morton cases does not exhibit this behaviour as they are not ordered by row. Figure 10 shows that for small cache sizes the miss rate is highest for the Hilbert ordering, but for cache size between $c = 64$ and $c = 1024$ the miss rate is lowest for the Hilbert case, closely followed by the Morton case, with row-major ordering having the highest miss rate. Figure 11 also shows that the ordering with the lowest miss rate depends on the cache size. Similar behaviour is seen for approximately spherical stencils: for example, Fig. 12 shows the miss rate for $M = 32$, an approximately spherical stencil with $g = 3$, and $b = 8$. Finally, Fig. 13 shows the miss rate data for $M = 64$, a block stencil with $g = 1$, and $b = 8$. Once again it is apparent that the ordering with the lowest miss rate depends critically on the cache block size, b, and the overall cache size, c.

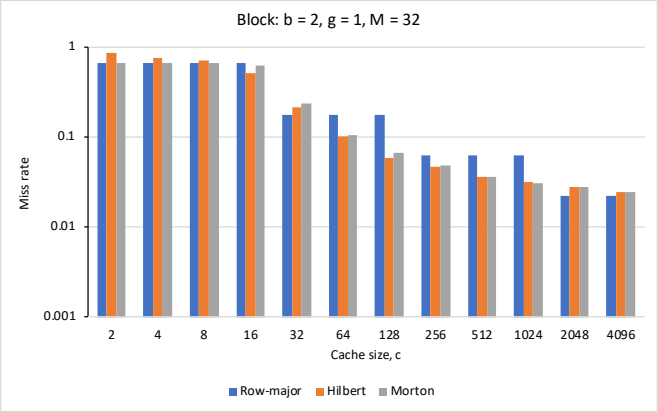


Figure 10. Miss rate as a function of cache size, c, for a block stencil with $g = 1$, an $M = 32$ array, and a cache block size of $b = 2$ bins.

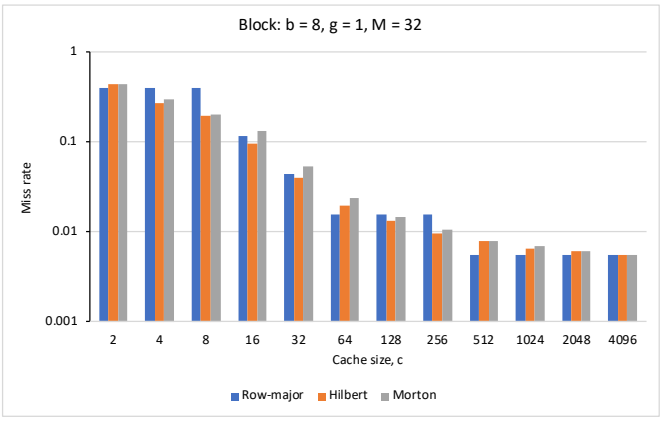


Figure 11. Miss rate as a function of cache size, c, for a block stencil with $g = 1$, an $M = 32$ array, and a cache block size of $b = 8$ bins.

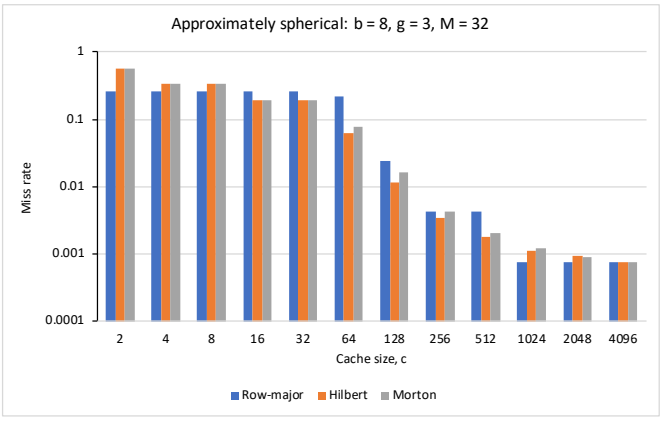


Figure 12. Miss rate as a function of cache size, c, for an approximately spherical stencil with $g = 3$, an $M = 32$ array, and a cache block size of $b = 8$ bins.

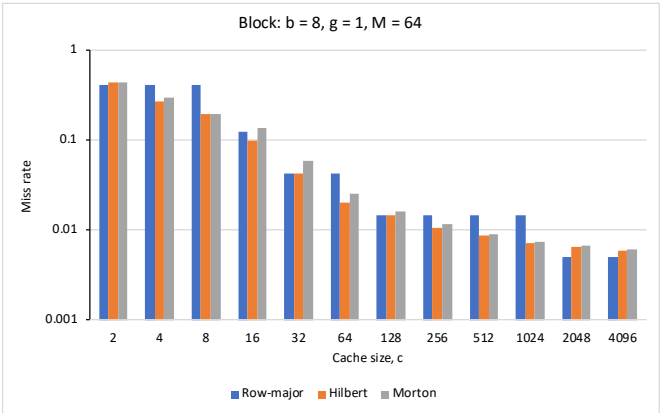


Figure 13. Miss rate as a function of cache size, c, for a block stencil with $g = 1$, an $M = 64$ array, and a cache block size of $b = 8$ bins.

4 Molecular Dynamics Simulations

A molecular dynamics simulation follows the trajectories of a set of n mutually-interacting particles through a series of discrete time steps, given the initial positions and velocities of all particles. In each time step, the force on each particle is found by summing over the forces due to all the other

particles, so the force on particle i is:

$$\mathbf{F}_i = \sum_{\substack{j=0 \\ j\neq i}}^{n-1} \mathbf{f}_{ij} \tag{7}$$

where $\mathbf{f}_{ij}$ is the force on particle i due to particle j. The force $\mathbf{F}_i$ is then applied to the particle i, thereby modifying its position and velocity. Given two particles, i and j, at positions $\mathbf{r}_i$ and $\mathbf{r}_j$, respectively, the force exerted on particle i by particle j is given by:

$$\mathbf{f}_{ij}(r_{ij}) = -\frac{\partial V}{\partial r}\bigg|_{r=r_{ij}} \hat{\mathbf{r}}_{ij} \tag{8}$$

where $V(r)$ is the interaction potential, $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$, and $\hat{\mathbf{r}}_{ij} = (\mathbf{r}_i - \mathbf{r}_j)/r_{ij}$ is a unit vector.

A number of different models have been developed to represent the potential between particles in a molecular dynamics simulation. Here we shall consider just the Lennard-Jones potential:

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{9}$$

where r is the separation between the particles, σ 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:

$$\mathbf{f}_{ij}(r_{ij}) = \frac{12\epsilon}{r_{ij}}\left[\left(\frac{r_m}{r_{ij}}\right)^{12} - \left(\frac{r_m}{r_{ij}}\right)^{6}\right]\hat{\mathbf{r}}_{ij} \tag{10}$$

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 $\mathbf{F}_j$.

