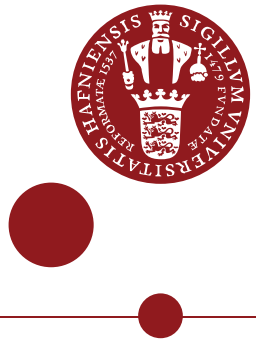
UNIVERSITY OF COPENHAGEN



BSc in Computer Science

Caching Strategies for Image Processing

A comparison of standard image processing algorithms and caching methods

Søren Hougaard Mulvad

Supervised by Jon Sporring

External examiner: Rasmus Reinhold Paulsen

June 8, 2020

Søren Hougaard Mulvad

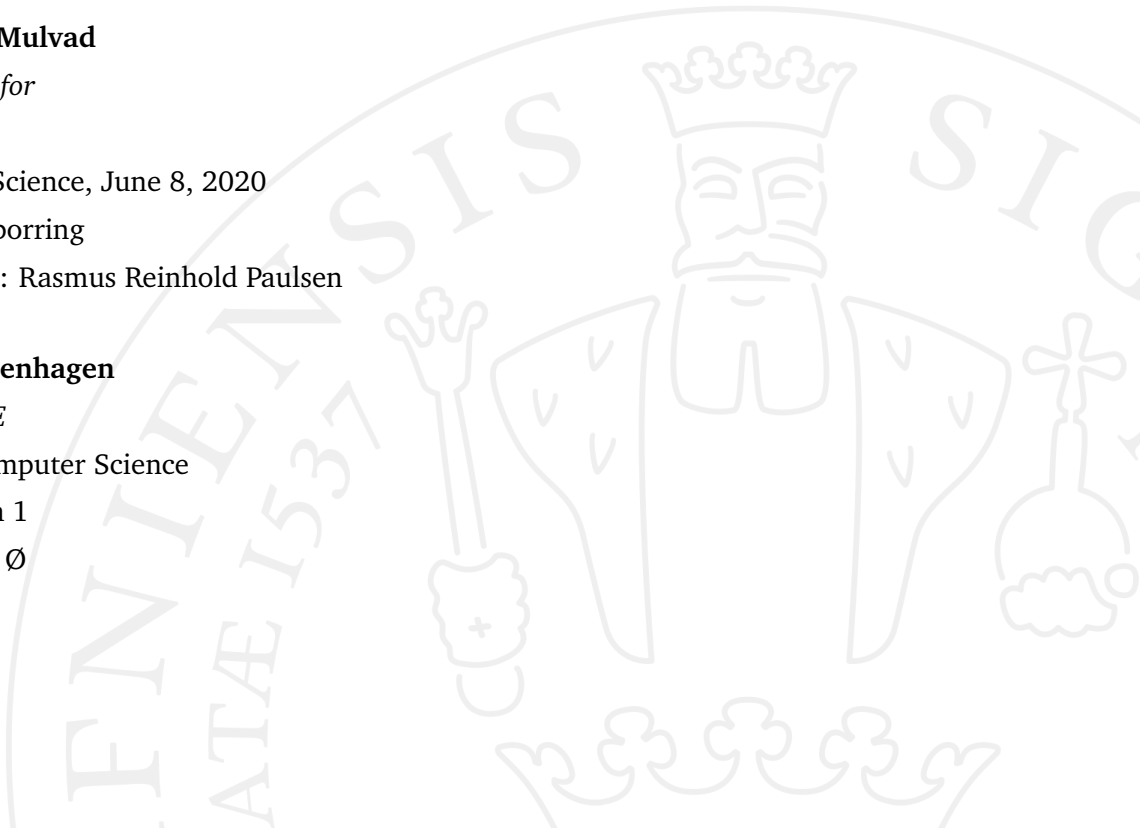
Caching Strategies for
Image Processing

BSc in Computer Science, June 8, 2020

Supervisor: Jon Sporring

External examiner: Rasmus Reinhold Paulsen

University of Copenhagen
Faculty of SCIENCE
Department of Computer Science
Universitetsparken 1
2100 Copenhagen Ø



Preface

This bachelor thesis has been completed as part of a 15 ECTS project at Department of Computer Science, University of Copenhagen (DIKU). Selecting this particular topic has been motivated by the fact that it is not unreasonable to have 3D-images in the magnitude 1000^3 and performing even simple operations on these can be challenging.

The intended reader of this paper is a student with basic knowledge of computer science equivalent to a Bachelor's degree. The student should be comfortable with the imperative programming paradigm and have a good grasp of computer architecture, in particular caching mechanisms, similar to the material taught in the course Computer Systems at DIKU. No prior specific knowledge of image processing algorithms is assumed.

I want to give big thanks to professor Jon Sporring for supervising me with this project. Without his help this paper could never have existed as he has been critical for feedback, contributing to the idea generation phase and making sure I stick to the time plan.

Also, big thanks to my mother, Lone Hougaard, who has assisted me in proofreading this paper even though she doesn't have the slightest interest in computer science.

I hope you will enjoy reading this paper and gain valuable information.

08/06/2020

Søren Hougaard Mulvad

Abstract

Working with large three-dimensional images is not a rare task in the field of image processing. Due to the large size of these images, they are unable to fit onto the faster storage devices on modern computers such as the L1 cache. And because of the CPU-memory gap present in computer architecture, this poses the need to utilize a data structure that minimizes the problems of locality when mapping multidimensional arrays onto the one-dimensional internal linear address space.

This thesis gathers the properties of three such data structures: Morton order, the block array and regular row-major array which is used as a standard in most programming languages. Four algorithms important in the field of image processing (recursive matrix multiplication, spatial convolution, fast Fourier transform and the fast marching method) are examined in a caching context when using these three data structures.

The results show that Morton order offers a promising increase in the runtime performance for common image processing tasks if it were to be efficiently implemented in a low-level language, but likely not an order of magnitude better. The block array was found to seldom perform better than Morton order.

Keywords: Image processing, caching, Morton order, voxel block array, row-major array, recursive matrix multiplication, spatial convolution, fast Fourier transform, fast marching method.

Contents

1 Introduction

In the field of image processing, it is not uncommon to work with large three-dimensional images i.e. of the size 1000^3. These can i.e. be generated from microscopes and performing even simple operations on them can be challenging.

The goal of this thesis is to study whether it is possible to reduce the runtime of standard image operations by simply changing the way the data is accessed. Two methods for storing the data known as *Morton order* as well as the *block array* are compared to *row-major arrays* which are used as a standard for multidimensional arrays in most programming languages.

The goal of Morton order and the block array for storing the data is to achieve better data locality in multidimensional space. To gain a better understanding of the data structures and their strengths and weaknesses, the theory behind these will at first be introduced and afterward a comparison of them is made in several situations in terms of the internal linear layout. Afterward, the algorithms for recursive matrix multiplication, spatial convolution, fast Fourier transform and the fast marching method will be introduced to the reader. These particular algorithms have been chosen since they are commonly used in image processing tasks, and they access the data in ways that are interesting in a caching context. Finally, the data structures' performances will be tested on these algorithms using a cache simulator and the findings will be discussed.

The code that has been used as a basis for the results and plots can be accessed at

`https://github.com/shmulvad/Caching-Strategies-for-Image-Processing`

and will be referred to as "the repo" in the remainder of this thesis.

Theory

2

2.1 Locality Properties

In computer architecture, there is a large *CPU-memory gap* between the speed of standard CPUs and memory hardware. The time between a request for data by the CPU and the arrival of the first requested piece of said data from main memory is currently[1] about 20 ns [3, p. 639]. During that time, a fast CPU can perform more than 60 floating-point operations. This memory gap has only gotten worse during the last 40-50 years since the speed of CPUs is progressing much faster than main memory latency [3].

To mitigate this, a *cache* is used. The cache is a memory component that replicates certain parts of main memory to allow faster access to the cached data. Cache memory can be made much faster and even keep up with CPU, but only if it is much smaller than typical main memory. Therefore, in modern computers the cache is typically orchestrated in a hierarchy of increasingly smaller but faster caches as shown in Figure 2.1:

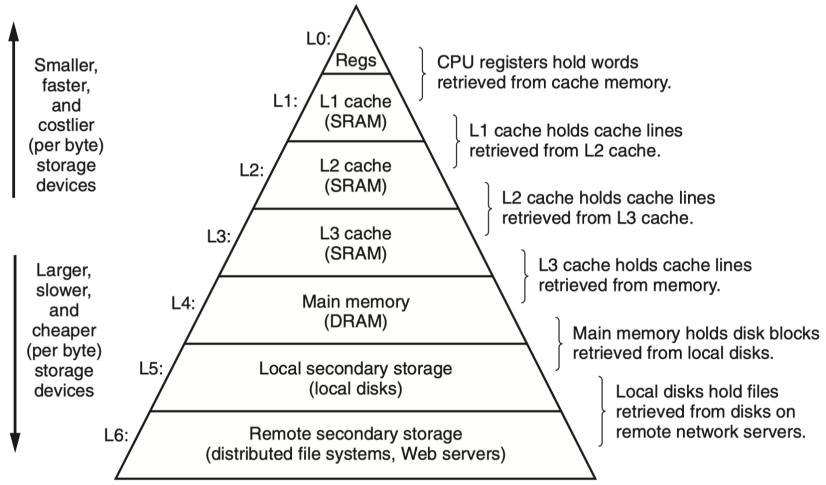


Figure 2.1: Example of cache hierarchy in modern computers. Image courtesy of [3].

[1] Numbers based on 2015.

This has the implication that the CPU will have vastly different access speeds to different parts of the main memory, leading us to the conclusion that classical analysis of algorithms, where we determine how the runtime depends on input size with the assumption that all operations execute with the same speed is perhaps a bit naive. Instead, we will often observe that for very small problem sizes, all data resides in the L1 cache and the algorithm will run at top speed. When the data no longer fits into the L1 cache, the speed is reduced to a level determined by the L2 cache and so on. This pattern is repeated for every level of the cache until we reach main memory.

This means that while the runtime of an algorithm will not change in asymptotic efficiency, these vast differences in access speed cannot be ignored in practice if we wish to fully exploit the available performance by making the implementations cache efficient.

2.1.1 Cache Memory

Caches do not store individual bytes or words, but *cache lines* which are small contiguous chunks of memory. Cache lines are always transferred to and from memory as one block meaning there is a direct mapping to corresponding lines in main memory. To speed up the process of the mapping, a constraint that these lines of memory can only be transferred to a small subset of lines is often implemented. If a memory line can be kept in n different cache lines, we say the cache is n-associative.

Whenever we use more data than can fit into the cache, we need a mechanism to decide what data should stay in the cache. Ideally, we would only remove cache lines that are no longer accessed. Naturally, the cache hardware can only guess the future access patterns and therefore cache lines are removed based on certain heuristics. A typical strategy is to remove the cache lines that were *least recently used* (LRU), but several other strategies also exist [3].

Hence an algorithm (and the implementation of it) will be faster if repeatedly accessed data is kept in the cache. Typically locality is described in two forms: *temporal* and *spatial* locality, where *temporal locality* means that a single piece of data, i.e. a single variable of a basic type, is repeatedly accessed during a short period of time and *spatial locality* means that after access to a data item, the next accesses will be to items that are stored in neighboring memory addresses. If the items belong to the same cache

line as the previously accessed item, then they have been loaded into the cache as well and the items are accessed extremely efficiently.

Whenever we access an item that has already been loaded into the cache we get a *cache hit*. If the item isn't to be found in the current cache level we get a *cache miss* and will have to search for the item in the next, slower cache level until we reach a sufficiently deep level where the data resides.

2.2 Data Structures

A fundamental problem when having to store multidimensional data such as images on a machine (whether on the drive or in the memory) is that the multidimensional indices have to be mapped to locations in the machine's *linear address space*. To accomplish this, we need a *space-filling curve* [2]. That is a curve that lays out the entire 2D square, or in general terms, an n-dimensional hypercube. The most well-known space-filling curve is probably the Hilbert-curve.

There are numerous different space-filling curves and therefore strategies for how to map multidimensional data.

