Part I

AN INVITATION TO SPARSE DISTRIBUTED MEMORY: FROM THE THEORETICAL MODEL TO THE SYSTEM DYNAMICS

1 INTRODUCTION

Before I came across it, I had read many articles describing intricate properties of neurons, the many varieties of neurotransmitters and second-messenger molecules, synaptic modification processes, complex and hypercomplex cells, columns in the visual cortex, and so on. All of this was fascinating but seemed very biological and microscopic: altogether quite distant from everyday mental experiences. I simply had never come across a cogent theory that addressed brain mechanisms on a global level. Pentti Kanerva's memory model was a revelation for me: it was the very first piece of research I had ever come across that made me feel I could glimpse the distant goal of understanding how the brain works as a whole. It gave me a concrete sense for how familiar mental phenomena could be thought of as distributed patterns of micro-events, thanks to beautiful mathematics.

— Douglas Hofstadter

Sparse Distributed Memory (SDM) [15] is a mathematical model of long-term memory that has a number of neuroscientific and psychologically plausible dynamics. This model may be applied in all sort of applications because of its incredible ability to closely reflect the human capacity to remember past experiences from clues of the present. For instance, when one is walking on a dark alley and is afraid of something, one cannot explain where one's fear come from. One just feels it. We may interpret this situation as clues of the present — a dark alley; a giant metropolitan area; people going on about their lives mostly indifferent from each other; constrained routes ahead and behind you; a general lack of activity; that very feeling that something isn't right... without knowing what isn't right, or even what 'right' means... etc. — recalling past experiences from memory and thus generating the feeling. Our memory is able to make a parallel between previous experiences and the clues. Although one has never been in the exactly same situation, one's brain involuntarily makes an analogy and recognizes the possibility of danger. This flexibility into mapping one situation in another is an important human feature which is hard to replicate in computers.

SDM has been applied in many different fields, like pattern recognition [21, 25], noise reduction [19], handwriting recognition [10], robot automation [24, 18], and so forth. Linhares et al. [17] has showed that SDM respects the limits of short-term memory discussed by **?**] and **?**]. Despite all those applications, there is not a

reference implementation which would allow one to replicate the results published in a paper, to check the source code for details, and to improve it. Thus, even though intriguing results have been achieved using SDM, it requires counter-productive, duplicate, effort from researchers to build on top of previous work.

It is our belief that a tool such as a framework could bring orders of magnitude more researchers and attention if they were able to use the model, at zero cost, with an easy to use high-level language such as python, in an intuitive platform such as juypyter notebooks. Neuroscientists interested in long-term memory storage should not have to worry about high-bandwidth vector parallel computation. This new tool would provide a ready to use system in which experiments could be executed almost as soon as designed — and it might accelerate research [26].

Our motivation was our own effort to run our models. As there is no reference implementation, we had to implement our own and run several simulations to ensure that our implementation was correct and bug free. Thus, we had to deviate from our main goal — which was to test our hypothesis and explore the 'idea space' — and to focus in the implementation itself. Furthermore, new members in our research group had to go through different source codes developed by former members.

Extensions of SDM have been used in many applications. For example, Snaider and Franklin [27] extended SDM to efficiently store sequences of vectors and trees. Rao and Fuentes [24] used a modified SDM in an autonomous robot. Meng et al. [19] modified SDM to clean patterns from noisy inputs. Fan and Wang [10] extended SDM with genetic algorithms. Chada [5] extended SDM creating the Rotational Sparse Distributed Memory (RSDM), which is used to modeling network motifs, dynamic flexibility, and hierarchical organization, all results from neuroscience literature.

The main contribution of this work is a reference implementation which yields (i) orders of magnitude gains in performance, (ii) has several backends and operations, (iii) has been validated against the mathematical model, (iv) is cross-platform, and (v) is easily extended to test new research ideas. Our reference implementation may, hopefully, accelerate research into the model's dynamics and make it easier for readers to replicate any previous results and easily understand the source-code of the model. Moreover, it is compatible with jupyter notebook and researchers may share their notebooks possibly accelerating the advances in their fields [26].

Other contributions have also been introduced, which include (i) a noise filtering approach, (ii) a supervised classification algorithm, (iii) and a reinforcement learning algorithm, all of them using only the original SDM proposed by Kanerva, i.e., with no additional mechanisms, algorithms, data structures, etc. Although some of

these applications have already been explored in previous work [19, 10, 25], all of them have done some adapting of SDM to their problems, and none of them have used just the ideas introduced by Kanerva. We have presented different approaches with no adaptations whatsoever.

Finally, I have striven to provide a visual tour of the theory and application of SDM: whenever possible, detailed figures should tell the story — or at least do the heavy lifting. In this study, we will see an anomaly in one of Kanerva's predictions, which I believe is related to SDM capacity. We will see tests of a generalized reading operation proposed by Physics Professor Paulo Murilo (personal communication). We will see what happens when neurons — and all their information — is simply and suddenly lost. We will see whether information-theory can improve some of Kanerva's ideas. From (basic) noise filtering to learning to play tic-tac-toe, we will review the entirety of Dr. Pentti Kanerva's proposal.

This time, however, it will be running on a computer.

2 NOTATION

n	Number of dimensions, i.e., $n = 1,000$.
N	Size of the binary space, $\lvert\{0,1\}^n\rvert = 2^n$.
N'	Number of hard-locations samples from $\{0,1\}^n$. Its typical value is 1,000,000, as suggested by Kanerva [15].
H	Same as N'.
r	Access radius, i.e., when $n = 1,000$ and $N' = 1,000,000$, its typical value is 451. This value is calculated to activate, on average, one thousandth of N'.
η	A bitstring, usually a datum.
η_x	A clue x bits away from η, i.e., $\text{dist}(\eta, \eta_x) = x$.
ξ	A bitstring, usually an address.
$\text{dist}(x, y)$	Hamming distance between x and y.
$d(x, y)$	Same as $\text{dist}(x, y)$.

3

SPARSE DISTRIBUTED MEMORY

Sparse Distributed Memory (SDM) is a mathematical model developed and suggested as a theory of human memory by Finish Scientist Pentti Kanerva [15]. It introduces many interesting mathematical properties of n-dimensional binary space that, in a memory model, seem to be remarkably psychologically plausible. Most notable among these are the tip-of-the-tongue phenomenon, conformity to Miller's magic number [17] and robustness against loss of neurons.

The data — and address space on which it is stored — are represented by large sequences of bits, called *bitstrings*. The *Hamming distance* provides comparisons between bitstrings and is used as a metric for the system. The Hamming distance is defined for two bitstrings of equal length as the number of positions in which bits differ. For example, 00110_b and 01100_b are bitstrings of length 5 and their Hamming distance is 2.

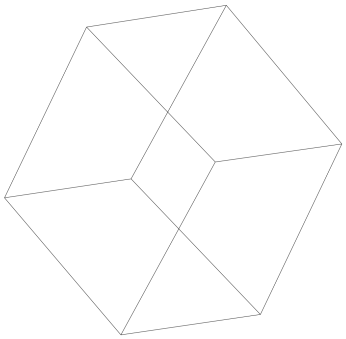
The space studied by Kanerva is also called the *hypercube graph*, or Q_n, as in Figure 1. For a fixed $n \in \mathbb{Z}$, the graph $G = (V, E)$, in which $v \in V$ iff there is a bijective function $b : V \to \{0,1\}^n$ and $(v_i, v_j) \in E$ iff $H(b(v_i), b(v_j)) = 1$, where H is the Hamming distance. That is, n-sized bitstrings correspond to nodes, and edges exist between nodes iff they flip a single bit. Though Kanerva has derived many combinatorial properties of the space, additional results have been found by the graph-theoretical community. A good survey is provided by Harary et al. [12].