4.1 Particle Binning

Finding the force on each particle by summing over all the other particles is an $O(n^2)$ algorithm. To reduce the amount of computation it is usual to impose a cutoff condition in which the force, $\mathbf{f}_{ij}$, between two particles is taken as zero if $r_{ij} \geq r_0$, where r_0 is known as the cutoff distance. This avoids having to evaluate f_{ij} using Eq. 10 when $r_{ij} \geq r_0$, but it is still necessary to find r_{ij} for every pair of particles, so the algorithm is still $O(n^2)$. To reduce the computational complexity the particles may be placed in spatial bins. Thus, the spatial domain of the problem is divided into a set of equally-sized bins, and we keep track of which particles are in each bin. To evaluate the force on a particle it is necessary to sum over only those particles that lie within the same bin and some set of nearby bins, rather than over all the particles. For example, if the bins are of size $r_0 \times r_0 \times r_0$, then to evaluate the force on particles in some bin, *ibin*, it is necessary to examine only those particles in the same bin and the 26 adjacent bins (for a 3D problem) because we know that particles in more distant bins must be more than r_0 from each of the particles in *ibin*. This type of binning corresponding to a block stencil with $g = 1$, as discussed in Sec. 3.1, and as noted in Table 2, on average only a fraction $4\pi/81$ of the interparticle distance calculations will have $r_{ij} \leq r_0$, so nearly 85% of the distance computations can be viewed as wasted, in the sense that they do not contribute to the force computation. The wastage can be decreased by making the bin size smaller (by increasing g) and/or by using an approximately spherical stencil in place of a block stencil.

In general, the particles are stored in a one-dimensional array indexed from 0 to $n-1$, and the bins are represented by a three-dimensional array of size $M \times M \times M$. For each bin it is necessary to keep track of which particles it contains. Larger values of M reduce the number of "wasted" distance computations for which $r_{ij} > r_0$, but requires more memory.

4.2 Neighbour Lists

Although decreasing the bin size improves computational efficiency, it increases the number of bins, and the amount of memory, required. Another approach is to maintain for each particle a *neighbour list* (also known as a Verlet list) of other particles that includes all those within a distance r_0. Then, when evaluating the force on a particle it is necessary to consider only those particles in its neighbour list. This requires memory nm, where m is the maximum number of neighbours that any particle has. In addition, to the extra memory needed, there is overhead in creating and maintaining the neighbour lists that adds to the execution time. To reduce this overhead the neighbour lists are rebuilt every t_b times steps, instead of at every time step. This necessitates adding a "skin" of depth r_s to the cutoff distance and building the neighbour lists to include all particles within distance $r_0 + r_s$. This will yield correct results provided no particle j can travel from a distance $r_{ij} > r_0 + r_s$ away from particle i to within a distance $r_{ij} < r_0$ in t_b (or fewer) time steps. This can be ensured by making r_s sufficiently large and t_b sufficiently small. A larger value of r_s will allow more time steps between rebuilding the neighbour lists, thereby reducing overhead, but will require more memory. A naive approach to building the neighbour lists will result in an $O(n^2)$ algorithm, however, this can be avoided by using a spatial binning procedure similar to that described above. Thus, the neighbour list for each particle is built by considering only those particles in bins within a stencil centred on the bin containing that particle. If the bins are cubes of size r_b, then a stencil with $g = \lceil (r_0 + r_s)/r_b \rceil$ is an appropriate choice.

In Sec. 4 it was pointed out that Newton's Third Law can be used to reduce the operation count in evaluating the force on the particles. If this is not done we have the *full-neighbour list* case, and particles in all all stencil bins are considered as potential neighbours. If Newton's Third Law is exploited we have the *half-neighbour list* case, and care must be taken to avoid counting any interaction between particles twice. In the half-neighbour list case, particles in the following stencil bins are checked for inclusion in the neighbour list:

1. All bins with $iz > 0$
2. If $iz = 0$, all bins with $iy > 0$, or $iy = 0$ and $ix > 0$.

where $-g \leq ix, iy, iz \leq g$ are indices to bin locations in the stencil. Thus, the number of stencil bins examined in the half

neighbour list case is:

$$1 + 3g + 6g^2 + 4g^3 \tag{11}$$

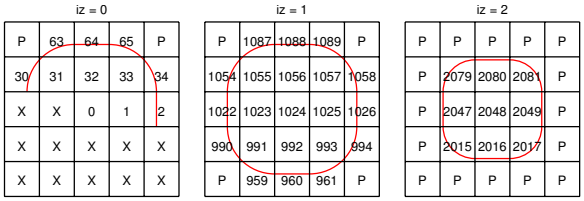


Figure 14. The numbered bins show the approximately spherical stencil for a $32 \times 32 \times 32$ array of bins with $r_0 + r_s = 2.8$ and $r_b = 2.067$, so that $g = 2$. The value iz labels the z plane. Half-neighbour lists are used so bins in the $iz = -2$ and $iz = -1$ planes are excluded by the half-neighbour list algorithm, as also are the bins marked X in the $iz = 0$ plane. Bins marked P are bins that would be included in a block stencil but which are excluded by the distance constraint in the approximately spherical stencil. The distance constraint is shown in red for each plane.

4.3 Periodicity and Ghost Particles

Periodicity of the spatial domain can be applied to a molecular dynamics simulation by surrounding the array of bins that cover the spatial domain by a layer g bins wide in each direction. These extra bins contain copies of the particles that lie within g bins of the edge of the domain, as shown in Fig. 15 for a slab of bins in the z plane for $g = 2$. These particle copies are usually called *ghost* particles. Since the force on a particle does not depend on velocity, only the position data needs to be stored for a ghost particle, and periodicity requires that the domain size needs to be added to, or subtracted from, one or more of the x, y, and z coordinates. For example, consider the three bins labelled $\mathrm{C_E}$ in Fig. 15. The ghost particles in the lower-left one of these have the same position coordinates as the corresponding particles in bin E, expect that the domain size is subtracted from their x coordinate. This ensures the correct answer is obtained when evaluating the distance between ghost particles and "real" particles in unshaded bins. Similarly, for the upper-right bin labelled $\mathrm{C_E}$, the domain size must be added to the y coordinate, and for the upper-left $\mathrm{C_E}$ bin the domain size must be added to the y coordinate and subtracted from the x coordinate. Once the ghost particles in the extra bins are in place, the force on the particles in the unshaded bins can be evaluated by processing the bins enumerated by the stencil.