The effectiveness of this mapping is crucial, since transversing the array in the same order that it is laid out in memory leads to a great spatial locality. However, if we are to access our array in such a way that it corresponds to jumping around in the linear address space we can expect a lot of cache misses, which leads to far worse running times. Therefore we are striving to find a mapping that preserves data locality.

2.2.1 Traditional Row/Column-Major Array

Most programming languages that offer support for multidimensional arrays generally use a *row-major order* (C, Python/Numpy by default, etc.) or *column-major order* (Fortran, etc.) to accomplish this. For an $M \times N$ 2D array A, the mapping $S(i, j)$, which expresses wherein linear address space the array element $A_{i,j}$ is stored, is (on a high level) given by the formulas

$$S^{(M,N)}_{\text{row-major}}(i, j) = j + N \cdot i, \qquad S^{(M,N)}_{\text{col-major}}(i, j) = i + M \cdot j \tag{2.1}$$

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63

(a) Row major order

	0	1	2	3	4	5	6	7
0	0	8	16	24	32	40	48	56
1	1	9	17	25	33	41	49	57
2	2	10	18	26	34	42	50	58
3	3	11	19	27	35	43	51	59
4	4	12	20	28	36	44	52	60
5	5	13	21	29	37	45	53	61
6	6	14	22	30	38	46	54	62
7	7	15	23	31	39	47	55	63

(b) Column major order

Figure 2.2: The same 8^2 array stored in respectively row major and column major order.

On a lower level, the number of bytes used per element w and the start address B of the array is also taken into account, giving us:

$$S^{(M,N)}_{\text{row-major}}(i,j) = B + w(j + N \cdot i)$$

For the rest of this paper, we will assume that the programming language takes care of handling the start address and the number of bytes per element and we will only ourselves consider the slightly higher level form seen in i.e. Eq. (2.1).

The equation we obtain when extending this schema to higher dimensions than 2D is simple to see and of the following form (here in row-major order):

$$S^{(M,N,C)}(i,j,k) = k + C \cdot j + C \cdot N \cdot i \tag{2.2}$$

For algorithms where we transverse the data in row-major order this kind of mapping ensures optimal spatial locality. However, a lot of standard image processing algorithms aren't necessarily biased towards accessing the elements in a row-major order fashion but tend to access pixels/voxels close to each other in abstract 2D/3D space. Consider a 3D image with dimensions $(124, 124, 124)$ where we are currently doing some operation at voxel $(50, 50, 50)$ and want to access the voxel at $(51, 50, 50)$. Using Eq. (2.2) where the only difference in these values is in the i-coordinate, this would have a distance in linear space of $124^2 \cdot (51 - 50) = 124^2 = 15376$.

Since the cache has multiple cache lines, this isn't always as big a problem as it might seem from the above example. We will now look at different types of data structures that map the indices that are in close proximity in multidimensional space to indices that are also close to each other in linear space.

2.2.2 Block Array

The fundamental idea behind the block array is to use several smaller square (K, K)-size normal row-major subarrays and stitch these together in a block of arrays [13]. We know that all the elements in a row-major array will be close in linear space if the array is fairly small.

	0	1	2	3	4	5	6	7
0	0	1	2	3	16	17	18	19
1	4	5	6	7	20	21	22	23
2	8	9	10	11	24	25	26	27
3	12	13	14	15	28	29	30	31
4	32	33	34	35	48	49	50	51
5	36	37	38	39	52	53	54	55
6	40	41	42	43	56	57	58	59
7	44	45	46	47	60	61	62	63

Figure 2.3: The layout of a 8^2 block array with $K = 4$.

For a block array, we first need to define the size K of the dimensions of the base row-major arrays used. We will assume a picture with dimensions $(M = m \cdot K, N = n \cdot K)$ is used where m and n are positive integers. When this is defined, we obtain the formula for accessing element (i, j) to be:

$$i_{\text{block}} = \lfloor i/K \rfloor, \qquad j_{\text{block}} = \lfloor j/K \rfloor, \qquad i_{\text{idx}} = i \text{ rem } K, \qquad j_{\text{idx}} = j \text{ rem } K$$
$$\text{block}_{\text{idx}} = j_{\text{block}} + n \cdot i_{\text{block}}, \qquad S^{(K,K)}_{\text{idx}}(i_{\text{idx}}, j_{\text{idx}}) = j_{\text{idx}} + K \cdot i_{\text{idx}}$$
$$S^{(M,N)}(i, j) = K^2 \cdot \text{block}_{\text{idx}} + S^{(K,K)}_{\text{idx}}(i_{\text{idx}}, j_{\text{idx}})$$

In reality, there is no need to implement this with actual subarrays but it is a useful abstraction to make to understand the operation. Since divisions are quite an expensive operation, we are interested in eliminating these. If we make sure to pick K such that $K = 2^k$ for some positive integer k, then we can make use of some tricks. We have that for some index i the floor of the division is

$$\lfloor i/K \rfloor = i \gg \log_2 K$$

where "$\gg$" denotes a bitwise right-shift. Right shifting by the log-value of a power of 2 is equivalent to first dividing by K and then discarding the remainder. Here, we can of course precompute $\log_2 K$ when creating the array so we don't have to perform an expensive $\log$ operation for every array indexing.

Furthermore, we can easily calculate the remainder as

$$i \text{ rem } K = i \,\&\, (K-1)$$

where we let "&" denote bitwise and. This holds since $(K-1)$ in binary will always be a bit string where the least significant bits are all 1's and the most significant bits are all 0's. When we do a *bitwise and* between this value and our index i we discard all the bits corresponding to values $\geq K$.

Like the traditional row/column-major arrays, the block array can easily be extended to higher dimensions. We will now use the tricks and notation described above to avoid divisions. Let the dimension of the original 3D picture be $(M = m \cdot K, N = n \cdot K, C = c \cdot K)$. The formula for accessing element (i, j, k) in a 3D voxel block array is then given by

$$\begin{aligned}
i_{\text{block}} = i \gg \log_2 K, \quad & j_{\text{block}} = j \gg \log_2 K, \quad & k_{\text{block}} = k \gg \log_2 K \\
i_{\text{idx}} = i \,\&\, (K-1), \quad & j_{\text{idx}} = j \,\&\, (K-1), \quad & k_{\text{idx}} = k \,\&\, (K-1) \\
\text{block}_{\text{idx}} &= k_{\text{block}} + c \cdot j_{\text{block}} + c \cdot n \cdot i_{\text{block}} \\
S^{(K,K,K)}_{\text{idx}}(i_{\text{idx}}, j_{\text{idx}}, k_{\text{idx}}) &= k_{\text{idx}} + K \cdot j_{\text{idx}} + K^2 \cdot i_{\text{idx}} \\
S^{(M,N,C)}(i, j, k) &= K^3 \cdot \text{block}_{\text{idx}} + S^{(K,K,K)}_{\text{idx}}(i_{\text{idx}}, j_{\text{idx}}, k_{\text{idx}})
\end{aligned}$$

If we were to consider our example from before of finding the distance in linear space between voxel $(50, 50, 50)$ and $(51, 50, 50)$ in a $(128, 128, 128)$-dimensional picture where we set $K = 16$, we notice that as long as i_{block} doesn't change the distance is given by:

$$i_{idx_{51}} = 51 \& (4-1) = 3, \qquad i_{idx_{50}} = 50 \& (4-1) = 2, \qquad 16^2 \cdot (i_{idx_{51}} - i_{idx_{50}}) = 256$$

which is far closer in linear space than our previous example. Of course, if we are at a location where the $\text{block}_{\text{idx}}$ changes in value the linear distance will be much bigger.

If we denote the dimensionality of our block array by d and the distance in neighbors we are considering from a given base pixel/voxel by n_e (which in the previous examples

have simply been 1), we can illustrate the pixels where a change in the value of $\text{block}_{\text{idx}}$ would occur as seen on Figure 2.4.

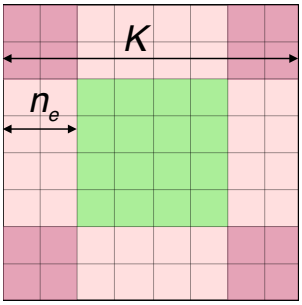


Figure 2.4: An illustration of the pixels that will need to access another subarray when in the block array for $K = 8$ and $n_e = 2$. Green pixels are ”safe” and only access elements inside their subarray. Light red pixels will need to access one additional subarray and the dark red pixels need to access three additional subarray.

From this, we see that we can easily write the proportion of pixels/voxels (the red pixels in Figure 2.4) where this happens as

$$\frac{K^d - (K - 2n_e)^d}{K^d} = 1 - \frac{(K - 2n_e)^d}{K^d} \tag{2.3}$$

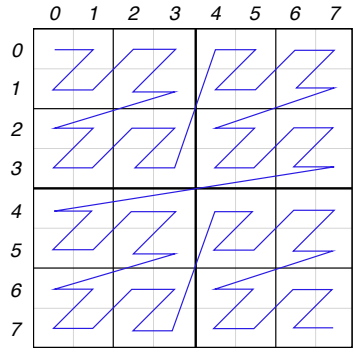
If we assume that d is fixed (a reasonable assumption since if we have a problem with i.e. a 2D image we can't just make it 3D), then it shows that we should make K as big as possible, while still making sure K^d is small enough to fit completely into memory.

2.2.3 Morton Order

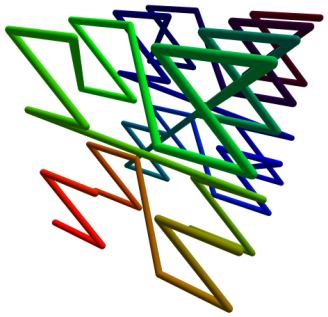
Morton order, also often called Z-order, is a type of storage similar to the block array but with some notable differences. Whereas the subarrays in the block array were fixed after we had settled on a given value for K, Morton order recursively divides our multidimensional array into smaller and smaller squares of size $(2^n)^2 = 2^{2n}$ in the 2D case or cubes of size $(2^n)^3 = 2^{3n}$ in the 3D case, both starting from $n = 1$.

	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

(a) Morton order of an 8^2 array.



(b) The Z-pattern drawn out for (a).



(c) Pattern of a 4^3 array. Image courtesy of [7].

Figure 2.5: Morton order (Z-order) in 2D of an 8^2 array and in 3D for a 4^3 array.

When introduced like this, it might seem like Morton order results in a large space waste for arrays that are not square/cube and not near a side length that is $\leq$ a power of 2. However, with some clever justification of all valid elements to the north and west the ”wasted” space lands in the south and east and can be viewed as a kind of padding, meaning the waste is only address space and therefore only little valuable fast memory is lost [17].

There are different strategies for how to obtain the 1D index from some given (i, j) or (i, j, k) coordinates. The most used and in most cases the fastest way to do this is through bit interleaving. This is accomplished by taking the bit representation of each index number, separating the bits by 1 in the case of 2D, 2 in the case of 3D and $n-1$ in the n-dimensional case. The act of separating the bits by some number is referred to as *parting* in the literature. When the bits have been parted, we left-shift the i-coordinate by 0, j-coordinate by 1, k-coordinate by 2 and so on and combine these bits by doing a ’bitwise or’ on them. The full process is best understood by studying Figure 2.6:

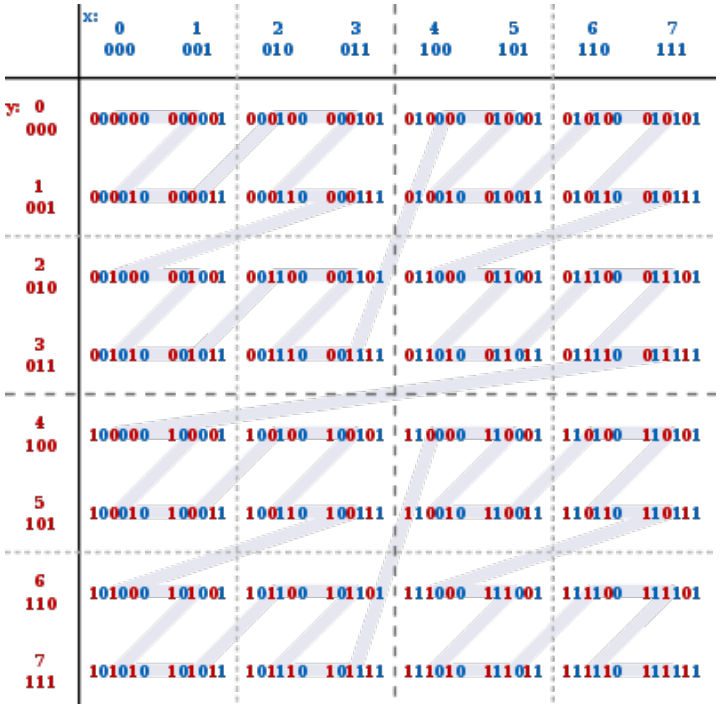


Figure 2.6: Morton bit interleaving visualized. Image courtesy of [7].