One has to be careful when thinking intuitively about distance in SDM because the Hamming distance does not have the same properties of, say, our 3-dimensional space.

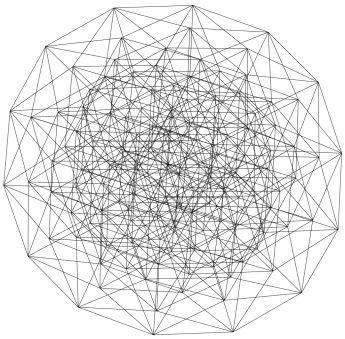
Though both follow the triangle inequality ($d(A, B) \leqslant d(A, C) + d(B, C)$), which in 3-d Euclidean distance may be loosely interpreted as "if A is close to B, and B is close to C, then A is also close to C" — $d(A, B) \leqslant r$ and $d(B, C) \leqslant r \Rightarrow d(A, C) \leqslant 2r$ —, but in SDM, although the inequality is also valid, two bitstrings would be close when, for instance, $r = 430$, so $2r = 860$ would cover all other bitstrings. Hence, it makes no sense to say that A is also close to C.

There are numerous, beautiful, counter-intuitive notions involved in this space. This difference in intuition may trick even experienced researchers when analyzing some situations.

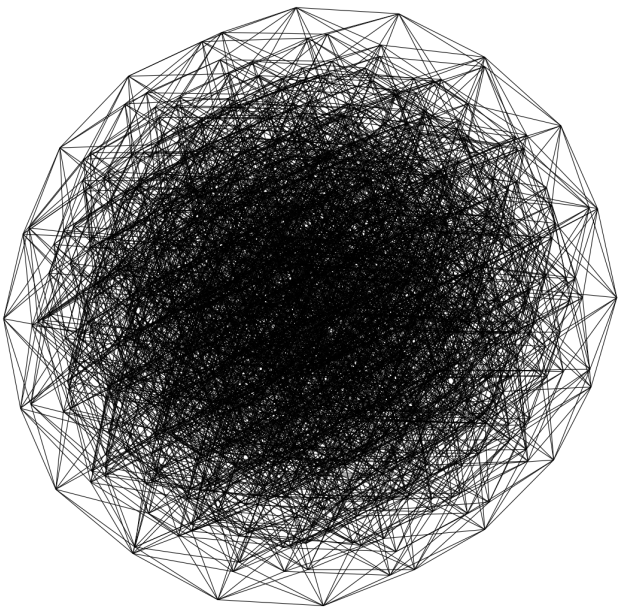
Unlike traditional memory used by computers, SDM performs read and write operations in a multitude of addresses, also called neurons. That is, the data is not written, or it is not read in a single address



(a) Q_3



(b) Q_7



(c) Q_{10}

Figure 1: Here we have Q_n, for $n \in \{3, 7, 10\}$. Each node corresponds to a bitstring in $\{0, 1\}^n$, and two nodes are linked iff the bitstrings differ by a single dimension. A number of observations can be made here. First, the number of nodes grows as $O(2^n)$; which makes the space rapidly intractable. Another interesting observation, better seen in the figures below, is that most of the space lies 'at the center', at a distance of around $n/2$ from any given vantage point.

spot, but in many addresses. These are called activated addresses, or activated neurons.

The activation of addresses takes place according to their distances from the datum. Suppose one is writing datum η at address ξ, then all addresses inside a circle with center ξ and radius r are activated. So, η will be stored in all these activated addresses, which are around address ξ, such as in Figure 2. An address ξ' is inside the circle if its hamming distance to the center ξ is less than or equal to the radius r, i.e. $distance(\xi, \xi') \leqslant r$.

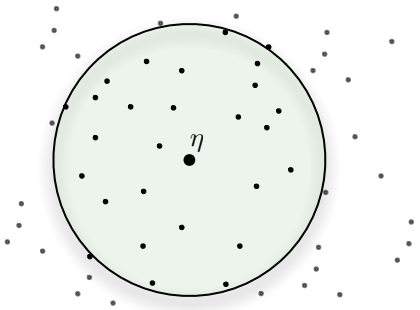


Figure 2: Activated addresses inside access radius r around center address.

Every time write or read in SDM memory activates a number of addresses with close distance. The data is written in these activated addresses or read from them. These issues will be addressed in due detail further on, but a major difference from a traditional computer memory is that the data are always stored and retrieved in a multitude of addresses. This way SDM memory has robustness against loss of addresses (e.g., death of a neuron).

In traditional memory, each datum is stored in an address and every look up of a specific datum requires a search through the memory. In spite of computer scientists having developed beautiful algorithms to perform fast searches, almost all of them do a precise search. That is, if you have an imprecise clue of what you need, these algorithms will simply fail.

In SDM, the data space is the same as the address space, which amounts to a vectorial, binary space, that is, a $\{0,1\}^n$ space. This way, the addresses where the data will be written are the same as the data themselves. For example, the datum $\eta = 00101_b \in \{0,1\}^5$ will be written to the address $\xi = \eta = 00101_b$. If one chooses a radius of 1, the SDM will activate all addresses one bit away or less from the center address. So, the datum 00101_b will be written to the addresses 00101_b, 10101_b, 01101_b, 00001_b, 00111_b, and 00100_b.

In this case, when one needs to retrieve the data, one could have an imprecise cue at most one bit away from η, since all addresses one bit away have η stored in themselves. Extending this train of thought

for larger dimensions and radius, exponential numbers of addresses are activated and one can see why SDM is a distributed memory.

When reading a cue η_x that is x bits away of η, the cue shares many addresses with η. The number of shared addresses decreases as the cue's distance to η increases, in other words, as x increases. This is shown in Figure 3. The target datum η was written in all shared addresses, thus they will bias the read output in the direction of η. If the cue is sufficiently near the target datum η, the read output will be closer to η than η_x was. Repeating the read operation increasingly gets results closer to η, until it is exactly the same. So, it may be necessary to perform more than one read operation in order to converge to the target data η.

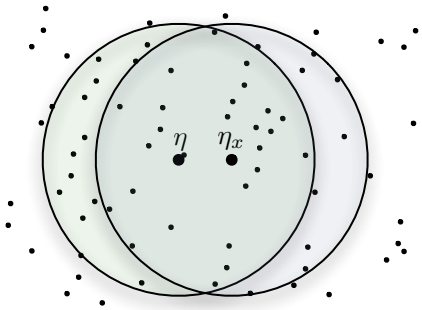


Figure 3: Shared addresses between the target datum η and the cue η_x.

The addresses of the $\{0,1\}^n$ space grows exponentially with the number of dimensions n, i.e. $N = 2^n$. For $n = 100$ we have $N \approx 10^{30}$, which is incredibly large when related to a computer memory. Furthermore, Kanerva [15] suggests n between 100 and 10,000. Recently he has postulated 10,000 as a desirable minimum N (personal communication). To solve the feasibility problem of implementing this memory, Kanerva made a random sample of $\{0,1\}^n$, in his work, having N' elements. All these addresses in the sample are called hard-locations. Other elements of $\{0,1\}^n$, not in N', are called virtual neurons. This is represented in Figure 4. All properties of read and write operations presented before remain valid, but limited to hard-locations. Kanerva suggests taking a sample of about one million hard-locations.