4.4 Sorting Particles

If particles are processed in a loop in random order, then data locality is expected to be poor. If, however, particles are processed in the order that they are created at the start of the simulation, data locality may be better, particularly if the particles initially have a crystalline structure. However, as the simulation progresses and particles move, data locality will degrade as the ordering becomes less spatially coherent. An alternative approach is to process particles by bins by means of an outer loop over bins and an inner loop over the particles in a particular bin. This is likely to improve data locality since the neighbours lists of particles in the same bin

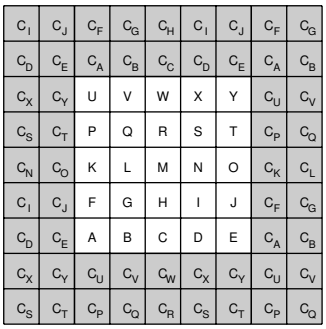


Figure 15. The domain of the problem is covered by a 5×5 array of bins, shown unshaded and labelled A to Y. The grey bins are periodic copies of bins within two bins of the edge of the domain. Thus, bin $\mathrm{C_A}$ is a copy of bin A, and similarly for the other grey bins.

will overlap. However, data locality can be directly improved by sorting the particles so that:

1. Particles in the same bin are nearby in memory. This requires that the particles be re-indexed so that particles in the same bin are consecutively indexed.
2. Particles are also sorted according to the bin ordering so that the particles for one bin are followed in memory by the particles in the next bin in the ordering, and so on.

Particle sorting affects the order of computation, and is expected to improve data locality both when the outer loop is over particles and when it is over bins. Particles are sorted every t_s times steps, where t_s is chosen to balance the gain from improving data locality against the overhead of doing the sorting.

5 GPU Implementation of a Molecular Dynamics Simulation

Graphical processing units (GPUs) are mainly designed for render-intensive graphical applications and computer games. However, the massive computing power of GPUs, coupled with their low cost, large memory, and low electrical power requirements, has led to the implementation of scientific applications on GPUs. The continuous advances in GPU technologies, such as multi-GPU clustering, connectivity to InfiniBand networks to support hybrid CPU/GPU implementations, and host memory mapping, have resulted in GPUs and other accelerators being widely adopted in high-performance computing. GPUs have different hierarchical levels of memory, varying from model to model, with different bandwidths and capacities. Frequent access to global memory results in a significant impact on application performance, so there are substantial benefits in reusing data stored in the higher levels of the memory hierarchy. Global memory is the lowest level of memory on a GPU, and accessing it incurs hundreds of clock cycles of latency, which may be hidden by dynamically scheduling other runnable threads while the original thread waits for its memory operations to complete. If data are not carefully ordered in memory, applications may fetch data from global memory into a higher level cache, but leave a number of related data items in the global memory, which then requires another request to fetch into cache. This type of inefficient use of

hierarchical memory increases the execution time. Other additional factors, such as cache sizes, synchronization, warp size, and thread divergence, also impact execution time.

The data locality properties of row-major, Hilbert, and Morton orderings have been investigated for a GPU implementation of the miniMD molecular dynamics simulation. The miniMD package [29] is a simplified version of the well-known LAMMPS simulation package [†], with both packages sharing the same solution methods. In particular, both use a combination of neighbour lists and spatial binning as discussed in Sec. 4, and support periodicity through the use of ghost particles. Full and half neighbour list algorithms may be used to determine the force exerted on each particle due to the other particles. miniMD supports the embedded atom model and Lennard-Jones potentials. In this work, data locality issues are studied for the Lennard-Jones case.

In each time step the computation of the force on the particles is the most computationally intensive phase. Therefore, we focus on the performance of the force computation, where the force on each particle is computed by a different thread on the GPU. An alternative approach would be to have each thread process a single bin so that all the particles in that bin would be processed by a single thread. This would result in less parallelism, and since bins may contain differing numbers of particles load imbalance would also be an issue.

When computing the force on particle i, force $\mathbf{f}_{ij}$ exerted on i by each particle, j, in its neighbour list that lies within the cutoff distance r_0 is found and added to the total force on particle i. In the half neighbour list algorithm $-\mathbf{f}_{ij}$ is also added to the total force on particle j. This reduces the number of computations but, since multiple threads may be concurrently updating the total force on a particle, a naive implementation results in a non-deterministic program. This problem does not arise in the full neighbour list case as each thread updates the force only for the particle it is responsible for. Thus, in the half neighbour list case the thread handling particle i must update the total force for particle j atomically.

The force computation in miniMD also involves the computation of macroscopic quantities, namely, the virial coefficient and the total potential energy. These computations require reduction operations in the loop over particles. Performing a reduction operation efficiently on a GPU is complicated by the fact that threads cannot be synchronized across all thread blocks. Typically partial sums are computed for each thread block using a tree-based algorithm or atomic addition. These partial sums must then be added together, which can be done either on the host or by invoking another kernel on the GPU or by having just one thread do the addition. The summations needed to find the viral coefficient and potential energy do not take much time compared with the computation of the inter-particle forces, but care must be taken to ensure they are done correctly.

6 GPU Performance Results and Analysis

The work presented in this paper was conducted on an NVidia GeForce GTX 960 with compute capability 5.2, eight streaming multiprocessors (SMP) and 128 cores per SMP. The maximum number of threads per block is 1024. The unified L1/texture cache is of size 48KB, and the L2 cache is 1MB. 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, at single precision 1365 particles would fit into the L1 cache, and 29127 particles would fit into the L2 cache. Suppose, for example, that there are a maximum 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 fit within a cache line.

The host computer for the GPU contains an 8-core Intel Core i7-5960X processor with a 20MB cache.

In Lennard-Jones 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 cutoff distance was $r_0 = 2.5$, and the skin thickness was $r_s = 0.3$.

6.1 Profiling

Timing experiments were first carried out to demonstrate that the force computation dominates the run time, as may be seen in Figs. 16 and 17. In all cases, a row-major ordering was used, and the GPU timings for the force computation include the time to transfer particle data between the host and the GPU in each time step. For the CPU timings in Fig. 16 the ratio of force computation time to total simulation time is consistently about 80% for both the half and full neighbour list cases. For the GPU timings in Fig. 17 the ratio decreases as the problem size increases. This is because the neighbour lists are managed on the host and the processing time for this increases more rapidly than for the force computation on the GPU. On the CPU the force computation time for the half neighbour list case is about 60% of that for the full neighbour list case. For the GPU the difference is much less, because the gain from performing fewer operations in the half neighbour list case is offset by the need to accumulate forces atomically.

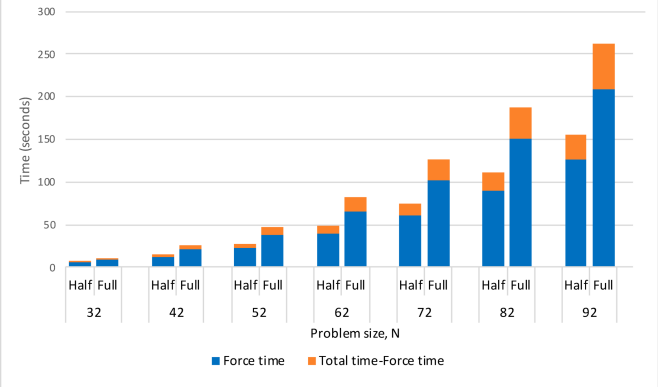


Figure 16. Total simulation time and force computation time for the CPU implementation, accumulated over the first 100 time steps, for different problem sizes, N. Times are shown for the half and full neighbour list algorithms. Note that the total height of each column is the total time for the simulation.