As an example, say we wanted the 1-dimensional index to access the element $(1, 6)$. Let "$\ll$" denote a bitwise left shift. Then we would do:

$$1_{10} = 001_2 \xrightarrow{\text{Part by 1}} 0\text{-}0\text{-}1 \xrightarrow{\ll 0} \text{-}0\text{-}0\text{-}1$$
$$6_{10} = 110_2 \xrightarrow{\text{Part by 1}} 1\text{-}1\text{-}0 \xrightarrow{\ll 1} 1\text{-}1\text{-}0\text{-}$$
$$\xrightarrow{\text{or}} 101001_2 = 41_{10}$$

Now that we know the overall process, the question becomes how to efficiently accomplish the first part of the process: Parting the bits. We can accomplish this by using a number of left shifts and some cleverly picked hexadecimal constants. Listing 2.1 shows the implementation used by this paper for the 3D Morton order. The code is heavily inspired by [7], but modified from C to Python and from parting by 1 to parting by 2. The comments show the positions of the bits after the operation on the given line has finished.

```python
def part1by2(x: int) -> int:
    """Inserts two 0 bits after each of the 11 low bits of x"""
    x &= 0x000007ff                  # x = ---- ---- ---- ---- ---- -a98 7654 3210
    x = (x | (x << 16)) & 0x070000ff  # x = ---- -a98 ---- ---- ---- ---- 7654 3210
    x = (x | (x <<  8)) & 0x0700f00f  # x = ---- -a98 ---- ---- 7654 ---- ---- 3210
    x = (x | (x <<  4)) & 0x430c30c3  # x = -a-- --98 ---- 76-- --54 ---- 32-- --10
    x = (x | (x <<  2)) & 0x49249249  # x = -a-- 9--8 --7- -6-- 5--4 --3- -2-- 1--0
    return x

def morton_encode_3(x: int, y: int, z: int) -> int:
    return (part1by2(z) << 2) | (part1by2(y) << 1) | part1by2(x)
```

Listing 2.1: Python code to get the internal Morton order index given three coordinates (x, y, z). The code is heavily inspired by [7].

We can take our previous example of computing a linear distance, but because the ordering of the "furthest" element is switched up as compared to the two other methods a fairer comparison is the distance between $(50, 50, 50)$ and $(50, 50, 51)$. Then we get $258108 - 258104 = 4$ (using Listing 2.1 to compute this). This number should be taken with a grain of salt since it highly depends on exactly at which coordinates we do the comparison. Morton order will have far worse results if it is at a location where two inner squares/cubes meet, i.e. between $(63, 63, 63)$ and $(63, 63, 64)$ the difference is 898780.

2.2.4 Comparison of the Three Data Structures

We see that the traditional row/column-major array is definitely the easiest way to get a space-filling curve. It has several advantages, namely that it is easy and fast to compute an index since it only uses $n - 1$ multiplications. Furthermore, no address space is wasted. If the end programmer has to do a simple `map` or `fold` over the data it is easy to write code to transverse an array in row-major order. The Morton order and the block array are significantly harder to transverse in a way that follows the internal space linearly, and it is unreasonable to expect the end-programmer to write loops doing this. Therefore their implementation has to support iterators or define higher-order functions for simple operations.

The big drawback of the row-major array and main reason this topic is studied is of course that for some tasks it doesn't do an optimal job of preserving data locality.

The defining feature of the block array is that it allows us to define how big each subarray should be (given a few constraints) and therefore tune it to the hardware. Furthermore, it can be adopted such that it doesn't waste any address space. It is also easy to adapt to higher dimensions. We do not however get the recursive data locality that we get with Morton order.

For a given voxel in an image, we can go through the $3 \times 3 \times 3$ grid of all the immediate neighbors of this voxel, compute the distance in linear address space and then take the average of these linear distances. If we go through this process of computing the average distance to the most immediate neighbors, but do it for all voxels in the image and at last take the average of all these averages, we get the results as shown in Table 2.1.

Image size	Morton order	Voxel block array	Row-major array
$4 \times 4 \times 4$	9.54	11.21	11.21
$8 \times 8 \times 8$	39.78	40.54	44.26
$16 \times 16 \times 16$	166.61	169.06	176.86
$32 \times 32 \times 32$	685.82	690.76	707.89
$64 \times 64 \times 64$	2787.28	2802.51	2833.36
$128 \times 128 \times 128$	11243.36	11282.46	11337.83
$256 \times 256 \times 256$	45169.30	45258.10	45360.92

Table 2.1: The average of the average immediate neighbor distance for the three different data structures.

Table 2.1 shows how Morton order in this specific way of measuring the distance is only slightly better than the row-major array in immediate average neighbor distance since there sometimes are very big distances when we "cross a border" in the subdivided squares.

This test can be extended to higher propagation levels. What is understood by "propagation level" is illustrated in Figure 2.7.

To get an idea of how the different data structures perform, we can plot the summed linear distance for a $128 \times 128 \times 128$ image measured in a few different ways as shown in Figure 2.8 and Figure 2.9.

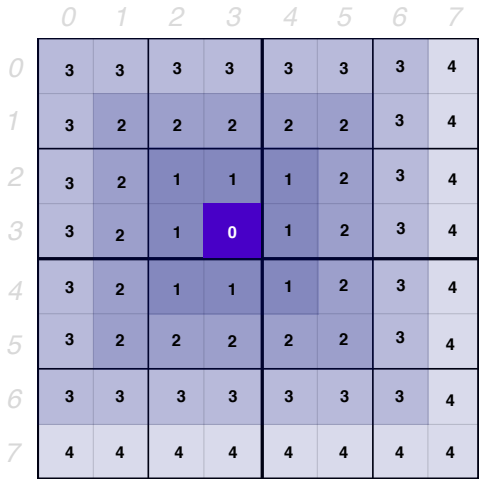


Figure 2.7: Propagation level in 2D with $(3, 3)$ as starting point illustrated. At each propagation level, the pixels with the given level number and all lower numbers are included.

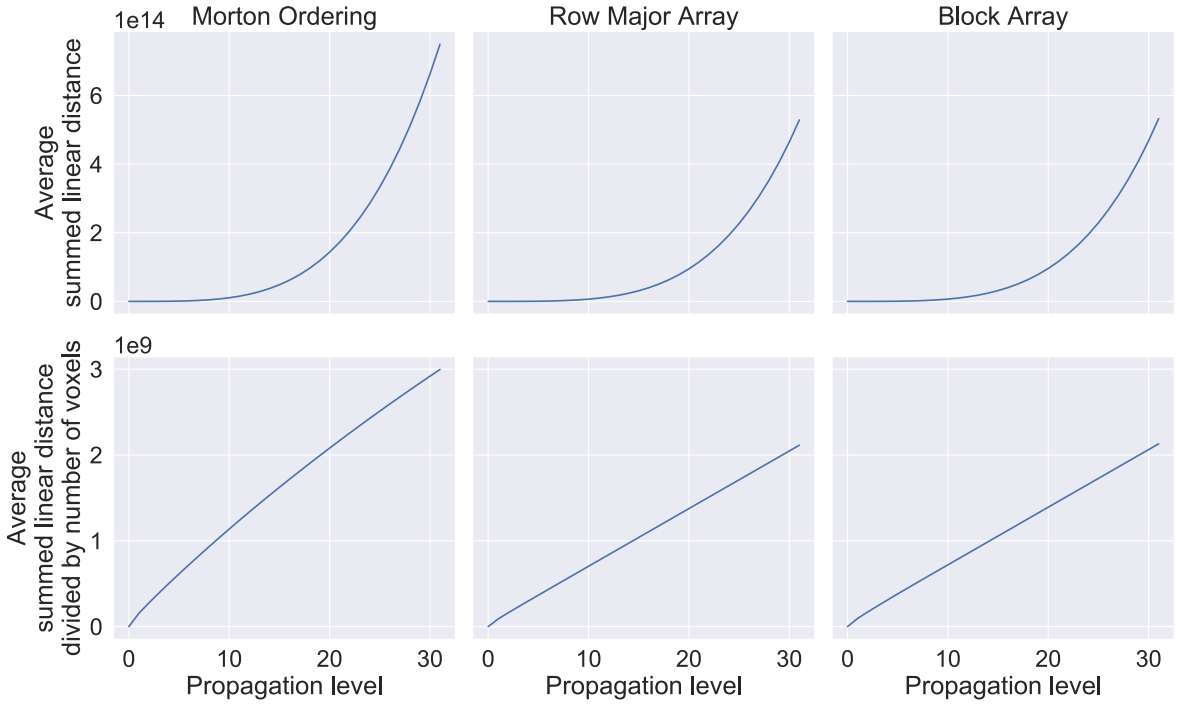
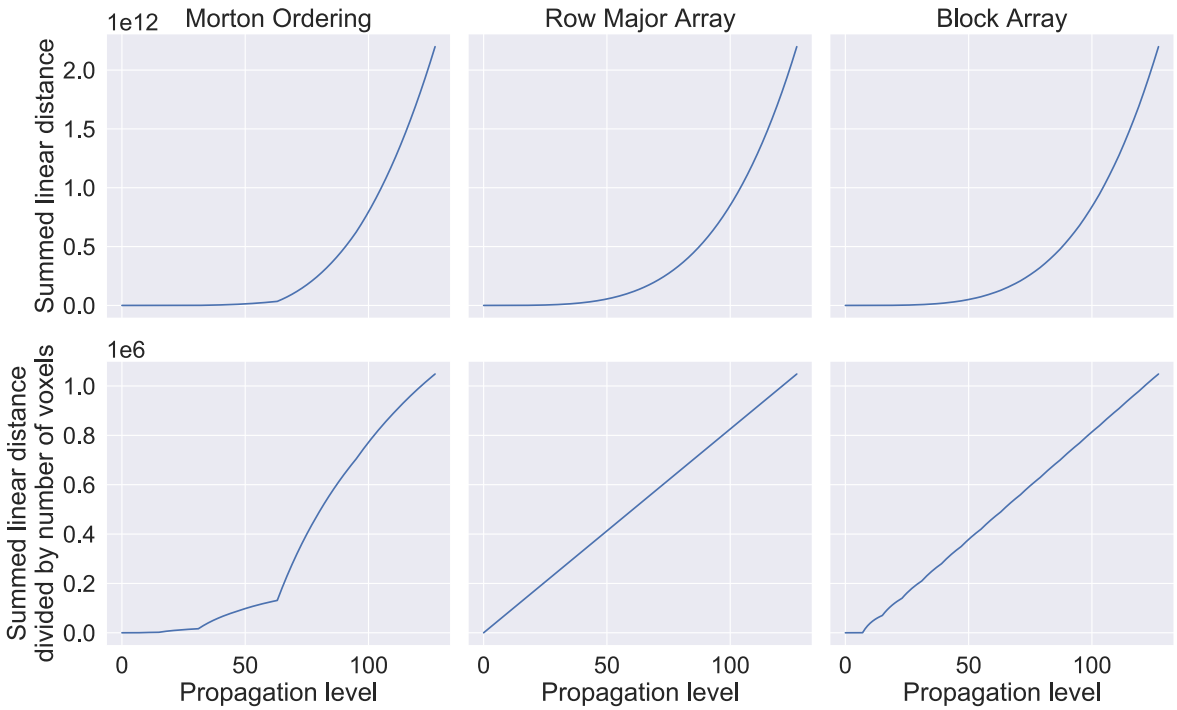
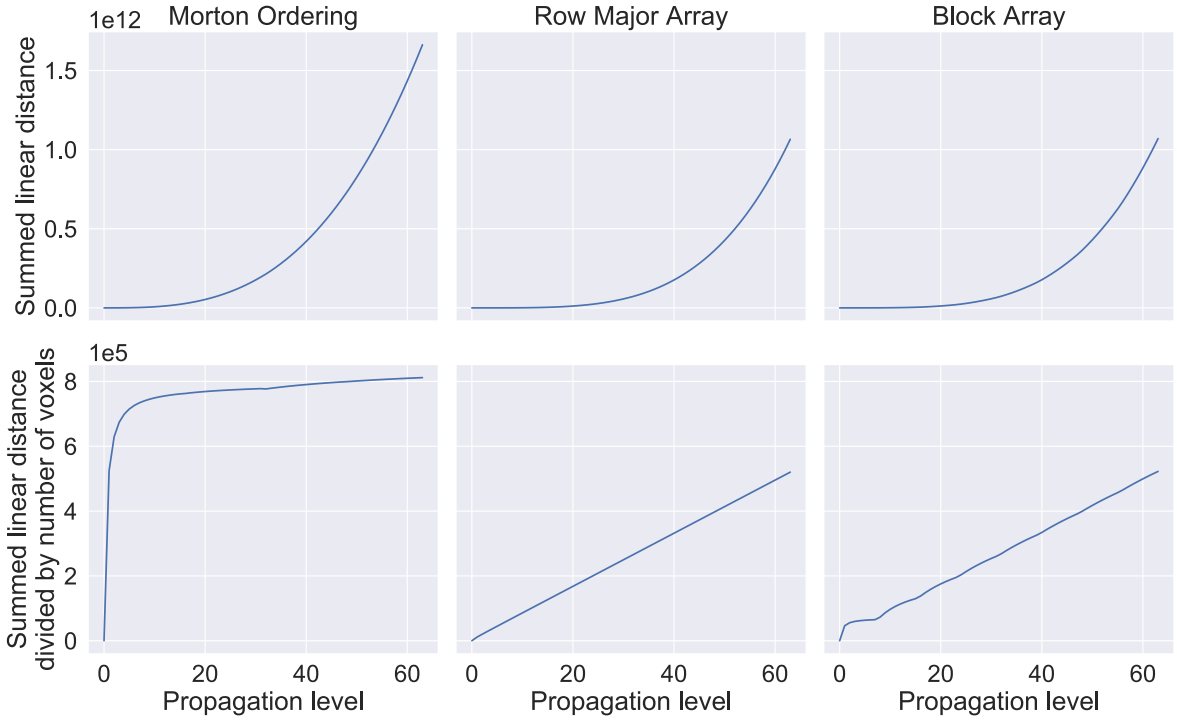