Using this sample of binary space, our data space does not exist completely. That is, the binary space has 2^n addresses, but the memory is far away from having these addresses available. In fact, only a fraction of this vectorial space is actually instantiated. Following Kanerva's suggestion of one million hard-locations, for $n = 100$, only $100 \cdot 10^6/2^{100} = 7 \cdot 10^{-23}$ percent of the whole space exists, and for $n = 1,000$ only $100 \cdot 10^6/2^{1000} = 7 \cdot 10^{-294}$ percent.

Kanerva also suggests the selection of a radius that will activate, on average, one one thousandth of the sample, which is 1,000 hard-locations for a sample of one million addresses. In order to achieve his suggestion, a 1,000-dimension memory uses an access radius $r = 451$, and a 256-dimensional memory, $r = 103$. We think that a 256-dimensional memory may be important because it presents conformity to Miller's magic number [17].

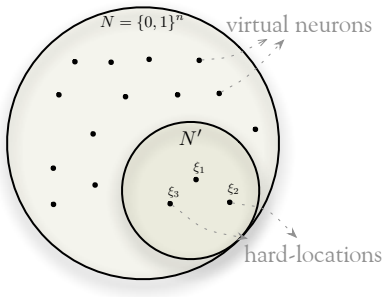


Figure 4: Hard-locations randomly sampled from binary space.

Since a cue η_x near the target bitstring η shares many hard-locations with η, SDM can retrieve data from imprecise cues. Despite this feature, it is very important to know how imprecise this cue could be while still giving accurate results. What is the maximum distance from our cue to the original data that still retrieves the right answer? An interesting approach is to perform a read operation with a cue η_x, that is x bits away from the target η. Then measure the distance from the read output and η. If this distance is smaller than x we are converging. Convergence is simple to handle, just read again and again, until it converges to the target η. If this distance is greater than x we are diverging. Finally, if this distance equals x we are in a tip-of-the-tongue process. A tip-of-the-tongue psychologically happens when you know that you know, but you can't say what exactly it is. In SDM mathematical model, a tip-of-the-tongue process takes infinite time to converge. Kanerva [15] called this x distance, where the read's output averages x, the critical distance. Intuitively, it is the distance from which smaller distances converge and greater distances diverge. In Figure 5, the circle has radius equal to the critical distance and every η_x inside the circle should converge. The figure also shows a convergence in four readings.

The $\{0,1\}^n$ space has $N = 2^n$ locations from which we instantiate N' samples. Each location in our sample is called a hard-location. On these hard-locations we do operations of read and write. One of the insights of SDM is exactly the way we read and write: using data as addresses in a distributed fashion. Each datum η is written in every activated hard-location inside the access radius centered on

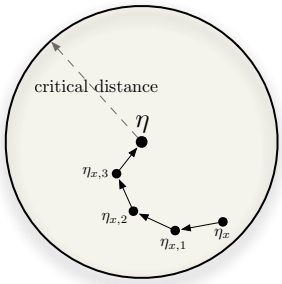


Figure 5: In this example, four iterative readings were required to converge from η_x to η.

η	0	1	1	0	1	0	0
ξ_{before}	6	-3	12	-1	0	2	4
	⇓ -1	⇓ +1	⇓ +1	⇓ -1	⇓ +1	⇓ -1	⇓ -1
ξ_{after}	**5**	**-2**	**13**	**-2**	**1**	**1**	**3**

Table 1: Write operation example in a 7-dimensional memory of data η being written to ξ, one of the activated addresses.

the address, that equals datum, $\xi = \eta$. Kanerva suggested using an access radius r having about one one thousandth of N'. As an imprecise cue η_x shares hard-locations with the target bitstring η, it is possible to retrieve η correctly. (Actually, probably more than one read is necessary to retrieve exactly η.). Moreover, if some neurons are lost, only a fraction of the datum is lost and it is possible that the memory can still retrieve the right datum.

A random bitstring is generated with equal probability of 0's and 1's in each bit. One can readily see that the average distance between two random bitstrings has binomial distribution with mean $n/2$ and standard deviation $\sqrt{n/4}$. For a large n, most of the space lies close to the mean and has fewer shared hard-locations. As two bitstrings with distance far from $n/2$ are very improbable, Kanerva [15] defined that two bitstrings are orthogonal when their distance is $n/2$.

The write operation needs to store, for each dimension bit which happened more (0's or 1's). This way, each hard-location has n counters, one for each dimension. The counter is incremented for each bit 1 and decremented for each bit 0. Thus, if the counter is positive, there have been more 1's than 0's, if the counter is negative, there have been more 0's than 1's, and if the counter is zero, there have been an equal number of 1's and 0's. Table 1 shows an example of a write operation being performed in a 7-dimensional memory.

The read is performed polling each activated hard-location and statistically choosing the most written bit for each dimension. It consists of adding all n counters from the activated hard-locations

and, for each bit, choosing bit 1 if the counter is positive, choose bit 0 if the counter if negative, and randomly choose bit 0 or 1 if the counter is zero.

3.1 NEURONS AS POINTERS

One interesting view is that neurons in SDM work like pointers. As we write bitstrings in memory, the hard-locations' counters are updated and some bits are flipped. Thus, the activated hard-locations do not necessarily point individually to the bitstring that activated it, but together they point correctly. In other words, the read operation depends on many hard-locations to be successful. This effect is represented in Figure 6: where all hard-locations inside the circle are activated and they, individually, do not point to η. But, like vectors, adding them up points to η. If another datum ν is written into the memory near η, the shared hard-locations will have information from both of them and would not point to either. All hard-locations outside of the circle are also pointing somewhere (possibly other data points). This is not shown, however, in order to keep the picture clean and easily understandable.

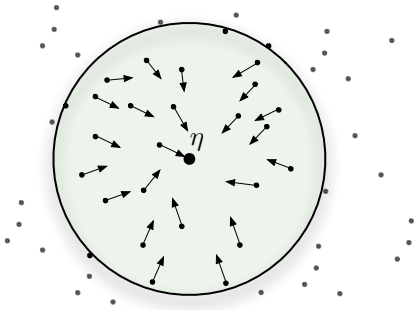


Figure 6: Hard-locations pointing, approximately, to the target bitstring.

3.2 CONCEPTS

Although Kanerva does not mention concepts directly in his book [15], the author's interpretation is that each bitstring may be mapped to a concept. Thus, unrelated concepts are orthogonal and concepts could be linked through a bitstring near both of them. For example, "beauty" and "woman" have distance $n/2$, but a bitstring that means "beautiful woman" could have distance $n/4$ to both of them. As a bitstring with distance $n/4$ is very improbable, it is linking those concepts together. Linhares et al. [17] approached this concept via "chunking through averaging".

Due to the distribution of hard-locations between two random bitstrings, the vast majority of concepts is orthogonal to all others. Consider a non-scientific survey during a cognitive science seminar, where students asked to mention ideas unrelated to the course brought up terms like birthdays, boots, dinosaurs, fever, executive order, x-rays, and so on. Not only are the items unrelated to cognitive science, the topic of the seminar, but they are also unrelated to each other.