Ratios of the times for the force computation on the CPU and the GPU are shown in Table 3. The ratio tends to increase

[†] https:/lammps.sandia.gov

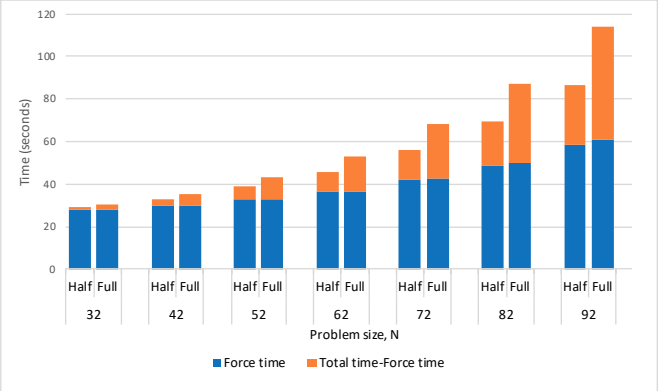


Figure 17. Total simulation time and force computation time for the GPU implementation with 1024 threads per block, accumulated over the first 100 time steps, for different problem sizes, N. Times are shown for the half and full neighbour list algorithms. Note that the total height of each column is the total time for the simulation.

slowly with problem size, N. The lower values for the half neighbour list case are due to the atomic additions performed in this case.

N	Row-major		Hilbert		Morton	
	Full	Half	Full	Half	Full	Half
32	11.67	6.89	8.07	5.97	8.66	6.38
42	12.27	7.38	7.32	6.58	7.10	6.76
52	12.29	7.75	7.65	6.84	7.48	7.01
62	12.14	7.84	7.94	7.20	7.69	7.08
72	12.66	7.87	8.02	7.07	7.84	7.56
82	12.32	8.01	7.96	7.14	7.56	7.22
92	12.55	8.12	7.95	7.23	7.85	7.26

Table 3. Ratio of CPU to GPU time for the force computation. In the GPU computations 1024 threads per block were used.

6.2 Locality Analysis

Timing experiments have been carried out to compare the impact on performance of the force computation on the GPU of the row-major, Hilbert, and Morton orderings. These timings are for the execution of the kernel code on the GPU and do not include the time to transfer particle data between the host and the GPU.

Figures 19 and 20 show the time for the force computation kernel per time step for differing thread block sizes, averaged over the first 100 time steps, for the full and half neighbour list cases, respectively. These figures show that the half neighbour list algorithm results in faster execution than the full neighbour list algorithm, particularly at larger problem sizes. The timings for the full neighbour list case in Fig. 19 show that for all problem sizes and thread block sizes the row-major ordering results in the fastest execution, with the effect being more pronounced for larger problem sizes. For all problem sizes, a thread block size of 1024 is fastest and 640 is slowest. A smaller block size increases the number of registers available per thread, which would tend to improve performance. However, it also decreases the number of warps per block, which may lead to inefficient GPU utilization. The results for the half neighbour list case in Fig. 20 also show that a row major ordering results in the fastest execution, although not by as large an amount as in the full neighbour list case. Also, it was found that in this case a thread block size of 576 was fastest, with 640 still being the slowest.

Figure 18 shows the relationship between thread grid configuration, theoretical occupancy, and total execution time for the full and half neighbour list cases. The problem size is $N = 92$, although similar results were obtained for other problem sizes. The theoretical occupancy is the maximum number of warps that can execute on a streaming multiprocesor of a GPU divided by the device limit, and is affected by factors such as the number of threads per block, the number of registers in use, and the capabilities of the GPU. The half neighbour list case has an optimal theoretical occupancy of 0.563, which is slightly lower than for the full neighbour list case of 0.625.

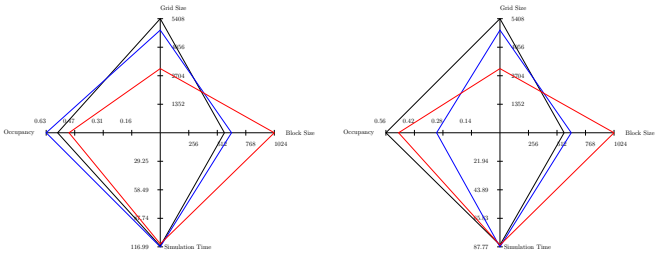


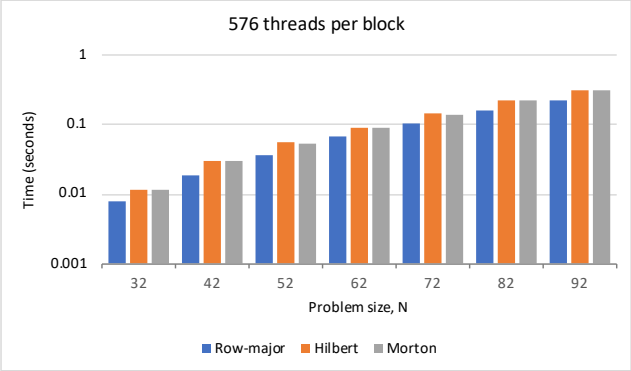
Figure 18. Dependency of total execution time and theoretical occupancy for different configurations of the thread grid for the full (left) and half (right) neighbour list cases. The problem size is $N = 92$.

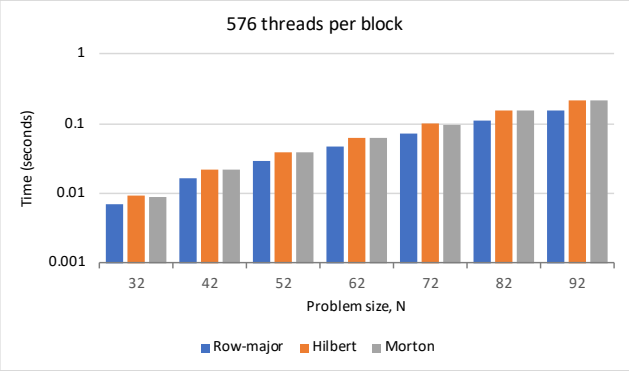
To interpret the timings in Figs. 19 and 20 the *nvprof* profiler has been used to collect data on execution of the force computation kernel to gain insights into how efficiently the GPU is being used. Profiling was done for a problem size of $N = 92$ (3114752 particles), with 1024 threads per block, for both the full and half neighbour list cases. The number of cycles and the number of eligible warps are shown in Table 4. 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.