Figure 2.8: Summed linear distance and summed linear distance divided by the number of voxels for a $128 \times 128 \times 128$ image for the three data structures when averaging over all starting points given by the inner cube $(32, 32, 32) - (96, 96, 96)$ propagating 32 levels out.



(a) Propagating out with $(0, 0, 0)$ as start point.



(b) Propagating out with $(64, 64, 64)$ as start point.

Figure 2.9: Summed linear distance and same summed linear distance divided by the number of voxels for a $128 \times 128 \times 128$ image for the three data structures propagating out with respectively $(0, 0, 0)$ and $(64, 64, 64)$ as starting points.

In Figure 2.8, we have the summed linear distance when starting in all different points given by the inner cube $(32, 32, 32) - (96, 96, 96)$, propagating 32 levels out and averaging the value for each propagation level. In Figure 2.9, we have the summed linear distance when starting in two ”extreme” points – a corner and a midpoint – and propagating out the maximum possible levels out. Note that the x-ticks are shared across columns and the y-ticks are shared across rows.

From these plots, we notice a number of things. First of all, when looking at the average summed linear distance inside the inner cube of the image in Figure 2.8 we see that Morton order performs about $1.5x$ worse than the row-major array and the block array. This is likely because we are comparing the data structures inside the inner part of the image where Morton order is the weakest. This same effect can also be seen when studying Figure 2.9b where we use a starting point that is right beside the ”borders” of many inner cubes in the Morton order, therefore resulting in a very steep rise as compared to the two other data structures.

However, when we are in one of the corners as plotted in Figure 2.9a and therefore not crossing a lot of the inner borders in every direction we see that Morton order has a much more gentle curve than the two other data structures in the start. We also see that the sharp angles in the plot are at numbers that are a power of 2 which fits with the theory. Likewise, we can see a similar effect for the block array here for all multiples of 8 (which is used as the K-value here).

We also see that the plots of the row-major array performance are almost a perfect exponential in the case of the summed linear distance and a perfect linear function in the case of the summed linear distance divided by the number of voxels which is what we would expect.

At last, we see that the row-major array and the block array perform almost equivalently after the first K propagation levels.

Implementation-wise, an abstract class `CachingDataStructure` was implemented that defines a number of methods and abstract methods and properties, and then the three data structures are defined as subclasses that inherit from this class. The code as well as some examples of how to use it can be seen on the repo in the `src/data_structures/`-folder.

2.3 Algorithms

In the following section, four algorithms are briefly introduced. The algorithms introduced will later be the ones that are experimented on. In general, these algorithms have been chosen partly because they are important in an image processing context and partly because they have interesting data access patterns. We will especially see algorithms that tend to subdivide the problem into smaller squares/cubes, and algorithms that tend to access neighbor pixels/voxels. For implementations of the algorithms, the reader is referred to the `src/algorithms/`-folder in the repo.

2.3.1 Recursive Matrix Multiplication

The reason matrix multiplication is interesting to study in the context of image processing and caching algorithms is that it has immense importance in scientific computing and is the backbone of many important algorithms. The tiled pattern below in the pseudocode is really interesting in a caching context and one can easily imagine how it might lend itself well to Morton order.

If we have two $n \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$ where $n = 2^t$, the matrix multiplication $\mathbf{C} = \mathbf{AB}$ can be expressed in block form as

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ \hline A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{array}\right]$$

where each of the blocks is of size $n/2 \times n/2$. From this, we easily see how we can do the matrix multiplication recursively. We can write this procedure in pseudocode as seen in Algorithm 1 [16].

All the multiplications will be of tiles of size $2^{t-r} \times 2^{t-r}$ where r is the recursion depth we wish to use. Note that the asymptotic complexity is still $\mathcal{O}(n^3)$ so the algorithm is mostly interesting in the context of caching.

Algorithm 1: Recursive Matrix Multiplication.

```
1  mm_recursive(A, B, C, n, r)
2      if r ≤ 0 or n ≤ 2
3          matmul(A, B, C)                    // Standard matrix multiplication
4      else
5          mm_recursive(A_00, B_00, C_00, n/2, r − 1)
6          mm_recursive(A_01, B_10, C_00, n/2, r − 1)
7          mm_recursive(A_00, B_01, C_01, n/2, r − 1)
8          mm_recursive(A_01, B_11, C_01, n/2, r − 1)
9          mm_recursive(A_10, B_00, C_10, n/2, r − 1)
10         mm_recursive(A_11, B_10, C_10, n/2, r − 1)
11         mm_recursive(A_10, B_01, C_11, n/2, r − 1)
12         mm_recursive(A_11, B_11, C_11, n/2, r − 1)
```

2.3.2 Spatial Convolution

Spatial convolution is an important step in many image-related tasks. The reason it is interesting to study in this context is its wide usability and the pattern of its pixel/voxel accesses. As we will see in the following pseudocode, spatial convolution essentially transverses all voxels in the image where it for each of these voxels visits its most immediate neighbors in all directions.

Spatial convolution is used to apply an n-dimensional *kernel* to all elements in an n-dimensional data structure. As an example, if we were to apply the 3×3 kernel `1/9 * [[1, 1, 1], [1, 1, 1], [1, 1, 1]]` to some 2D picture it would have the effect of taking the average value of all the neighbor pixels which effectively applies a blurring to the image. Pseudocode of the spatial convolution algorithm can be written as seen in Algorithm 2 assuming a square kernel and no special handling of the borders [11, p. 207].

Algorithm 2: Spatial Convolution.

```
1  spatial_convolution_3d(image, output, kernel)
2      for x = 0 to image.x_dim−1
3          for y = 0 to image.y_dim−1
4              for z = 0 to image.z_dim−1
5                  output [x, y, z] = convolve_voxel(image, kernel, x, y, z)
6      return output
7  convolve_voxel(image, kernel, x, y, z)
8      voxel_sum = 0
9      k_2 = ⌊kernel.length/2⌋
10     for i = 0 to kernel.length−1
11         for j = 0 to kernel.length−1
12             for k = 0 to kernel.length−1
13                 voxel_sum[x, y, z]+ =
                     kernel[i, j, k] · image[x + i − k_2, y + j − k_2, z + k − k_2]
14     return voxel_sum
```

Note that the order the `for`-loops is run, and even how every individual voxel is accessed, has no effect on the algorithm's correctness. Therefore, when implementing the pseudocode, we can choose the ordering most suitable for good spatial locality. In practice, spatial convolution is often implemented with several matrix multiplications.

2.3.3 Fast Fourier Transform

The reason the fast Fourier transform (FFT) is interesting in a caching context is that it accesses the elements in a recursively subdivided even-odd index fashion along a given axis and then repeats for remaining axes. In image processing, it is a common method for image recognition, image filtering as well as a method to speed up certain other computations [14].

The one dimensional discrete Fourier Transform (DFT) transforms an array consisting of the complex numbers $x_0, \ldots, x_{N-1}$ as: [5]

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2i\pi kn/N} \qquad k = 0, \ldots, N-1 \tag{2.4}$$

We see that computing Eq. (2.4) directly has an asymptotic runtime of $\mathcal{O}(N^2)$. However, it can be shown that numerous computations are repeated and an $\mathcal{O}(N \log N)$ asymptotic runtime is obtainable using a divide-and-conquer approach. The approaches with this runtime are referred to as fast Fourier transform (FFT). The problem can be repeatedly split into subdivisions of respectively even and odd indices such that we get the following sub-arrays at each recursion level:

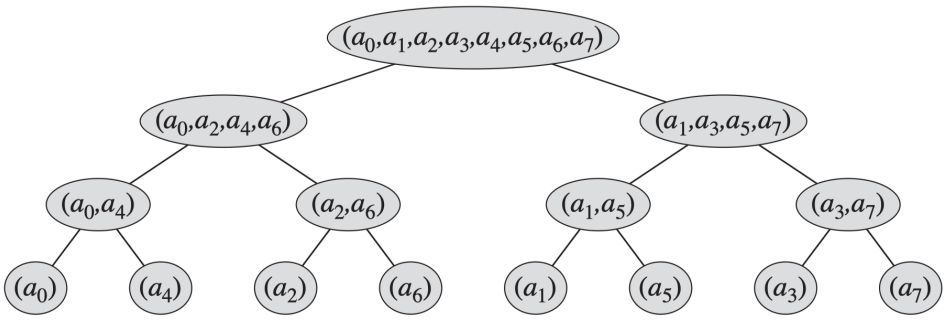


Figure 2.10: Subarrays obtained at each recursion level in typical recursive FFT implementations such as Cooley–Tukey factorization. Image courtesy of [5].

Since the recursive method involves creating a large number of temporary arrays it can be hard to measure the effects on the simulated cache. CLRS gives an iterative algorithm as shown in Algorithm 3 based on the recursive Cooley-Tukey factorization

[5]. The iterative algorithm accesses the array indices similarly to Figure 2.10 but without creating auxiliary arrays (it can be implemented with $\mathcal{O}(1)$ auxiliary storage).