For any two memory items, one can readily find a stream of thought relating two such items ("Darwin gave dinosaurs the boot"; "she ran a fever on her birthday"; "isn't it time for the Supreme Court to x-ray that executive order?", ... and so forth). Robert French presents an intriguing example in which one suddenly creates a representation linking the otherwise unrelated concepts of "coffee cups" and "old elephants" [11].

This mapping from concepts to bitstrings brings us two main questions: (i) Suppose we have a bitstring that is linking two major concepts. How do we know which concepts are linked together? (ii) From a concept bitstring how can we list all concepts that are somehow linked to it? This second question is called the problem of spreading activation.

3.3 READ OPERATION

In his work, Kanerva proposed and analyzed a read algorithm called here Kanerva's read. His read takes all activated hard-locations counters and sum them. The resulting bitstring has bit 1 where the result is positive, bit 0 where the result is negative, and a random bit where the result is zero. In a word, each bit is chosen according to all written bitstrings in all hard-locations, being equal to the bit more appeared. Table 2a shows an example of Kanerva's read result bitstring.

Daniel Chada, one member of our research group, proposed another way to read in SDM, in this work called Chada's read. Instead of summing all hard-location counters, each hard-location evaluates its resulting bitstring individually. Then, all resulting bitstrings are summed again, and the same rule as Kanerva applies. Table 2b shows an example of Chada's read result bitstring. The counter's values are normalized to 1, for positive ones, or -1, for negative ones, and the original values are the same as in Table 2a.

The main difference between Kanerva's read and Chada's is that, in the former, a hard-location that has more bitstrings written has a greater weight in the decision of each bit. In the latter, all hard-locations have the same weight, because they can contribute to the sum with only one bitstring.

It is important to say that Chada's read came from Anwar and Franklin [1] which gave a misguided description of the read operation. The original description is the following:

> With our datum distributively stored, the next question is how to retrieve it. With this in mind, let us ask first how one reads from a single hard location, x. Compute ζ, the bit vector read at x, by assigning its ith bit the value 1 or 0 according as the ith counter at x is positive or negative. Thus, each bit of ζ results from a majority rule decision of all the data that have been written at x. [...] Knowing how to read from a hard location allows us to read from any of the 2^{1000} arbitrary locations. Suppose ζ is any location. The bit vector, ξ, to be read at ζ, [...] Put another way, pool the bit vectors read from hard locations accessible from ζ, and let each of their ith bits vote on the ith bit of ξ.
> — Anwar and Franklin [1, p.342]

This fact just highlihts how important it is to have a reference implementation that one may read the code to clarify one's understanding about the details of each operation.

3.3.1 *Generalized read operation*

A member of my Master's committee, Prof. Paulo Murilo[1], has proposed a generalized reading operation (personal communication), which covers both Kanerva's and Chada's read — and opens a new venue of potential discoveries. He proposed summing all hard-location counters raised to the power of z while holding the original sign of the counter (positive or negative). Thus, Kanerva's read would be the same as $z = 1$, while Chada's would be the same as $z = 0$. Hence, we will here explore how SDM would behave with other values of z, such as 0.5, 2, and 3. Mathematically, let A be the set of the counters of the activated hardlocation, and c_i be the counter of the i-th bit. Then,

$$s_i = \sum_{c \in A} \frac{c_i}{|c_i|} |c_i|^z$$

Finally, the i-th bit of the resulting bitstring is 1 if $s_i > 0$, or 0 if $s_i < 0$, or random if $s_i = 0$. Notice that when $z = 1$, then $s_i = \sum_{c \in A} c_i$, which is the Kanerva's read; and when $z = 0$, then $s_i = \frac{c_i}{|c_i|} = \text{sign}(c_i)$, which is the Chada's read.

1 Universidade Federal Fluminense's Physics Professor Paulo Murilo

ξ_1	-2	12	4	0	-3
ξ_2	-5	-4	2	8	-2
ξ_3	-1	0	-1	-2	-1
ξ_4	3	2	-1	3	1
$\sum$	**-5**	**10**	**4**	**3**	**-5**
	⇓	⇓	⇓	⇓	⇓
	0	1	1	1	0

(a) Kanerva's read example

ξ_1	-1	1	1	1	-3
ξ_2	-1	-1	1	1	-1
ξ_3	-1	1	-1	-1	-1
ξ_4	1	1	-1	-1	1
$\sum$	**-2**	**1**	**0**	**0**	**-2**
	⇓	⇓	⇓	⇓	⇓
	0	1	1	1	0

(b) Chada's read example

Table 2: Comparison of Kanerva's read and Chada's read. Each ξ_i is an activated hard-location and the values come from their counters. Gray cells' value is obtained randomly with probability 50%.

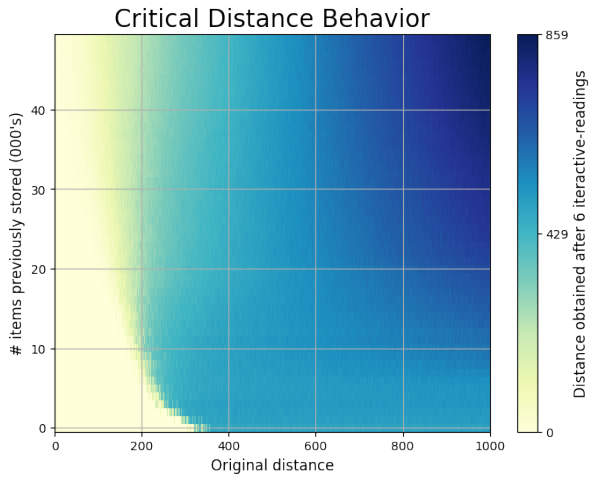
3.4 CRITICAL DISTANCE

Kanerva describes the critical distance as the threshold of convergence of a sequence of read words. It is "the distance beyond which divergence is more likely than convergence"[15]. Furthermore, Kanerva explains that "a very good estimate of the critical distance can be obtained by finding the distance at which the arithmetic mean of the new distance to the target equals the old distance to the target"[15]. In other words, the critical distance can be equated as the edge to our memory, the limit of human recollection.

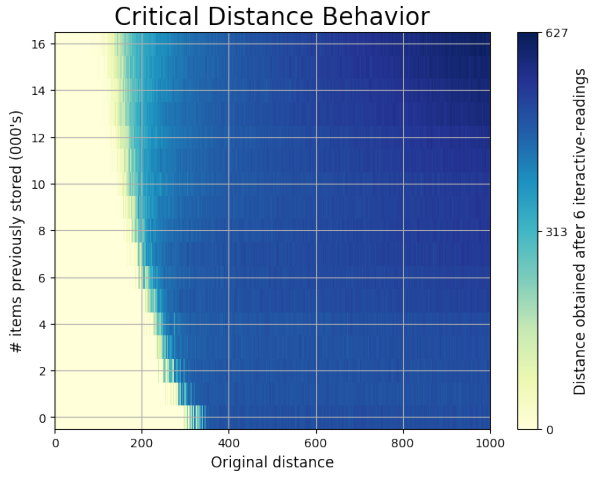
In his book, Kanerva analyzed a specific situation with $n = 1000$ ($N = 2^{1000}$), 1 million hard-locations $N' = 1,000,000$, an access-radius of 451 (within 1,000 hard-locations in each circle) and 10 thousand writes of random bitstrings in the memory. As computer resources were very poor those days, Kanerva couldn't make a more generic analysis.