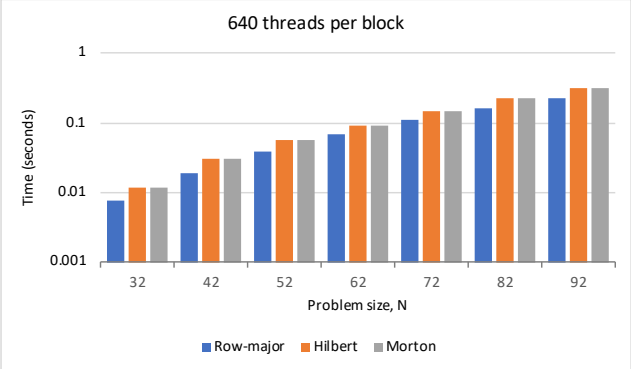
		Number of Cycles	Eligible Warps (%)
Row-major	Full	11.80×10^9	7.5
	Half	10.70×10^9	10.3
Hilbert	Full	12.00×10^9	7.3
	Half	10.74×10^9	10.3
Morton	Full	12.02×10^9	7.4
	Half	10.68×10^9	10.3

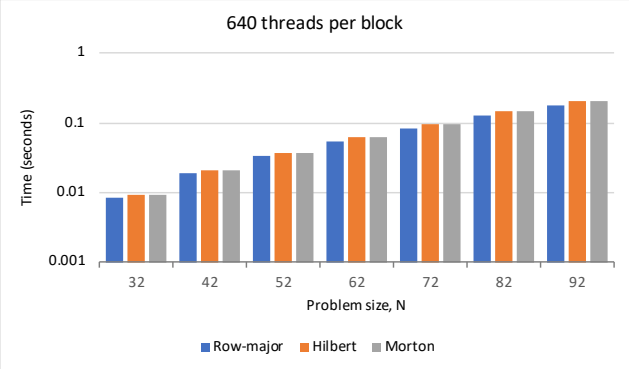
Table 4. Number of active cycles and percentage of active warps that are eligible per active cycle, for $N = 92$ and 1024 threads per block.

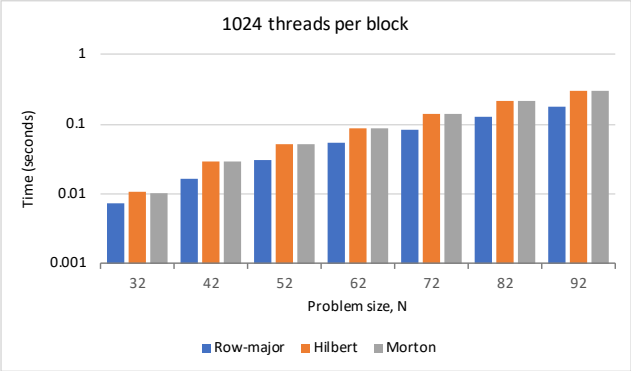
Table 4 shows that the number of cycles to execute the force computation kernel is approximately 10-12% larger in the full neighbour list case, compared with the half neighbour list case. This is because the number of computations is higher in the full neighbour list case. In addition, the percentage of active warps that are eligible is only about 7.5% for the row-major and Morton orderings in the full neighbour list case, but is larger in the half neighbour list case: 10.3%. However, for Hilbert and Morton orderings the percentage of eligible warps is less than for row-major order











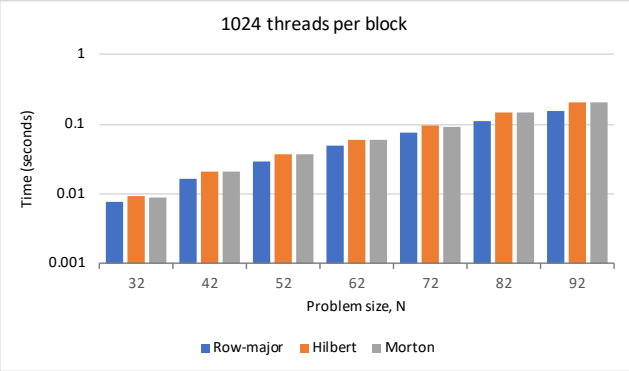


Figure 19. Full neighbour list algorithm: Time for execution of force computation kernel on the GPU for one time step.

Figure 20. Half neighbour list algorithm: Time for execution of force computation kernel on the GPU for one time step.

by 0.2 and 0.1, respectively, in the full neighbour list case and they are the same for the half neighbour list case.

Further information on the cause of the stalled warps in the force computation kernel is presented in Figs. 21 and 22. These figures show that warp stalls are mainly dues to memory dependency and execution dependency. Memory dependency stalls occur when a warp must wait for a previous memory operation. In the full neighbour list case, memory dependencies are the main reason for warp stalls, accounting for 84-90% of all stalls, compared with a more uniform value of 90% for the half neighbour list case. An execution dependency stall occurs when an input required by an instruction is not yet available. In the full neighbour list case, execution dependencies account for 12 % and 14% of stalls for Morton and row-major orderings, respectively, but for only 9% of stalls for Hilbert ordering. In the half neighbour list case execution dependencies consistently account for 9% of stalls. It should be noted that there are no synchronization stalls as the kernel code contains no explicit synchronization.

Warp execution efficiency is the ratio of the average number of active threads per warp to the maximum number of threads per warp supported on a multiprocessor. Warp execution efficiency is affected by intra-warp divergence, which occurs when threads in a warp execute different control paths through a kernel, and by non-coalesced memory accesses. 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 are shown in Fig. 23, for problem size $N = 92$ and 1024 threads per block. Figure 23 shows that the warp execution efficiency is about 57% and 83%for the half and full neighbour list cases, respectively, for all orderings. 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.

A memory transaction is the movement of data between two areas of memory. When accessing data it is more efficient to do so with a smaller number of transactions.

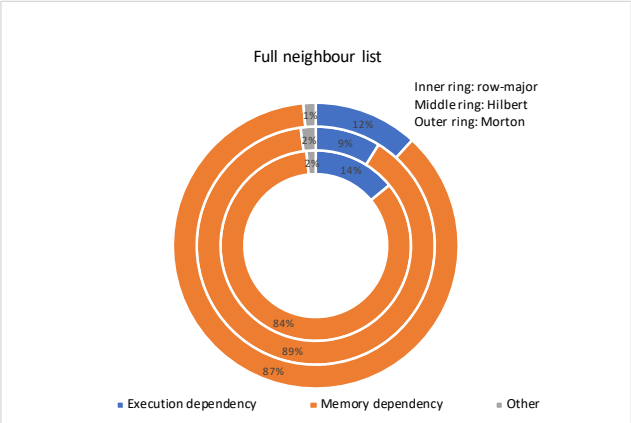


Figure 21. Full neighbour list case: Output from *nvprof* giving the cause of warp stalling in the force computation kernel.