Algorithm 3: In-place iterative FFT based on CLRS p. 918 [5].

```
1  iterative_fft(a)
2      bit_reverse_inplace(a)
3      n = a.length                                  // n is a power of 2
4      for s = 1 to lg n
5          m = 2^s
6          ω_m = e^{2πi/m}
7          for k = 0 to n − 1 by m
8              ω = 1
9              for j = 0 to m/2 − 1
10                 t = ωa[k + j + m/2]
11                 u = a[k + j]
12                 a[k + j] = u + t
13                 a[k + j + m/2] = u − t
14                 ω = ω ω_m
15     return a
```

Here `bit_reverse_inplace(a)` is a function to modify an array a in-place where we for each index swap the value with the value at the index given by the integer of the reverse bit string. I.e. for an array of length 8 at index 3, the swap-index would be computed as $3_{10} \rightarrow 011_2 \xrightarrow{\text{bit-reverse}} 110_2 \rightarrow 6_{10}$. Doing this for all indices gives us the array we would obtain if we concatenated all the values at the bottom recursion level of Figure 2.10.

The DFT is also defined in multidimensional space: [1, p. 20]:

$$X_{\mathbf{k}} = \sum_{\mathbf{n}=0}^{\mathbf{N}-1} x_{\mathbf{n}} e^{-2i\pi \mathbf{k} \cdot (\mathbf{n}/\mathbf{N})} \tag{2.5}$$

where $x_{\mathbf{n}}$ is an array consisting of a d-dimensional vector of indices $\mathbf{n} = (n_1, \ldots, n_d)$. This can be computed by performing one-dimensional FFTs along each axis for each row/column/etc. and then performing along the remaining axes as illustrated in Figure 2.11. The order this is done in is irrelevant from a correctness perspective.

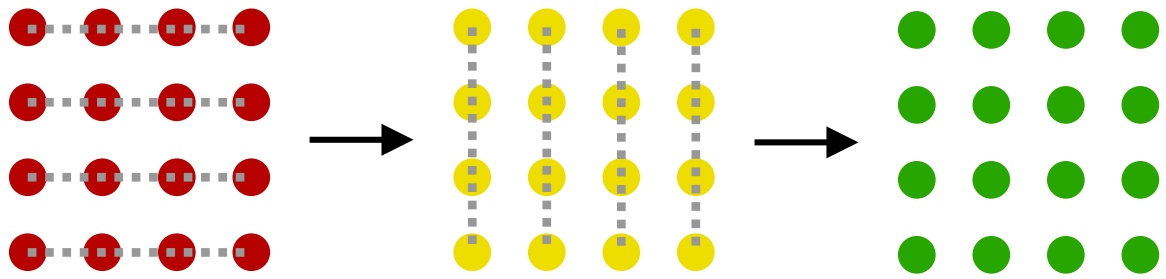


Figure 2.11: Computing the multidimensional FFT for a 4×4 array using eight individual 1D FFTs. At first (red entries) the array is given and four individual 1D FFTs are performed on the rows, modifying the values. Afterward (yellow entries) four individual 1D FFTs are executed on the columns, modifying the values. When this has finished (green entries) the multidimensional FFT has been computed.

Algorithm 4: Multidimensional FFT

```
1 multidim_fft(a)
2     n = a.length                                  // n is a power of 2
3     for axis = 0 to a.dim − 1
4         for key in all permutations of (indices in a excluding axis)
5             Perform iterative_fft(a) holding key fixed and varying the
                index along the excluded axis
6     return a
```

2.3.4 Fast Marching Method

The reason the fast marching method (FMM) is interesting in a cache context is that it – like spatial convolution – tends to access its immediate neighbors often. However, it isn't done nearly as structured. Furthermore, a fair amount of arrays are in play (the original array containing the speed function F, an array containing the values for the time function T and a temporary array containing the status which can be either accepted, neighbor, or far for each point. All of these arrays are introduced properly and elaborated on in the following section). FMM can be used for i.e. image segmentation, path-finding or computing signed distance fields [9].

The goal of FMM is to solve a discretized version of the Eikonal equation, $|\nabla T| = 1/F$, on a uniformly spaced grid where F is a *speed function* and the solution $|\nabla T|$ is the

”arrival time” for each point on the grid based on an originating point chosen manually [12].

FMM can be thought of as a liquid that is dropped on an uneven surface (in two dimensions or higher) at an initial point. The way this liquid propagates depends on the smoothness of the surface at all grid points. FMM computes the arrival time of the first drop of liquid at each point on the surface.

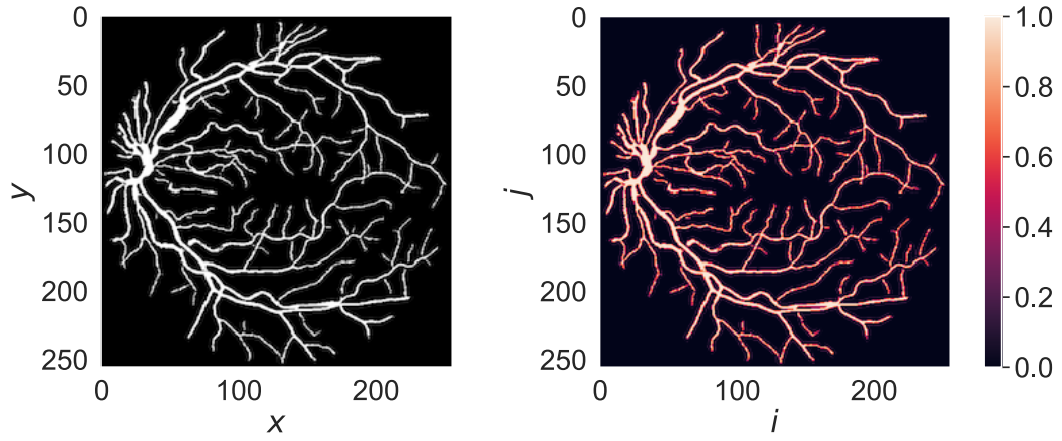
To compute this, the *speed function* F has to be supplied. As an example, consider Figure 2.12 in which the grayscale pixel values scaled to $[0, 1]$ of a pre-segmented image of blood vessels is used as the speed function F_{ij}. Here we have chosen our values such that it is faster to move along a blood vessel. If we start propagating from $(20, 120)$ and always move in the fastest direction, we get the arrival times at each point as shown in Figure 2.12c.

As described, for initializing FMM we have to choose a starting point (i, j) on the grid. When this has been done, we define $T_{ij} = 0$. All the points on the grid are divided into three different sets of either *accepted*, *neighbor* or *far*. When the algorithm first starts, the starting point is set to *accepted* (since we know T_{ij} at this point), its immediate neighbors are set to *neighbor* and all remaining points are set to *far*. This division can be seen in Figure 2.13.

When this division into sets has finished, FMM computes the value of T_{ij} for all (i, j)-values of newly added neighbors and updates them if the new value is smaller than the previous value. Afterward, the coordinates of the neighbor with the smallest value are extracted, it is moved to the set *accepted* and its immediate neighbors are moved to the set *neighbor*.

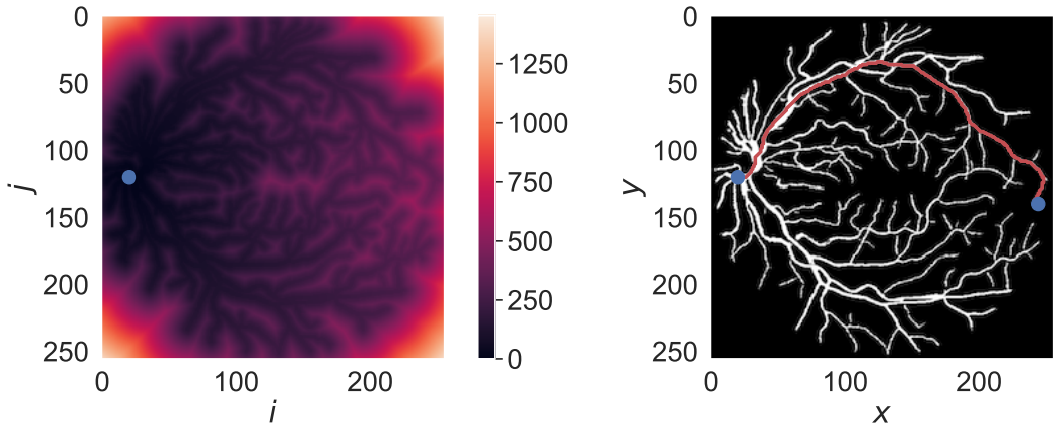
This process continues until we either have found our desired endpoint or all points in the grid are *accepted*.

FMM can be described as shown on Figure 2.14.



(a) Original segmented blood vessel image.

(b) Speed function F_{ij} based on (a).



(c) Arrival time function T_{ij} computed using F_{ij} originating from the blue point $(i, j) = (20, 120)$.

(d) Shortest path (red) from start point $(20, 120)$ to end point $(245, 140)$ computed using T_{ij}.

Figure 2.12: The application and results of FMM on a 256×256 pre-segmented blood vessel image. Image courtesy of [8] (of the original segmented image shown in (a)).

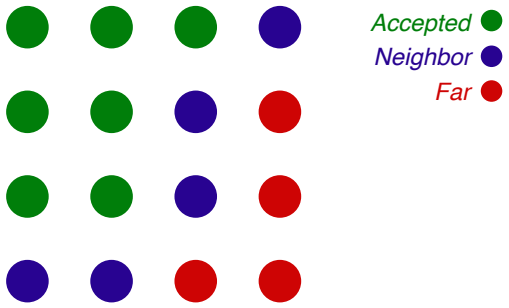


Figure 2.13: The *accepted*-, *neighbor*- and *far*-sets of a 4×4 grid with upper-left coordinate chosen as the start point and six full iterations of FMM having passed.

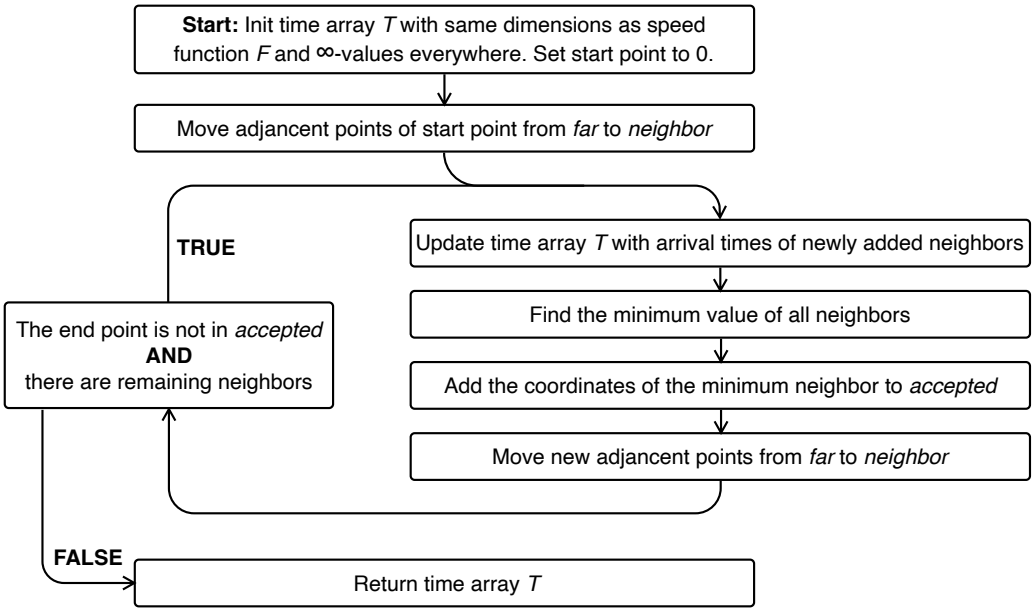


Figure 2.14: Flowchart of the FMM algorithm.

For more detailed info about the Eikonal equation or how the arrival times are computed, the reader is referred to [10, p. 6]. From Figure 2.14 we see a strong similarity with Dijkstra's algorithm in the regard that one new element is added to the set of accepted elements during each iteration and that we always choose the minimum value so far as our basis for further computation.

2.3.5 Other Notable Algorithms

Other image processing algorithms such as dilation and erosion have also been considered to be implemented and run experiments on. However, the way these two algorithms transverse the elements are largely symmetric to convolution and it was therefore judged that it would not be beneficial to test these since the result would almost mirror spatial convolution. Vector distance transform is a method to compute signed distance fields (SDF) that have also been considered, but in the end judged not to be tested since FMM can already be used to compute SDF [9].