Starting from the premise of SDM as a faithful model of human short-term memory, a better understanding of the critical distance may shed light on our understanding of the thresholds that bind our own memory.

Figure 7 compares the critical distance behavior under different scenarios. This replicates our previous results [2, 4] and is a first part of the process of framework validation, to which we throw our attention next.



(a) Kanerva's original model



(b) Chada's read

Figure 7: How far, in hamming distance, is a read item from the original stored item? Kanerva demonstrated that, after a small number of iterative readings (6 here), a critical distance behavior emerges. Items read at close distance converge rapidly; whereas farther items do not converge. Most striking is the point in which the system displays the tip-of-tongue behavior. Described by psychological moments when some features of the item are prominent in one's thoughts, yet the item still cannot be recalled (but an additional cue makes convergence 'immediate'). Mathematically, this is the precise distance in which, despite having a relatively high number of cues (correct bits) about the desired item, the time to convergence is infinite. Heatmap colors display the hamming distance the associative memory is able to cleanly converge to—or not. In the x-axis, the distance from the desired item is displayed. In the y-axis, we display the read operation's behavior as the number of items registered in the memory grows. These graphs are computing intensive, yet they can be easily tested by readers in our provided jupyter notebooks. Note the different scales.

4

FRAMEWORK ARCHITECTURE

The framework implements the basic operations in a Sparse Distributed Memory which may be used to create more complex operations. It is developed in C language and the OpenCL parallel framework — which may be loaded in many platforms and programming languages — with a wrapper in Python. The Python module makes it easy to create and execute simulations in a Sparse Distributed Memory and works properly in Jupyter Notebook [16]. It works in both Python 2 and Python 3.

We split the SDM memory in two parts: the hard-location addresses and the hard-location counters. Thus, the addresses (bitstrings) of the hard-locations are stored in one array, while their counters in another. This makes possible to create multiple SDMs using the same address space, which would save computational effort to scan a bitstring in all the SDMs — since they share the same address space, the activated hard-locations will be the same in all of them. As the slowest part of reading and writing operations is scanning the address space, the performance benefits are significant.

Each part may be stored either in the RAM memory or in a file. The RAM memory is interesting for quick experiments, automated tests, and others scenarios in which the SDM may be lost, while the file is interesting for a long-term SDM, like creating an SDM file with 10,000 random writes, which will be copied over and over to run multiple experiments. The file may also be sent to another researcher or may be published within the paper to let others run their own checks and verify the results. In summary, the framework fits many different uses and necessities.

Let a SDM memory with N dimensions and H hard-locations. Then, in a 64-bit computer, the array of hard-location addresses will use $H \cdot 8 \cdot \lceil N/64 \rceil$ bytes of memory, and there will be $H \cdot N$ hard-location counters. For example, in a SDM memory with 1,000 dimensions and 1,000,000 hard-locations, using 32-bit integers for the counters, the array of addresses will use 122MB of memory and the counters will use 3.8 GB of memory.

Basic operations were grouped in four sets: (i) for bitstrings, (ii) for addresses, (iii) for counters, and (iv) for memories (SDMs). Operations include creating new bitstrings, flipping bits, generating a bitstring with a specific distance from a given bitstring, scanning the address space using different algorithms, writing a bitstring to a counter, writing in an SDM, reading from an SDM, and iteratively reading from an SDM until convergence.

4.1 BITSTRING

Bitstrings are the main structure of SDM. The addresses are represented in bitstrings, as well as the data. A bitstring is stored as an array of integers. Each integer may be 16-bit, 32-bit, or 64-bit long, depending on the configuration. By default, each integer is 64-bit long.

For instance, a 1,000-bit bitstring will have $\lceil 1000/64 \rceil = 16$ integers. These integers will have a total of $16 \cdot 64 = 1,024$ bits. The remaining 24 bits are always zero, so they do not affect the result of any operation. The memory usage efficiency is $1 - 24/1024 = 97.65\%$. Bitstrings store neither how many bits they have nor the array length. These pieces of information are only stored in the address space.

4.1.1 *The distance between two bitstrings*

The distance between two bitstrings is calculated by the hamming distance, which is the number of different bits between them. It is calculated counting the number of ones in the exclusive or (xor) between the bitstrings, i.e., $d(x, y) =$ number of ones in $x \oplus y$.

There are several algorithms to calculate the number of ones [29], but the performance depends on the processor. So, we have implemented three different algorithms and one may be selected through compiling flags. The default algorithm is to use a built-in __popcnt() instruction from the compiler.

There is also the naive algorithm, which really counts the number of ones checking bit by bit. It is available only to testing purposes and should never be used.

The other algorithm available is the lookup. It pre-calculates a table with the number of ones of all possible 16-bit integers. This table is accessed a few times to calculate the number of ones of a 64-bit integer, i.e., to calculate the distance between two bitstrings, it sums the distance of each 16-bit part of the bitstrings, i.e., $d(x[0:63], y[0:63]) = d(x[0:15], y[0:15]) + d(x[16:31], y[16:31]) + d(x[32:47], y[32:47]) + d(x[48:63], y[48:63])$ where $x[i:i+15]$ and $y[i, i+15]$ are the 16-bit integers formed by the bits between i and $i+15$ of x and y, respectivelly. Each 16-bit distance is calculated through a single table access. As each distance is calculated in O(1), this algorithm runs in $O(\lceil bits/16 \rceil)$. This table uses 65MB of RAM. One may change the table from 16-bit integers to 32-bit integers, which would halve the number of accesses at the expense of 4GB of RAM (instead of 65MB).

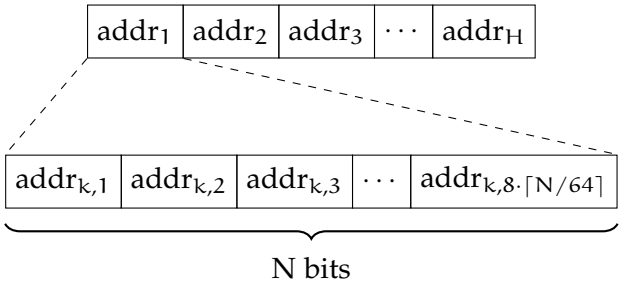


Figure 8: Address space's bitstrings are stored in a contiguous array. In a 64-bit computer, each bitstring is stored in a sub-array of 64-bit integers, with length $8 \cdot \lceil N/64 \rceil$.

4.2 ADDRESS SPACE

An address space is a fixed collection of bitstrings, and each bitstring represents a hard-location address. They store the number of bitstrings, as well as the number of bits, number of integers per bitstring, and the number of remaining bits.

Bitstrings are stored in a contiguous array of 64-bit integers, as shown in Figure 8. Hence, basic pointer arithmetic provides us with performance improvements in their access, as processors realize fetches of contiguous chunks of memory [23].

The scan for activated hard-locations is performed in an address space. It returns the indexes of the bitstrings which were inside the circle (and their distances). Then, each operation uses these pieces of information in a different way.

4.2.1 *Scanning for activated hard-locations*

Scanning for the activated hard-locations is a problem similar to well-known problems in computational geometry called "range reporting in higher dimensions". In this case, none of the known algorithms is able to solve our problem faster than $O(H)$. The algorithm which seems to best fit in our problem consumes $O(H)$ space and runs in $O(\log^n(H))$ [7], which is really slower than $O(H)$ when, for instance, $H = 1,000,000$ and $n = 1,000$. For a review of the range reporting algorithms, see Chan et al. [6].