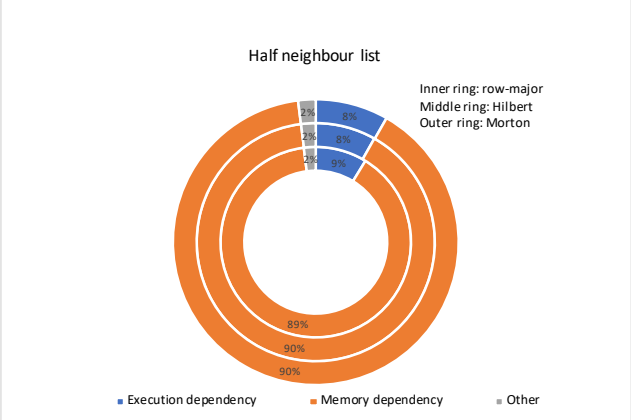


Figure 22. Half neighbour list case: Output from *nvprof* giving the cause of warp stalling in the force computation kernel.

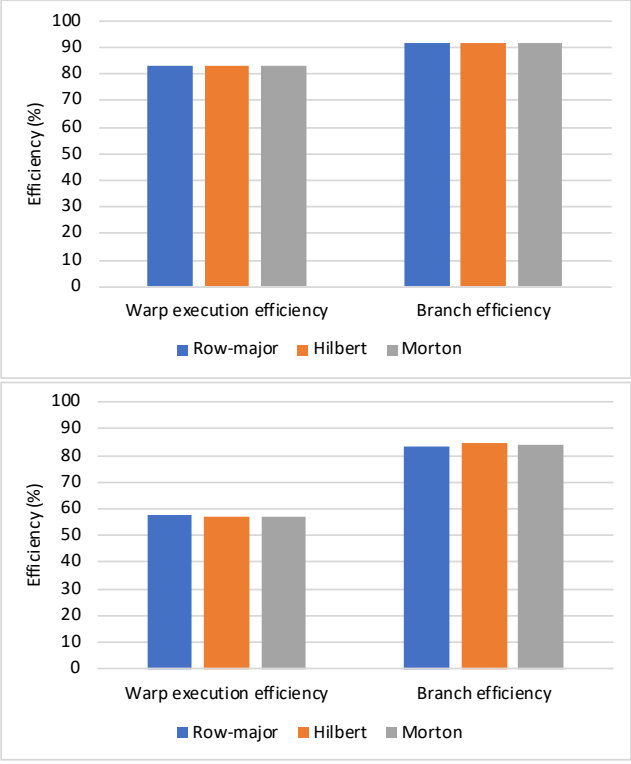


Figure 23. Warp execution and branch efficiency for the full (upper plot) and half (lower plot) neighbour list algorithms.

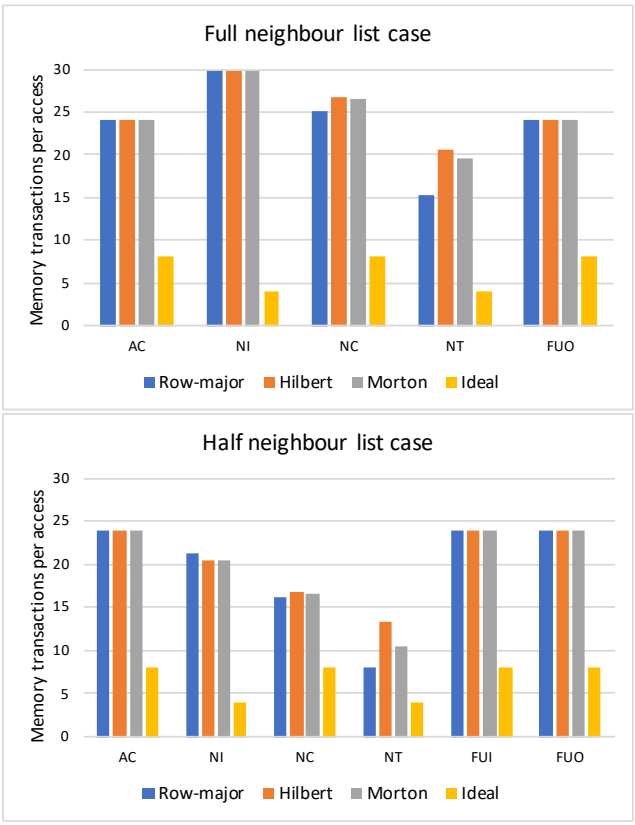


Figure 24. Number of memory transactions per access for the full (upper plot) and half (lower plot) neighbour list algorithms.

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 number of threads in a warp is 32, the data item size is 8 bytes, and the cache line size is 32 bytes, then ideally 8 memory transactions are needed. Figure 24 shows 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 neighbour list.
NT: Access the 4-byte type of a particle.
FUI: Access an 8-byte force component for a particle in neighbour list (half neighbour list case only).
FUO: Access an 8-byte force component for a thread's particle.

Figure 24 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. Particles data is 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, so that the x values of successive particles are separated by 24 bytes and it is expected that accessing a single x value

for all the threads in a warp should require 24 memory transactions (and similarly for the y and z values). This is the case for the AC, FUI, and FUO accesses in Fig. 24, but for the NC accesses the value is slightly larger than 24 for the full neighbour list case, and less for the half neighbour list case. Larger values occur when not all the accesses for a warp are in the same block of 32 bytes, and smaller values occur when the neighbour lists for successive particles have particles in common. Also the times for the NI, NC, and NT accesses are smaller for the half neighbour list case, possibly because fewer particles are processed in this case which means particle data stays in L2 cache for longer.

In the Maxwell architecture the L1 and texture caches are combined in a single unit referred to as "unified cache", the size of which is 48KB in this work, with a cache line size of 128 bytes. Since there are 32 threads per warp there are 1536 bytes for each thread. The position and force components for a particle corresponds to six 8-byte values for a total of 192 bytes, so ideally 6 particles per thread will fit into unified cache. However, when data is accessed for neighbour list particles this displaces data already in the unified cache, reducing the hit rate. Note that the cache line size for L2 cache is 32 bytes. Figure 25 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 GPU 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.

From Fig. 25 it can be seen that the hit rate for a row-major ordering is slightly larger than for a Hilbert or Morton ordering. The half neighbour list case has fewer misses to unified cache, but more misses when writing to L2 cache.

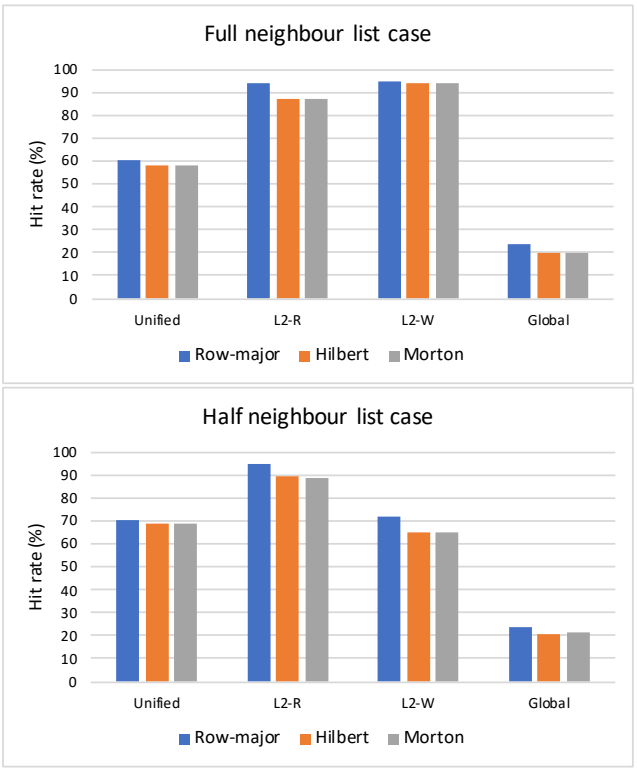


Figure 25. Cache hit rates for the full (upper plot) and half (lower plot) neighbour list algorithms.