3 Experiments and Results

3.1 Experimental Setup

The plan for conducting the experiments was at first to write programs that used different algorithms and data structures, execute these while measuring the time it took for them to complete and compare this (while ensuring the programs' correctness).

However, it quickly became clear that this would not be a good idea. The programmer typically has none or very limited influence on what data is kept in the cache and therefore testing different algorithms without taking this into account would give skewed results. Furthermore, comparing directly against i.e. the built-in method for performing FFT in Scipy would be unfair since a lot of the code of this manner has been written in C. From quick experimentation, running the naive recursive fibonacci function, `fib = lambda n: 1 if n <= 1 else fib(n-1) + fib(n-2)` in Python and in C shows C is about $50x$ faster.

To ensure this wouldn't cause any problems, instead of measuring the time directly the cache hierarchy has been simulated and then cache hits, misses, etc. can be compared. The open-source `pycachesim` library [4] is used for the cache simulation.

3.1.1 Cache Simulation With pycachesim

When performing operations, the different data structures have been set such that whenever the element at a given index is read or written to, the accessed address location gets sent to the simulator. `pycachesim` supports simulating common cache architectures found in modern processors (multiple levels mimicking a normal cache hierarchy) as well as multiple replacement strategies and has been extensively tested.

The simulator keeps a statistic about accumulated hit, miss, load, store and evict counts. After the caches have been warmed up, the statistic is reset and data accesses are

```
1  from cachesim import CacheSimulator, Cache, MainMemory
2  l2 = Cache("L2", 512, 8, 64, "LRU")  # 256KB: 512 sets, 8-ways with cacheline size of 64 bytes
3  l1 = Cache("L1", 64, 8, 64, "LRU", store_to=l2, load_from=l2)  # 32KB
4  mem = MainMemory(); mem.load_to(l2); mem.store_from(l2)
5  cs = CacheSimulator(l1, mem)
6
7  cs.load(2400)
8  cs.store(256, length=8)
9  cs.load(256, length=8)
10 cs.force_write_back(); cs.print_stats()
11 # CACHE     ***HIT***     ***MISS***     ***LOAD***     ***STORE***    ***EVICT***
12 #    L1     1 (  8B)      2 ( 65B)       3 ( 73B)       1 ( 8B)        1 (64B)
13 #    L2     0 (  0B)      2 (128B)       2 (128B)       1 (64B)        1 (64B)
14 #   MEM     2 (128B)      0 (  0B)       2 (128B)       1 (64B)        0 ( 0B)
```

Listing 3.1: A minimal working example of the cache simulator using `pycachesim`. In line 7 we load one byte from address 2400, which should be a miss in both cache levels. In line 8 we store 8 bytes to addresses 256-263. This will result in a load miss due to write-allocate. In line 9 load from address 256 until (exclusive) 264 which will be a hit.

passed to the simulator. To get an idea of how the simulator works, the very simple simulation in Listing 3.1 can be considered.

In the experiments conducted, it is assumed we are working with data that takes up 8 bytes for each element. To allow for easy setting of the values, in the abstract class for the different data structures `__getitem__(self, key)` and `__setitem__(self, key, value)` have been implemented as shown in Listing 3.2.

```
1  def __setitem__(self, key: tuple, value: Any) -> None:
2      idx = self.internal_index(*key)
3      if self.cache: self.cache.store(8 * (idx + self.offset), length=8)
4      self.data.__setitem__(idx, value)
5
6  def __getitem__(self, key: tuple) -> Any:
7      idx = self.internal_index(*key)
8      if self.cache: self.cache.load(8 * (idx + self.offset), length=8)
9      return self.data.__getitem__(idx)
```

Listing 3.2: Template for setting and getting items. `self.internal_index(key)` returns the index in the space-filling curve, i.e. the Morton encoding in the case of Morton order. This allows us to do i.e. `morton_data[5, 19, 200] = 0.81` and a store operation will be sent to the cache simulation at the appropriate address. Here `data` is a 1D array.

The cache that has been used in the experiments is described in Table 3.1:

	Sets	Ways	Cacheline size	Total size
L1	64	8	64	32 KB
L2	512	8	64	256 KB
L3	20480	16	64	20 MB
MEM	-	-	-	∞

Table 3.1: The cache used for the simulations. All the levels uses LRU policy. Numbers based on documentation from Intel [6].

These numbers mimic caches in modern computer processors based on documentation from Intel [6]. In the plots and tables in the following sections, only the effects (and in particular hits and misses) on the L1 cache has been studied in depth. However, the data for all cache levels is available at the repo in the `src/results/`-folder in JSON-format.

3.2 Recursive Matrix Multiplication Results

A matrix multiplication using the algorithm described in section 2.3.1 is first performed for a pair of each of the data structures. All data structures have been assigned their own cache to simulate on. The first multiplication is to warm up the cache. Afterward, these caches' stats are reset and one more matrix multiplication is performed for each of the data structures. The stats after the matrix multiplication has finished are saved. In pseudocode, the process can be described as follows:

```
results = []
for n in ns:
    for tile_size in tile_sizes:
        if tile_size >= n:
            continue
        // make cache1, cache2 and cache3
        // make two n x n arrays of random values
        // assign two morton arrays cache1 and random values
        // assign two row major arrays cache2 and random values
        // assign two block arrays cache3 and random values
```

```
11          // warm up caches by performing the matrix multiplications
12          // reset cache1/2/3 stats
13          matmul_rec(morton1, morton2) // do experiment1
14          matmul_rec(row_arr1, row_arr2) // do experiment2
15          matmul_rec(block_arr1, block_arr2) // do experiment2
16          results.append({ "morton": cache1.stats(),
17                           "row_arr": cache2.stats(),
18                           "block_arr": cache3.stats() })
```

The n-values that have been tested have been chosen partly on the basis that they should be some value 2^i where i is an integer and partly based on the fact that performing the cache simulation takes quite some time, and it should be able to be run in a reasonable time. The tile sizes have been chosen as all possible tile sizes $2^i \leq$ the given n tested. Since it is difficult to visually compare the absolute number of hits, the relative performance is plotted. Using this setup, the results in Figure 3.1 have been obtained for the hit count for the L1 cache:

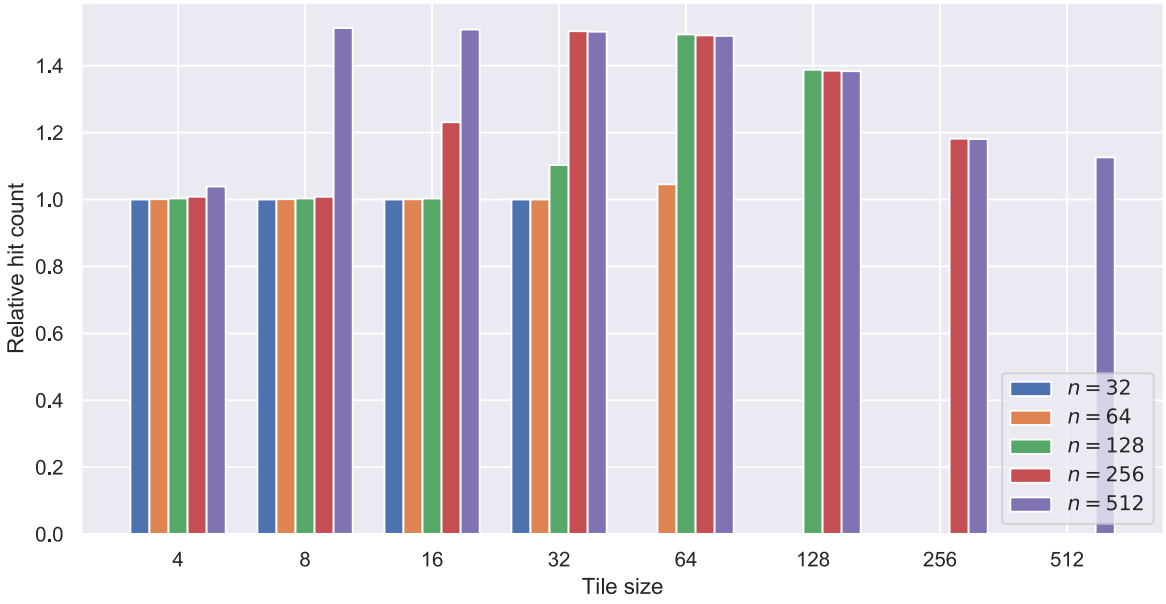


Figure 3.1: Relative hit count of Morton order vs. row-major array for $n \times n$ arrays in the L1 cache. A value above 1 means that Morton order has more L1 cache hits.

We see that the relative performance is almost the same for tile sizes less than 32 when $n < 128$. We also see that when the sizes of the data are small we don't get that much gain since most of it can fit into memory. For values above this, we see that i.e. $n = 256$

gives Morton order a relative performance of about $1.5x$ for tile sizes greater than 32, looking to be slightly decreasing.

When looking at the miss count in the L1 cache, we get the results shown in Figure 3.2:

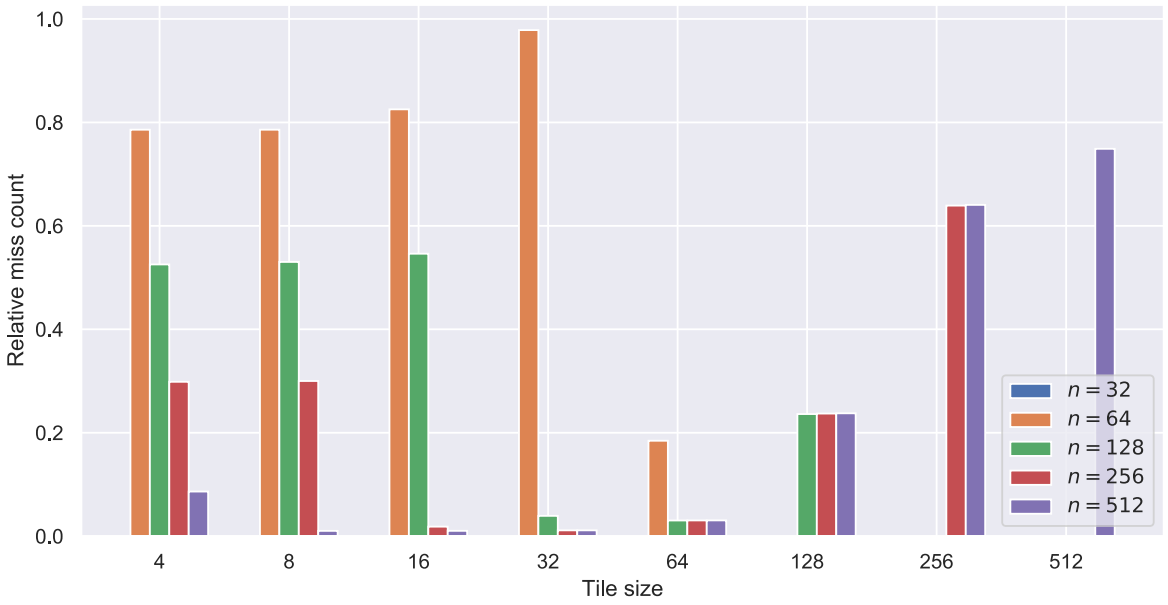


Figure 3.2: Relative miss count of Morton order vs. row-major array for $n \times n$ arrays in the L1 cache. A value below 1 means that Morton order has fewer L1 cache misses.

At first glance, it might seem baffling that Figure 3.1 looks so different from Figure 3.2. But this is due to there being way more hits than misses. We know that the sum of hits plus misses should be consistent across data structures. Consider if we i.e. for Morton order have 199,000 hits and 1,000 misses while we for the row-major array have 190,000 hits and 10,000 misses. Then the discrepancy in the two figures should make sense since the relative hit rate in this example will be $1.05x$ while the relative miss rate is $0.10x$. If we assume a 1 ns latency of the L1 cache and a 10 ns latency of the L2 cache, going from a 99% hit rate to a 95% hit rate will not result in a 4% decrease in latency performance – it will result in a 25% decrease (assuming the missed data is always available in the L2 cache).