In 2014, there was published a solution to fast search in hamming space which seems applicable to our problem Norouzi et al. [22]. It provides a fast search when $r/n < 0.11$ or $r/n < 0.06$, where r is the radius and n is the number of bits. But, in our case, for a 1,000 bits SDM, $r/n = 0.451$, which changes the runtime to $O(H^{0.993})$. This is really close to $O(H)$, but with a larger constant. Unfortunately, $O(H)$ is still faster.

It is intriguing that none of those algorithms is able to solve our scanning problem. The idea behind those computational geometry

algorithms is roughly to split the search space in half each step, which would take $O(\log(H))$ to go through the whole space. But this approach does not work because of the high number of dimensions (i.e., 1,000) and because the hard-locations' addresses are randomly sampled from the $\{0,1\}^n$ space. Although each addresses' bit itself splits the hardlocations in half, it does not split the search space in half since both halves still must be convered by the algorithm. For instance, let's say we have $n = 1,000$ dimensions with $H = 1,000,000$ hard-locations, and we are scanning within a circle with radius $r = 451$, then after checking the first bit we have two cases: (i) for the half with the same first bit, we must keep scanning with radius 451; and (ii) for the half with a different first bit, we must keep scanning with radius 450. Hence, the search space has not been split in half because both halves have been covered (and one of them should have been skipped).

Finally, as our best approach is to scan through all hard-locations, we may distribute the scan into many tasks which will be executed independently. The tasks may be executed in different processes, threads, or even computers. They may also run in the CPU or in the GPU. In this case, we may take into account both the time required to distribute the tasks and the time to receive their results.

The framework implements three main scanner algorithms: linear scanner, thread scanner, and OpenCL scanner. The linear scanner runs in a single core, is the slowest one, and was developed only for testing purposes; the thread scanner runs at the CPU in multiple threads sharing memory (and our recommendation is to use the number of threads equals to twice the number of CPU cores); and the OpenCL scanner runs in multiple GPU cores and support multiple devices. The speed of a scan depends on the CPU and GPU devices, thus the best approach to choose which scanner is best for one's setup is to run a benchmark.

The OpenCL must be initialized, which just copies the address space's bitstrings to the GPU's memory. Then, many scans may be executed with no necessity to upload the bitstrings again. The OpenCL scanner supports running into multiple devices.

4.2.2 *OpenCL kernel*

There are 8 OpenCL kernels which explore differently the GPU architecture to improve performance. It is necessary because there are several GPU microarchitecture and a single kernel will never be optimal for all of them. In simplified form, OpenCL splits the tasks into workgroups which, in turn, split their part of the task into workers. The works are like threads in a computer. OpenCL specifies four levels of memory hierarchy for the GPUs: global memory, read-only memory, local memory, and private memory. The global

memory and read-only memory are accessible by all workgroups, while each workgroup has its own local memory, accessible by its workers. Finally, each worker has its own private memory. The number of workers per workgroup is defined by user and must be multiple of the number of tasks.

All 8 kernels do the same thing: calculate the exclusive OR (XOR) between two 64-bit integers and count the number of bits one in the result. They just do it with different approaches. For instance, `single_scan0` calculates one distance between bitstrings per worker (Listing 1); while `single_scan2` uses a whole workgroup to calculate each distance, distributing each element of the 64-int integer array per worker (Listing 3).

The OpenCL kernels `single_scan3` (Listing 4), `single_scan4` (Listing 5), `single_scan5` (Listing 6), `single_scan5_unroll` (Listing 7), `single_scan6` (Listing 8) explore the GPU architecture to improve the sum of the partial distances. Each workgroup calculates the distance of several bitstrings. During the distance calculation, each worker calculates the exclusive OR (XOR) between two 64-bit integers and use the built-in popcount function to count the number of ones. Then, they update an array of partial distances with their results. This array is stored in the local memory and is shared between all workers of the same workgroup. This whole step happens simultaneously in the GPU. Then, a reduction algorithm is used to sum the partial distances array in order to calculate the total distance. This reduction algorithm is also distributed between the workers and runs in $O(\log_2(\text{bs\_step}))$. Finally, the first worker of each workgroup checks whether the distance is less than or equal to the radius to include the bitstring index into the resulting array.

Some of the optimizations may not work in some GPUs, because not all their premises are valid. Before choosing a kernel, one should check whether it works property for one's specific GPU device.

4.3 COUNTERS

Each hard-location has one integer of data per bit. For instance, each hard-location of a 1,000 bits SDM has 1,000 bits. Those integers are stored in a counter.

A counter is an array of integers which stores the data of all hard-locations. So, the counter's array has $n \cdot H$ integers.

When two counters are added in a third counter, there may occur an overflow. It is not supposed to be a problem because, by default, each counter is a signed 32-bit integer that can store any number between -2,147,483,648 and 2,147,483,647, which means they will not overflow with less writes than $2^{31} - 1$ divided by the average number of activated hard-locations. For instance, when $n = 1,000$, $H = 1,000,000$, and $r = 451$, the average number of activated

```
__kernel
void single_scan0(
                __constant const uchar *bitcount_table,
                __global const ulong *bitstrings,
                const uint bs_len,
                const uint sample,
                __constant const ulong *bs,
                const uint radius,
                __global uint *counter,
                __global uint *selected,
                __local uint *partial_dist)
{
        uint id = get_global_id(0);

        if (id < sample) {
                ulong a;
                uint dist;

                const __global ulong *row = bitstrings + id*bs_len;

                dist = 0;
                for(uint j=0; j<bs_len; j++) {
                        a = row[j] ^ bs[j];
                        dist += popcount(a);
                }
                if (dist <= radius) {
                        selected[atomic_inc(counter)] = id;
                }
        }
}
```

Listing 1: OpenCL kernel `single_scan0`. It calculates one distance per worker and let the GPU decide how to distribute this task between workgroups and workers. It is the most simple kernel and does not explore any details of the GPU architecture.

```
__kernel
void single_scan1(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint id;
    ulong a;
    uint dist;
    const __global ulong *row;

    for (id=get_global_id(0); id < sample; id += get_global_size(0)) {

        row = bitstrings + id*bs_len;

        dist = 0;
        for(uint j=0; j<bs_len; j++) {
            a = row[j] ^ bs[j];
            dist += popcount(a);
        }
        if (dist <= radius) {
            selected[atomic_inc(counter)] = id;
        }

    }
}
```

Listing 2: OpenCL kernel `single_scan1`. It is just like `single_scan0`, but it distribute several distances per workgroup, which, in turn, distribute the distances among their workers.

```
__kernel
void single_scan2(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint dist;
    ulong a;
    uint j;

    for (uint id = get_group_id(0); id < sample; id += get_num_groups(0)) {

        const __global ulong *row = bitstrings + id*bs_len;

        dist = 0;
        j = get_local_id(0);
        if (j < bs_len) {
            a = row[j] ^ bs[j];
            dist += popcount(a);
        }
        partial_dist[get_local_id(0)] = dist;

        barrier(CLK_LOCAL_MEM_FENCE);

        if (get_local_id(0) == 0) {
            dist = 0;
            for(uint t = 0; t < bs_len; t++) {
                dist += partial_dist[t];
            }
            if (dist <= radius) {
                selected[atomic_inc(counter)] = id;
            }
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 3: OpenCL kernel `single_scan2`. It calculates one distance per workgroup, distributing each 64-bit integer operation per worker, and then summing the results obtained by the workers. The sum algorith is done by only the first worker of each workgroup.