7 Related Work

7.1 Space Filling Curves

In the late 19^{th} century Peano proposed a continuous mapping of points within the unit interval $[0, 1]$ onto a subset of a unit square. This can be viewed as projecting coordinate points arranged in a $2^n \times 2^n$ grid onto one dimension. More recently, Sagan[3,33] has presented an extensive study of space filling curves and their applications, and there is an extensive literature on generating multi-dimensional space filling curves. For example, 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[4,21,41]. 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 multidimensional point onto the Hilbert value, as in[12,23].
- Using only computation, which generates the Hilbert curve without using any predefined values, as in[5,6]. This approach enables the mapping of any arbitrary 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.

A comprehensive study of 3-dimensional Hilbert curves has been conducted by Haverkort[14,15,32]. Haverkort's work[15] 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. Lawder[21] has also investigated the encoding and decoding of multi-dimensional Hilbert curves. This work has proposed an inverse mapping from n-dimensions, and enhanced the encoding procedures suggested by Butz[5,6]. Chen[7] proposed mappings based on a replication process of the Hilbert square matrix rather than the bit-processing technique proposed by Fisher[12]. Liu and Schark[12] have implemented rotation matrices and vector functions in their three-dimensional Hilbert encoding/decoding approach. Feng et al.[11] investigated the GPU performance of two-dimensional Hilbert curve generation algorithms that are based on a block matrix iteration method and a state diagram method.

The Hilbert curve has been adopted in various applications as an approach to compress multiple keys, weights, or data in order to search or store the data. For instance, in database management where multi-dimensional points have to be

sorted into a one-dimensional structure[10]. In addition, the Hilbert curve has been applied widely in other application areas as described by Zhang[41], where it is used to perform a fast scan of an arbitrarily-sized cuboid region.

The literature on Morton orderings has been reviewed by Sagan[33]. The Morton ordering is simpler than the Hilbert curve, and its path changes from one dimension to another every two steps along the path. Thus, the nodes at the end of a linear segment and the first node at its subsequent segment are not adjacent. Morton encoding and decoding algorithms have been studied by Raman and Wise[30] and have been used to partition matrices into blocks for optimal memory hierarchy utilization. Morton order has been adopted for constructing linear quad-trees and to restructure matrices into one-dimensional arrays[18,35,39]. Most of the literature that uses Morton orderings is for partitioning matrices into blocks or tiles for mathematical applications.

Morton orderings to improve data locality in tiled matrix algorithms has been investigated by Evangelia and Nectarios[2] by restructuring the memory layout of multidimensional arrays using binary mask operations. In this study, data are divided into blocks that can be processed independently of other blocks; this is not the case in molecular dynamics simulations where evaluating the force on a particle requires data from other cells. A number of studies have been conducted that use space-filling curves for data tiling, especially with multidimensional matrices[2,34,38], to map the multidimensional iteration indices to a linear index space.

Performance evaluation of canonical (row-major and column-major) and Morton layouts on various CPU platforms has shown that Morton ordering gives consistent performance and its mapping approach makes it a competitive memory layout[37].

7.2 Data Ordering in Molecular Dynamics Simulations

Different molecular dynamics simulations are designed to address specific objectives according to the computational accuracy, simulation time, and inter-particle forces used. Therefore, GPUs are increasingly used to reduce execution time due to their hardware featues, for which molecular dynamics simulations may be optimized. Hou et al.[16] and Juekuan et al.[40] have investigated molecular dynamics simulations with Lennard-Jones and Tersoff potentials, respectively, on GPUs. However, they consider only the solid phase in which the neighbour lists are fixed, which avoids the need to periodically re-create them.

The work of Meloni et al.[26] seeks to improve locality of reference by re-ordering particles to transform the sparse interaction matrix to a banded matrix. Element (i, j) of the interaction matrix is 1 is particle i interacts with particle j, and is 0 otherwise. This re-ordering may be done with the Reverse Cuthill-McKee (RCM) algorithm, however, an improved algorithm is based on the linked-cell approach in which 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[24] have 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[13] addresses the large memory requirements of storing a neighbour list for each particle by means of *pseudo-Verlet lists*. 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 are then used to determine if two particles interact, rather than using traditional neighbour lists that stores 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.

Anderson, Lorenz, and Travesset[1] were among the first researchers to fully implement molecular dynamics simulations on a GPU using CUDA. They present performance results 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. More recently, Tang and Karniadakis[36] 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 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.

The GPU performance of molecular dynamics simulations with Tersoff, embedded-atom model and Lennard-Jones potentials has been compared by Minkin et al.[27]. The 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 considered here.

The use of Hilbert and Morton orderings to enhance data locality for molecular dynamics and other irregular applications has been investigated by Mellor-Crummey, Whalley and Kennedy[25]. 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.[20] 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 re-ordering, which depends on temperature.

An alternative to the stencil-based approach to building neighbour lists has been proposed by Howard et al.[17]. 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 larger boxes, and so on, 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 find that the latter is significantly faster for colloidal systems characterised by large size disparities.

Wu, Zhang and Shen[19] have proposed the use of asynchronous data transformations to speed up 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 overhead associated with data ordering. Wu, Zhang and Shen have 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.

7.3 Efficient Access to Hierarchical Memory on GPUs

The memory hierarchy of different GPUs, and its impact on an application's performance, varies from one vendor/architecture to another. However, they have a number of optimization principles in common, which may be summarised as follows[8,31]:

- Optimize high level memory utilization: this can be achieved by data ordering, coalesced memory access, and the order of computation of operations. However, the limitations of GPU memory size, the fine granularity of running threads and available resources, and the level of data sharing between threads, are obstacles to achieving data reuse in the upper levels of memory, which are embedded on-chip.
- Coalesced access to relevant data significantly enhances computation performance due to the GPU's capability to apply the same instruction to multiple data. However, this is not possible for algorithms that use divergent control flow to access different data or branches of the algorithm. In addition, coordinating the memory accesses of different threads leads to better performance.
- Synchronization limits parallel execution speed.
- Increasing the number of threads allows the latency of global memory accesses to be hidden. However, this might increase contention for scarce resources such as with local variables and registers. Theoretical occupancy can help to indicate the best configuration for the grid of threads, but it is not guaranteed that this will provide optimal performance.