From Figure 3.2, we see that Morton order results in significantly fewer misses for certain tile sizes. However, the proportion of hits vs. misses isn't possible to see from the graph as discussed above and has to be taken into account. Across all three data structures and sizes of n, it was found that the hit-to-miss ratio was the highest

(meaning they performed the best in a caching sense) when using a tile size of 4. For this tile size, the hit-to-miss ratios are shown in Table 3.2.

n	64	128	256	512
Morton order	283.12	283.12	283.12	283.12
Row-major array	222.29	148.28	83.82	23.54
Block array	222.29	222.29	222.29	222.29

Table 3.2: L1 cache hit to miss ratio for different sizes of n when using a tile size of 4.

Maybe somewhat surprisingly, when using a tile size of 4 the hit-to-miss ratio is consistent to two decimal places across different sizes of n for Morton order and the block array. This is likely because the recursive nature of the algorithm fully utilizes the properties of Morton order and the block array. When the tile size is the same as the side length, standard naive matrix multiplication is performed right away. In these cases, we see from the plots that Morton order still performs better which is because naive matrix multiplication is equivalent to both performing a row-major transversal combined with a column-major transversal. The row-major transversal will of course be faster in the row-major array, but performing the combination of these gives Morton order an edge. This effect is examined in detail in [15].

This shows that Morton order vastly improves locality for this algorithm, but it should be kept in mind it is mostly interesting from a theoretic point of view since the asymptotic runtime is $\mathcal{O}(n^3)$. These findings mimic [16] where a similar experiment has been run, but for bigger n-values and where the execution time has been measured instead.

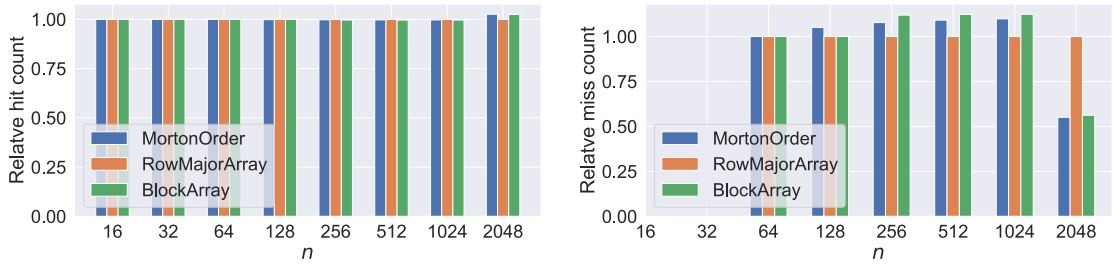
3.3 Spatial Convolution Results

The spatial convolution algorithm has been tested in a similar manner as for matrix multiplication, but naturally only with varying sizes of n. A kernel of size 3×3 has been used in the 2D case and $3 \times 3 \times 3$ in the 3D case. When looping through all the entries, it has been done in an order that follows the internal linear layout of the data structure in question such that cache locality is maximized in all cases.

```
1  results = []
```

```
 2  for n in ns:
 3      // make cache1, cache2 and cache2
 4      // make a n x n (or n x n x n) array of random values
 5      // assign caches and random values to morton, rm_arr, block_arr
 6      // warm up caches by performing one spatial convolution
 7      // reset cache1/2/3 stats
 8      spatial_convolution(morton, kernel)    // do experiment1
 9      spatial_convolution(rm_arr, kernel)    // do experiment2
10      spatial_convolution(block_arr, kernel) // do experiment3
11      results.append({ "morton": cache1.stats(),
13                       "rm_arr": cache2.stats(),
14                       "block_arr": cache3.stats() })
```

With this setup, the results in Figure 3.3 is obtained:



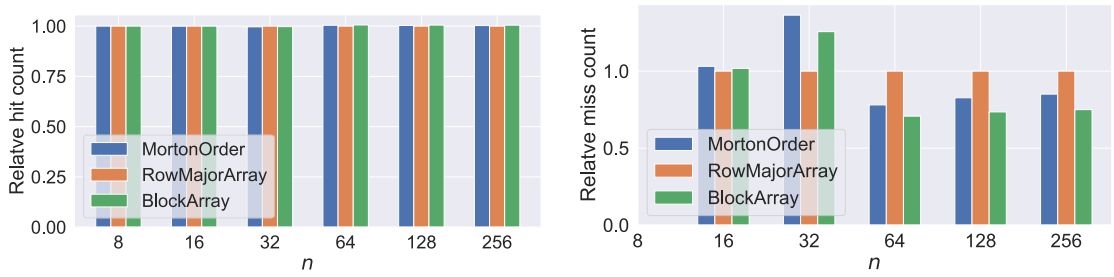
(a) Relative L1 cache hit count scaled by the performance of the row-major array. A value above 1 means more hits than the row-major array.

(b) Relative L1 cache miss count scaled by the performance in the row-major array. A value below 1 means fewer misses than the row-major array.

Figure 3.3: Relative performance of 2D spatial convolution scaled by the performance in the row-major array.

We see that the performance between the different data structures is almost similar and neither Morton order nor the block array does a better job than standard row-major arrays up until $n \leq 1024$ – actually, they perform slightly worse. For $n = 2048$ the relative miss count is significantly better.

In 3D, the results as seen in Figure 3.4 have been obtained.



(a) Relative L1 cache hit count scaled by the performance in the row-major array. A value above 1 means more hits than the row-major array.

(b) Relative L1 cache miss count scaled by the performance in the row-major array. A value below 1 means fewer misses than the row-major array.

Figure 3.4: Relative performance of 3D spatial convolution scaled by the performance in the row-major array.

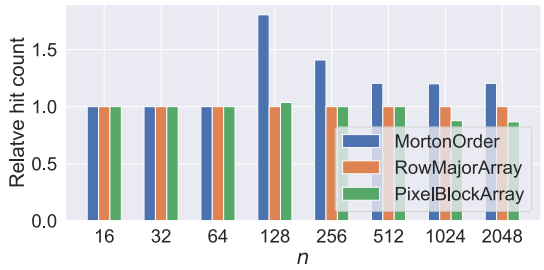
In the 3D case, we again see that Morton order and the block array perform ever so slightly worse than the row-major array for small images ($n \in \{16, 32\}$) and slightly better for bigger images ($n \geq 64$) when looking at relative hit count. The hit-to-miss ratio is about 60, explaining that the lower amount of misses which is generally about $0.7 - 0.8x$ that of the row-major array.

The reason we get these results is likely explained by what we saw in the plots for the summed linear distance in Figure 2.9. Around the corners, Morton order and the block array will perform significantly better than the row-major array. But the very large linear distances in Morton order and to some degree the block array when in the middle of the image is enough to overall drag them closer to row major's level – resulting in a cache performance difference that is worse for tiny n and small compared to the performance differences we will later see when using bigger n.

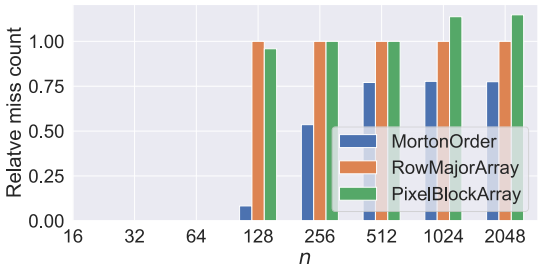
3.4 Fast Fourier Transform Results

A method for obtaining the results for different sizes of n similar to the one in Spatial Convolution has been used, where n here denotes the size of the $n \times n$ signal in 2D and $n \times n \times n$ signal in 3D consisting of complex numbers (the built-in `complex` type in Python has been used).

With this setup, the results in 2D in Figure 3.5 have been obtained:



(a) Relative L1 cache hit count scaled by the performance in the row-major array. A value above 1 means more hits than the row-major array.

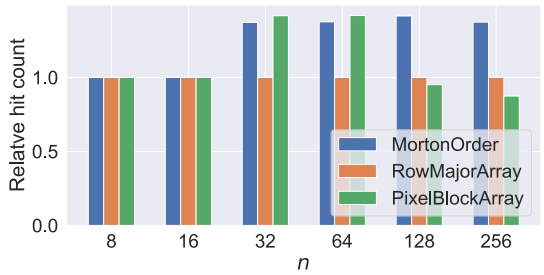


(b) Relative L1 cache miss count scaled by the performance in the row-major array. A value below 1 means fewer misses than the row-major array.

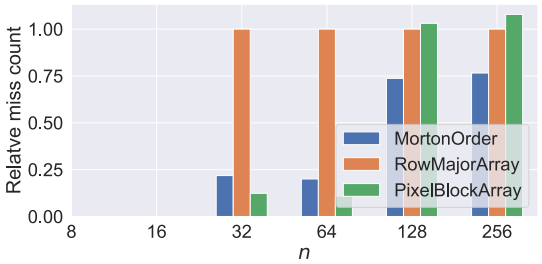
Figure 3.5: Relative performance of FFT on a 2D signal scaled by the performance in the row-major array.

We see that Morton order performs almost twice as good as the row-major arrays when the cache can't fit completely into memory anymore and from here on comes closer to the same performance. The block array in the start performs very similar to the row-major array and then starts performing worse for bigger n-values.

In 3D, the results as seen in Figure 3.6 have been obtained.



(a) Relative L1 cache hit count scaled by the performance in the row-major array. A value above 1 means more hits than the row-major array.



(b) Relative L1 cache miss count scaled by the performance in the row-major array. A value below 1 means fewer misses than the row-major array.

Figure 3.6: Relative performance of FFT on a 3D speed function f_{ijk} scaled by the performance in the row-major array.

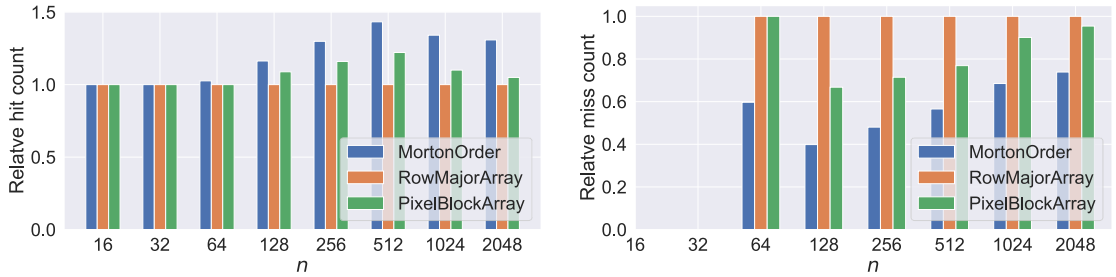
In the 3D case, we see that Morton order and the block array perform worse for small images, but better for bigger arrays. The amount of misses is about $0.7 - 0.8x$ the amount of a standard row-major array. It should be noted that the hit-to-miss count is about 1.12.

These results are likely due to the fact that performing the FFT with the Cooley-Tukey algorithm after the array indices have been bit-swapped is to a degree similar to performing first a row-major transversal, then a column-major transversal and so on for remaining dimensions. As discussed earlier, a row-major array will naturally perform better during a row-major transversal, but when having to perform all types of transversals it will overall perform worse than Morton order. The block array performs worst of all data structures, likely due to having all the faults of both the row-major array and the Morton order in this case (not excellent at either a row-major transversal nor a column-major transversal, etc).

3.5 Fast Marching Method Results

A method for obtaining the results for different sizes of n similar to the one in Spatial Convolution has been used, where n here denotes the size of the $n \times n$ speed function f_{ij} in 2D and the $n \times n \times n$ speed function f_{ijk} in 3D. Respectively $(0, 0)$ and $(0, 0, 0)$ have been used as starting points. To make the results reproducible, a seed has been used when generating the random values as the values influence the order in which the elements are accessed.

With this setup, the results in Figure 3.7 is obtained:



(a) Relative HIT count scaled by the performance in the row-major array. A value above 1 means more hits than the row-major array.

(b) Relative MISS count scaled by the performance in the row-major array. A value below 1 means fewer misses than the row-major array.

Figure 3.7: Relative performance of FMM on a 2D speed function f_{ij} scaled by the performance in the row-major array with $(0, 0)$ used as starting point.

We see that Morton order performs significantly better than the row-major arrays when the cache can't fit completely into memory anymore. The block array also performs a little better.