```
__kernel
void single_scan3(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint dist;
    ulong a;
    uint j;

    for (uint id = get_group_id(0); id < sample; id += get_num_groups(0)) {

        const __global ulong *row = bitstrings + id*bs_len;

        dist = 0;
        j = get_local_id(0);
        if (j < bs_len) {
            a = row[j] ^ bs[j];
            dist = popcount(a);
        }
        partial_dist[get_local_id(0)] = dist;

        // Parallel reduction to sum all partial_dist array.
        for(uint stride = get_local_size(0)/2; stride > 0; stride /= 2) {
            barrier(CLK_LOCAL_MEM_FENCE);
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] +=
                    partial_dist[get_local_id(0) + stride];
            }
        }

        if (get_local_id(0) == 0) {
            if (partial_dist[0] <= radius) {
                selected[atomic_inc(counter)] = id;
            }
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 4: OpenCL kernel `single_scan3`. It calculates one distance per workgroup, distributing each 64-bit integer operation per worker, and then summing the results obtained by the workers. The sum algorith is a parallel reduction, in which the workers split the array in two parts and sum the second part in the first part every loop. So, the sum is calculated in $O(\log_2(\text{number of workers per workgroup}))$. This kernel only works when the number of workers per workgroup is a power-of-2.

```
__kernel
void single_scan4(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint dist;
    ulong a;
    uint j;

    for (uint id = get_group_id(0); id < sample; id += get_num_groups(0)) {

        const __global ulong *row = bitstrings + id*bs_len;

        dist = 0;
        j = get_local_id(0);
        if (j < bs_len) {
            a = row[j] ^ bs[j];
            dist = popcount(a);
        }
        partial_dist[get_local_id(0)] = dist;

        uint old_stride = get_local_size(0);
        __local uint extra;
        extra = 0;
        for(uint stride = get_local_size(0)/2; stride > 0; stride /= 2) {
            barrier(CLK_LOCAL_MEM_FENCE);
            if ((old_stride&1) == 1 && get_local_id(0) == old_stride-1) {
                extra += partial_dist[get_local_id(0)];
            }
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] +=
                    partial_dist[get_local_id(0) + stride];
            }
            old_stride = stride;
        }

        if (get_local_id(0) == 0) {
            if (partial_dist[0] + extra <= radius) {
                selected[atomic_inc(counter)] = id;
            }
        }

        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 5: OpenCL kernel `single_scan4`. This kernel is just like `single_scan3`, but it works with any number of workers per workgroup. The tradeoff is that it includes an aditional step in the parallel reduction algorithm.

```
__kernel
void single_scan5(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint dist;
    ulong a;
    uint j;

    for (uint id = get_group_id(0); id < sample; id += get_num_groups(0)) {
        const __global ulong *row = bitstrings + id*bs_len;

        dist = 0;
        j = get_local_id(0);
        if (j < bs_len) {
            a = row[j] ^ bs[j];
            dist = popcount(a);
        }
        partial_dist[get_local_id(0)] = dist;

        uint stride;
        for(stride = get_local_size(0)/2; stride > 32; stride /= 2) {
            barrier(CLK_LOCAL_MEM_FENCE);
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] +=
                    partial_dist[get_local_id(0) + stride];
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        for(/**/; stride > 0; stride /= 2) {
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] +=
                    partial_dist[get_local_id(0) + stride];
            }
        }

        if (get_local_id(0) == 0) {
            if (partial_dist[0] <= radius) {
                selected[atomic_inc(counter)] = id;
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 6: OpenCL kernel `single_scan5`. This kernel is just like `single_scan3`, but it explores one more detail of many GPU microarchitecture: the size of the warp. As the workers in the same warp are always synchronized, there is no need to synchronize them using a barrier. This specific kernel only works when the number of workers per workgroup is a power-of-2.

```
__kernel
void single_scan5_unroll(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint dist;
    ulong a;
    uint j;

    for (uint id = get_group_id(0); id < sample; id += get_num_groups(0)) {
        const __global ulong *row = bitstrings + id*bs_len;

        dist = 0;
        j = get_local_id(0);
        if (j < bs_len) {
            a = row[j] ^ bs[j];
            dist = popcount(a);
        }
        partial_dist[get_local_id(0)] = dist;

        for(uint stride = get_local_size(0)/2; stride > 32; stride /= 2) {
            barrier(CLK_LOCAL_MEM_FENCE);
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] += partial_dist[get_local_id(0) + stride];
            }
        }

        // We do not need to sync because they all run in the same warp.
        if (get_local_id(0) < 32 && get_local_size(0) >= 64) {
            partial_dist[get_local_id(0)] += partial_dist[get_local_id(0) + 32];
        }
        if (get_local_id(0) < 16 && get_local_size(0) >= 32) {
            partial_dist[get_local_id(0)] += partial_dist[get_local_id(0) + 16];
        }
        if (get_local_id(0) < 8 && get_local_size(0) >= 16) {
            partial_dist[get_local_id(0)] += partial_dist[get_local_id(0) + 8];
        }
        if (get_local_id(0) < 4 && get_local_size(0) >= 8) {
            partial_dist[get_local_id(0)] += partial_dist[get_local_id(0) + 4];
        }
        if (get_local_id(0) < 2 && get_local_size(0) >= 4) {
            partial_dist[get_local_id(0)] += partial_dist[get_local_id(0) + 2];
        }

        if (get_local_id(0) == 0) {
            partial_dist[0] += partial_dist[1];
            if (partial_dist[0] <= radius) {
                selected[atomic_inc(counter)] = id;
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 7: OpenCL kernel `single_scan5_unroll`. This kernel is exactly like `single_scan5`, but it unrolls the last for since it has at most 5 loops.

```
__kernel
void single_scan6(
        __constant const uchar *bitcount_table,
        __global const ulong *bitstrings,
        const uint bs_len,
        const uint sample,
        __constant const ulong *bs,
        const uint radius,
        __global uint *counter,
        __global uint *selected,
        __local uint *partial_dist)
{
    uint dist;
    ulong a;
    uint j;

    for (uint id = get_group_id(0); id < sample; id += get_num_groups(0)) {
        const __global ulong *row = bitstrings + id*bs_len;

        dist = 0;
        j = get_local_id(0);
        if (j < bs_len) {
            a = row[j] ^ bs[j];
            dist = popcount(a);
        }
        partial_dist[get_local_id(0)] = dist;

        uint old_stride = get_local_size(0);
        uint stride;
        __local uint extra;
        extra = 0;
        for(stride = get_local_size(0)/2; stride > 32; stride /= 2) {
            barrier(CLK_LOCAL_MEM_FENCE);
            if ((old_stride&1) == 1 && get_local_id(0) == old_stride-1) {
                extra += partial_dist[get_local_id(0)];
            }
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] +=
                    partial_dist[get_local_id(0) + stride];
            }
            old_stride = stride;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        for(/**/; stride > 0; stride /= 2) {
            if ((old_stride&1) == 1 && get_local_id(0) == old_stride-1) {
                extra += partial_dist[get_local_id(0)];
            }
            if (get_local_id(0) < stride) {
                partial_dist[get_local_id(0)] +=
                    partial_dist[get_local_id(0) + stride];
            }
            old_stride = stride;
        }

        if (get_local_id(0) == 0) {
            if (partial_dist[0] + extra <= radius) {
                selected[atomic_inc(counter)] = id;
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 8: OpenCL kernel `single_scan6`. This kernel is just like `single_scan5`, but it works with any number of workers per work. The tradeoff is an aditional step in the parallel reduction algorithm.

hard-locations is 1,000 and it would require at least one million writes before being possible to a counter to overflow. Note also that it would be more likely to saturate the memory before any overflow.