8 Conclusions and Future Work

The use of row-major, Hilbert, and Morton orderings in storing data and sequencing computations in molecular dynamics simulations has been investigated as a mechanism for efficiently accessing hierarchical memory on a GPU. At each time step each particle interacts with particles stored in its neighbour list, which is periodically updated. The simple cache model in Sec. 3.2 shows that Hilbert and Morton orderings appear to have better spatial data locality than a row-major ordering as stencil bins are more clustered in memory about the stencil centre. However, Hilbert and Morton orderings have a long "memory tail" in the sense that 10-20% of stencil bins are further away in memory than any of the stencil bins in the row-major case. This suggests that the performance benefits of the different orderings will depend on the sizes of the different levels in the memory hierarchy and the cache line sizes. This is also apparent from the miss rates determined from the cache model.

The force computation dominates the execution time of molecular dynamics simulations, and observed timings for the force computation on the GPU for a range of problem sizes are consistent with the results of the cache model: in general, a row-major ordering results in slightly faster execution than Hilbert and Morton orderings (see Sec. 6. This can be accounted for by the slightly higher L2 cache hit rates for row-major orderings, as shown in Fig. 25.

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 data locality properties of the different data orderings.

Future work will extend the timing experiments and analysis to other NVidia GPUs and a broader range of molecular dynamics simulations. In addition, we shall investigate the impact of by-passing the unified cache so that only the L2 cache is used.

Acknowledgements

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

1. J. A. Anderson, C. 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.
2. 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*, pages 107–119, February 2004. doi: 10.1109/INTERA.2004.1299515.
3. M. Bader. *Space-Filling Curves: An Introduction With Applications in Scientific Computing*. Texts in Computational Science and Engineering. Springer, Berlin, Heidelberg, 2012. ISBN 9783642310454.
4. 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.
5. A. 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.
6. A. R. Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Trans. Comput.*, 20(4):424–426, April 1971. ISSN 0018-9340. doi: 10.1109/T-C.1971.223258.
7. N. Chen, N. Wang, and B. 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.
8. K. Choo, W. Panlener, and B. Jang. Understanding and optimizing GPU cache memory performance for compute workloads. In *IEEE 13th International Symposium on Parallel and Distributed Computing*, pages 189–196, June 2014. doi: 10.1109/ISPDC.2014.29.
9. J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990. ISSN 0098-3500. doi: 10.1145/77626.79170. URL http://doi.acm.org/10.1145/77626.79170.
10. C. Faloutsos and S. 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.
11. C. Feng, S. Shu, J. Wang, and Z. Li. The parallel generation of 2-D Hilbert space-filling curve on GPU. In *Fifth International Conference on Biomedical Engineering and Informatics*, pages 1359–1362. IEEE, 2012.
12. 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.
13. P. Gonnet. Pseudo-verlet lists: a new, compact neighbour list representation. *Molecular Simulation*, 39(9):721–727, 2013.
14. H. Haverkort. An inventory of three-dimensional Hilbert space-filling curves. *Computing Research Repository*, abs/1109.2323, 2011. URL http://arxiv.org/abs/1109.2323.
15. H. Haverkort. How many three-dimensional Hilbert curves are there? *Journal of Computational Geometry*, 8(1):206–281, 2017.
16. C. Hou, J. Xu, P. Wang, W. Huang, and X. Wang. Efficient GPU-accelerated molecular dynamics simulation of solid covalent crystals. *Computer Physics Communications*, 184(5):1364–1371, 2013. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2013.01.001. You may need to understand the used algoritm.
17. M. P. Howard, J. A. Anderson, A. Nikoubashman, S. C. Glotzer, and A. Z. Panagiotopoulos. Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units. *Computer Physics Communications*, 203:45–52, 2016.
18. C.-Y. Huang and Y.-W. Chen. Linear quadtree construction in real time. *J. Inf. Sci. Eng.*, 26:1917–1930, 2010.
19. P. Jiang, L. Chen, and G. 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.
20. M. Kunaseth, K. Nomura, H. Dursun, R.K. Kalia, A. Nakano, and P. Vashishta. Memory-access optimization of parallel molecular dynamics simulation via dynamic data reordering. In *Lecture Notes in Computer Science, vol 7484*, pages 781–792. Springer-Verlag, Berlin, Heidelberg, 2012.
21. 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, August 2000. Technical Report no. JL1/00.
22. A. Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18:280–315, 1968.
23. X. Liu and G. F. Schrack. An algorithm for encoding and decoding the 3-D Hilbert order. *IEEE Transactions on Image Processing*, 6(9):1333–1337, 1997.
24. J. Luo and L. 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.
25. J. Mellor-Crummey, D. Whalley, and K. 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.
26. S. Meloni, M. Rosati, and L. Colombo. Efficient particle labeling in atomistic simulations. *Journal of Chemical Physics*, 126(12):121102, 2007. doi: 10.1063/1.2719690.
27. A. S. Minkin, A. B. Teslyuk, A. A. Knizhnik, and B. V. Potapkin. GP-GPU performance evaluation of some basic molecular dynamics algorithms. In *International Conference on High Performance Computing Simulation (HPCS)*, pages 629–634, July 2015. doi: 10.1109/HPCSim.2015.7237104.
28. Prusinkiewicz P. and Lindenmayer A. *The Algorithmic Beauty of Plants*. Springer Verlag, New York, 2nd. edition, 1996.
29. S. Plimpton and P. Crozier. Mantevo Project: MiniMD version 1.2, 2016. URL http://mantevo.org/download/.
30. R. Raman and D. S. Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573,

2008.

31. S. Ryoo, C. I. Rodrigues, S. Baghsorkhi, S. Stone, D. B. Kirk, and W.-M. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345220.
32. H. Sagan. A three-dimensional Hilbert curve. *International Journal of Mathematical Education in Science and Technology*, 24(4):541–545, 1993. doi: 10.1080/0020739930240405.
33. H. Sagan. *Space-filling curves*. Universitext Series. Springer-Verlag, 1994. ISBN 9780387942650.
34. F. K. A. Salem and Al-Arab M. Comparative study of space filling curves for cache oblivious TU decomposition. *Computing Research Repository*, abs/1612.06069, 2016.
35. L. Stocco and G. F. Schrack. Integer dilation and contraction for quadtrees and octrees. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 426–428. IEEE, 1995.
36. Y.-H. Tang and G. E. Karniadakis. Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications. *Computer Physics Communications*, 185: 2809–2822, 2015.
37. J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience*, 18(11):1509–1539, 2006. ISSN 1532-0634. doi: 10.1002/cpe.1018.
38. D. 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, 2018. doi: 10.1177/1094342017725568.
39. D. S. Wise. Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In *Proceedings of Euro-Par 2000*, pages 774–784. Springer, 2000.
40. J. Yang, Y. Wang, and Y. Chen. GPU accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2):799–804, 2007. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2006.06.039. this is similiar to your work ... see how cite it.
41. J. Zhang and S.-I. 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.