In 3D, the results as seen on Figure 3.8 have been obtained.

(a) Relative HIT count scaled by the performance in the row-major array. A value above 1 means more hits than the row-major array.

(b) Relative MISS count scaled by the performance in the row-major array. A value below 1 means fewer misses than the row-major array.

Figure 3.8: Relative performance of FMM on a 3D speed function f_{ijk} scaled by the performance in the row-major array with $(0, 0, 0)$ used as starting point..

In the 3D case, we see that Morton order and the block array perform worse for small images, but better for bigger arrays. The amount of misses is about $0.7x$ the amount of a standard row-major array for $n \geq 64$. It should be noted that the hit-to-miss ratio is about 1.3.

In these results, we again see that of course there is no difference when all the data fits into the L1 cache but afterward notable differences are observed. However, these results should be taken with a grain of salt due to two things: (1) the transversal of entries is dependent on the values and therefore different for different images and (2) the cache performance is highly dependent on what is chosen as the starting point. In these examples, we have chosen $(0, 0)$ or $(0, 0, 0)$ as the starting point. As shown in the plots for summed linear distance, Morton order and to some degree the block array performs comparatively much better when propagating out from a corner rather than from the middle of the image. If we were to choose a starting point closer to the middle, we would likely see different results where the performance between the data structures was more similar or maybe even favored the row-major array.

4 Discussion

4.1 Summary of Findings

Four algorithms have been tested on a cache simulator for different n-values in 2- and 3D and their results have been compared to a standard row-major array. The relative hit rates have been summarized in Table 4.1, the relative miss rates in Table 4.2 and the ratio between hits and misses in Table 4.3. For the matrix multiplication results, all numbers are based on choosing a tile size of 4 (which is the tile size that all three data structures performed the best for).

		2D						3D					
n		64	128	256	512	1024	2048	8	16	32	64	128	256
Matrix Multiplication	Morton	1.00	1.00	1.01	1.04	-	-	-	-	-	-	-	-
	Block array	1.00	1.00	1.01	1.04	-	-	-	-	-	-	-	-
Spatial Convolution	Morton	1.00	1.00	1.00	1.00	1.00	1.03	1.00	1.00	1.00	1.00	1.00	1.00
	Block array	1.00	1.00	1.00	1.00	1.00	1.03	1.00	1.00	1.00	1.01	1.01	1.00
Fast Fourier Transfrom	Morton	1.00	1.80	1.40	1.20	1.20	1.20	1.00	1.00	1.37	1.38	1.42	1.37
	Block array	1.00	1.04	1.00	1.00	0.88	0.87	1.00	1.00	1.42	1.42	0.95	0.87
Fast Marching Method	Morton	1.03	1.16	1.30	1.43	1.34	1.31	1.00	1.20	1.39	1.38	1.38	1.38
	Block array	1.00	1.09	1.16	1.22	1.10	1.05	1.00	1.11	1.13	1.05	1.02	1.02

Table 4.1: L1 cache hits scaled relative to a standard row-major array. A value greater than 1 means more cache hits than the row-major array.

		2D						3D					
n		64	128	256	512	1024	2048	8	16	32	64	128	256
Matrix Multiplication	Morton	0.78	0.53	0.30	0.09	-	-	-	-	-	-	-	-
	Block array	1.00	0.67	0.37	0.10	-	-	-	-	-	-	-	-
Spatial Convolution	Morton	1.00	1.05	1.08	1.09	1.10	0.55	nan	1.03	1.36	0.78	0.83	0.85
	Block array	1.00	1.00	1.12	1.12	1.12	0.56	nan	1.02	1.26	0.71	0.74	0.75
Fast Fourier Transfrom	Morton	nan	0.08	0.54	0.77	0.78	0.77	nan	nan	0.22	0.20	0.74	0.77
	Block array	nan	0.96	1.00	1.00	1.14	1.15	nan	nan	0.12	0.11	1.03	1.08
Fast Marching Method	Morton	0.60	0.40	0.48	0.57	0.68	0.73	nan	0.36	0.59	0.70	0.73	0.73
	Block array	1.00	0.67	0.71	0.77	0.90	0.95	nan	0.65	0.86	0.95	0.98	0.99

Table 4.2: L1 cache misses scaled relative to a standard row-major array. A value lower than 1 means fewer cache misses than the row-major array.

		2D						3D					
n		64	128	256	512	1024	2048	8	16	32	64	128	256
Matrix Multiplication	Morton	283.12	283.12	283.12	283.12	-	-	-	-	-	-	-	-
	Row-major	222.29	148.28	83.82	23.54	-	-	-	-	-	-	-	-
	Block array	222.29	222.29	222.29	222.29	-	-	-	-	-	-	-	-
Spatial Convolution	Morton	34.75	33.38	32.70	32.36	32.19	32.10	nan	91.66	73.71	66.89	63.84	62.39
	Row-major	34.75	35.13	35.31	35.41	35.45	17.24	nan	94.55	100.89	51.99	52.62	52.93
	Block array	34.75	35.13	31.45	31.42	31.43	31.44	nan	92.90	80.08	73.89	71.95	70.92
Fast Fourier Transfrom	Morton	nan	24.85	2.99	1.75	1.72	1.72	nan	nan	13.20	14.62	1.22	1.12
	Row-major	nan	1.14	1.14	1.12	1.12	1.11	nan	nan	2.10	2.13	0.63	0.63
	Block array	nan	1.23	1.14	1.12	0.86	0.83	nan	nan	24.09	27.70	0.58	0.51
Fast Marching Method	Morton	24.16	9.84	4.35	2.37	1.73	1.46	nan	9.90	2.39	1.55	1.36	1.31
	Row-major	14.06	3.37	1.61	0.94	0.88	0.83	nan	2.94	1.02	0.78	0.72	0.69
	Block array	14.06	5.50	2.61	1.49	1.08	0.91	nan	5.01	1.33	0.86	0.74	0.71

Table 4.3: L1 cache hit to miss ratio. I.e. a value of 10 would mean that for every 10 hits there is one miss.

For the recursive matrix multiplication, it is hard to notice a difference in the relative amount of hits, but this is due to the big hit-to-miss ratio. When looking at the relative miss rate, we see a significant difference for big n. This is likely due to the algorithm utilizing the properties of Morton order and the block array fully. For spatial convolution, we also see relative hit rates close to 1.00, again due to a large hit-to-miss ratio. The reason the performance is not even better for bigger n is likely due to what we saw in the plots for the summed linear distance in Figure 2.9. For FFT, significant improvements are observed, which can be explained by the fact that the algorithm performs row-major transversals followed by column-major transversals and so on. FMM is also shown to perform significantly better for Morton order and the block array, but the vast difference is likely because of the choice of the start point in the corner $(0, 0, 0)$.

While the results are promising, in practice the end-programmer has limited direct control of what data is kept in the cache. Chip manufactures are known to employ several clever tricks and keep secret about these to gain competitive advantages [3]. Some of these tricks might be based on the properties of the row-major array (since it is the standard) and have been fine-tuned over many years. In any case, we aren't guaranteed we would get the same results if we could measure the cache statistics directly. Furthermore, while computing Morton order and block array indices are fairly efficient, it still involves more expensive steps than for row-major arrays. And finally, while getting the data from memory definitely adds up a large part of the runtime required to perform a task, a proportion of the time used is also spent on raw computation on that data. How big this proportion is also matters for how big a performance gain we can expect from better data locality.

5 Conclusion

In this work, the theory of locality properties in modern computers has been summarized. Afterward, Morton order and block arrays have been presented and compared to row-major arrays which are standard for multidimensional arrays in most programming languages. It has been shown how to efficiently compute the indices for these data structures and various aspects have been discussed. Subsequently, the three different data structures have been compared.

As expected from the way the internal layout is structured, it was shown how Morton order has a flatter curve in the beginning than the row-major array when we propagate out from one of the optimal places which in this case was the point $(0, 0, 0)$. The same effect can also be shown for the block array, though to a lesser degree. However, when we use a starting point close to the middle of the image, the performance for Morton order will be much worse. When averaging over all points of an inner cube of an image and comparing the internal linear distance, Morton order (and to some degree the block array) is also shown to perform worse.

To test the data structures in practice, they have been implemented in Python with methods that allow easily simulating a cache. Also, four algorithms that are used in image processing and are interesting in a caching context have been implemented.

First, an algorithm for recursively performing matrix multiplication has been described and tested. The amount of L1 cache misses of Morton order and the block array was shown to be significantly lower than that of the row-major array. For spatial convolution, no significant differences between the different data structures were found. For the fast Fourier transform, a relative L1 cache hit ratio of about $1.2 - 1.4x$ was found for Morton order while it in some cases performed worse for the block array, likely because the FFT is somewhat equivalent to performing a combination of row-major, column-major, etc. transversals. At last, for the fast marching method relative L1 cache hit rates of about $1.3 - 1.4x$ were found. For almost all cases, Morton order outperformed the block array and when it didn't, the edge the block array had was insignificant.

All in all, it can be concluded that Morton order offers a promising runtime performance increase for common image processing tasks if it were to be efficiently implemented in a low-level language in commonly used software libraries such as Numpy, but for most algorithms, it would likely not yield an increase that is an order of magnitude better. Testing in a non-simulated environment would be needed to confirm this. Also, it was shown that the block array seldom performs better than the two other data structures presented and therefore is of limited interest for image processing tasks.

Bibliography

6

[1] Isaac Amidror. *Mastering the Discrete Fourier Transform in One, Two or Several Dimensions: Pitfalls and Artifacts*. Springer Publishing Company, Incorporated, 2013.

[2] Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer Publishing Company, Incorporated, 2012.

[3] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd. USA: Addison-Wesley Publishing Company, 2016.

[4] cod3monk. *pycachesim*. https://github.com/RRZE-HPC/pycachesim. Commit: 804bac0026. 2019.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[6] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Page 41, Table 2.9. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (visited on June 2, 2020).

[7] Asger Hoedt. *Morton Codes*. 2012. URL: http://asgerhoedt.dk/?p=276 (visited on Apr. 20, 2020).

[8] Zhexin Jiang, Hao Zhang, Yi Wang, and Seok-Bum Ko. „Retinal blood vessel segmentation using fully convolutional network with transfer learning". In: *Computerized Medical Imaging and Graphics* 68 (2018), pp. 1–15.

[9] Mark W. Jones, J. Andreas Baerentzen, and Milos Sramek. „3D Distance Fields: A Survey of Techniques and Applications". In: *IEEE Transactions on Visualization and Computer Graphics* 12.4 (July 2006), pp. 581–599.

[10] Ron Kimmel. „Fast Marching Methods for Computing Distance Maps and Shortest Paths". In: (1996). eprint: https://escholarship.org/uc/item/7kx079v5.

[11] Paul M. Mather. *Computer Processing of Remotely-Sensed Images: An Introduction*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004.

[12] Régis Monneau. „Introduction to the Fast Marching Method“. In: (Oct. 2010). eprint: https://hal.archives-ouvertes.fr/hal-00530910/document.

[13] Victor Adrian Prisacariu, Olaf Kähler, Ming-Ming Cheng, Carl Yuheng Ren, Julien P. C. Valentin, Philip H. S. Torr, Ian D. Reid, and David W. Murray. „A Framework for the Volumetric Integration of Depth Images“. In: *ArXiv* abs/1410.0925 (2014).

[14] Feifei Shen, Zhenjian Song, Congrui Wu, Jiaqi Geng, and Qingyun Wang. „Research on the fast Fourier transform of image based on GPU“. In: *CoRR* abs/1505.08019 (2015). arXiv: 1505.08019.

[15] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul Kelly. „An Exhaustive Evaluation of Row-Major, Column-Major and Morton Layouts for Large Two-dimensional Arrays“. In: (July 2003).

[16] David W Walker. „Morton ordering of 2D arrays for efficient access to hierarchical memory“. In: *The International Journal of High Performance Computing Applications* 32.1 (2018), pp. 189–203. eprint: https://doi.org/10.1177/1094342017725568.

[17] David Wise and Jeremy Frens. „Morton-order Matrices Deserve Compilers' Support Technical Report 533“. In: (1999).