Anyway, counters may have overflow protection depending on compiling options. By default, there is no overflow check for performance reasons (and because it does not seem necessary).

4.4 READ AND WRITE OPERATIONS

The reading and writing operations are executed in two steps: first, the address space is swept looking for the activated addresses; then, the operation is performed in the counters. Reading operation assemblies the bitstring according to the counters of the activated addresses, while the writing operation changes the counters.

The iterated reading keeps reading until it gets exactly the same bitstring (or the number of maximum interations has been reached), i.e., it performs $\eta_{i+1} = \text{read}(\eta_i)$ and stops when $\eta_{k+1} = \eta_k$. If the initial bitstring is inside the critical distance of η, it will converge to η, but, if it is not, it will diverge and reach the maximum number of iterations.

The framework has both Kanerva's read and Murilo's generalized read. The generalization brings a parameter z, which is the exponent. In this case, the results are floating point instead of integer, which considerably reduces performance. When $z = 1$, it is exactly as the Kanerva's read. When $z = 0$, it is the Chada's read. We also explored how SDM would behave for different values of z.

There is another special read operation: the weighted reading. In the weighted reading, the value of the counters are multiplied by a weight which depends only on the distance between the reading address and the hard-location address. The weight is retrieved from a lookup table of integers indexed by the distance. The rest of the read operation is exactly the same.

There is also a weighted writing operation. In this case, the weight is applied when the counters are updated, i.e., if the weight is 2, the counters are increased twice when bits are 1, and decreased twice when bits are 0. Just as in the weighted reading, the weights depend only on the distance between the writing address and the hard-location address. The weights are retrieved from a lookup table of integers indexed by the distance.

5 RESULTS (I): FRAMEWORK VALIDATION

The framework has been validated comparing its results with the expected results from Kanerva [15]. Thus, we run simulations which were then compared to the theoretical analysis conducted some decades ago.

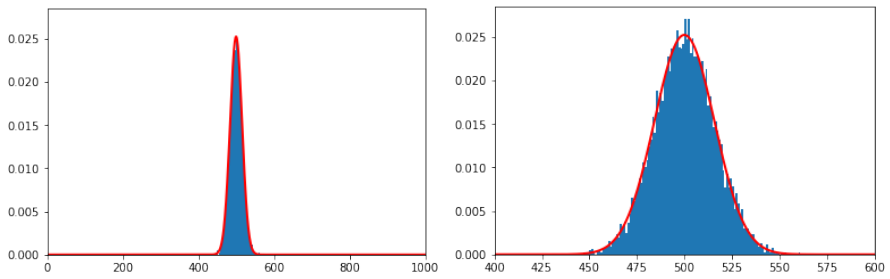
5.1 DISTANCE BETWEEN RANDOM BITSTRINGS

As showed by Kanerva [15], the distance between two bitstrings follows a binomial distribution with mean $\mu = n/2$ and standard deviation $\sigma = \sqrt{n}/2$. For large values of n, it may be approximated by a normal distribution with the same mean and standard deviation.

In order to validate our random bitstring generation algorithm, we have calculated 10,000 distances between two random bitstrings with $n = 1,000$ bits. In total, 20,000 random bitstrings have been generated during the simulation. The code is available in the "Distance between bitstrings" notebook [3].

In figure 9, we can notice that the theoretical model and the simulation matches. Hence, it seems the random bitstring generation algorithm works properly.

This also validates the algorithm used to calculate the distance between two bitstrings. In this simulation, we have used the built-in __popcnt() function.



(a) Full histogram (b) Zoom in the interval $[400, 600]$

Figure 9: Histogram of 10,000 distances between two random bitstrings with 1,000 bits. The curve in red is the theoretical normal distribution with $\mu = 500$ and $\sigma = \sqrt{500}/2$.

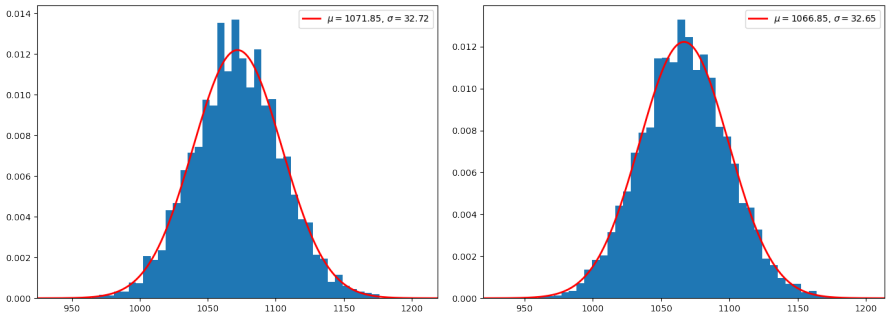
5.2 NUMBER OF ACTIVATED HARD-LOCATIONS

In his seminal work, Kanerva proposed to use a sample of 1,000,000 hard-locations in a 1,000 bits SDM. He also proposed to activate only 1,000 of them, on average. He calculated that an access radius of $r = 451$ would activate, on average, 0.00107185004892 of the whole space, or, in this case, 1,071.85 hard-locations.

We extended his results, calculating the distribution of the number of activated hard-locations. As each hard-location has probability $p = 0.00107185004892$ of being activated, the probability of activating exactly a out of H hard-locations follows a binomial distribution with mean $\mu = pH$ and standard deviation $\sigma = \sqrt{Hp(p-1)}$. In this case, $\mu = 1071.85$ and $\sigma = 32.72$.

In order to validate our scan algorithm, we have run 10,000 scans from a random bitstring and counted the number of activated hard-locations. The code is available in the "Number of activated hard-locations" notebook [3].

In figure 10, we can notice that the theoretical model and the simulation matches. Hence, it seems that both the address space generation algotihm and the scan algorithm work properly. Notice that the curve is almost the same for $n = 1,000$ and $n = 256$. It happens because the access radius is adjust to have p as close as possible to 0.001. They are not exactly the same because their p differs a little.



(a) $n = 1,000$, $H = 1,000,000$, $r = 451$, and $p = 0.00107185$

(b) $n = 256$, $H = 1,000,000$, $r = 103$, and $p = 0.00106684$

Figure 10: Histogram of the number of activated hard-locations in 10,000 scans from a random bitstring. The curve in red is the theoretical normal distribution with $\mu = Hp$ and $\sigma = p(p-1)H$.

Besides the number of activated hard-locations, we have also extended Kanerva's results to calculate the distribution of distances between the center of the circle and the activated hard-locations. Let A be the set of activated hard-locations, ξ be the center of the circle, and r be the access radius, then:

$$P(d(a,\xi) = x | a \in A) = \frac{P(d(a,\xi) = x)}{P(a \in A)} \tag{1}$$

$$= \frac{\binom{n}{x}}{\sum_{k=0}^{r} \binom{n}{k}} \tag{2}$$

In order to check Equation 2, we have calculated the distances of the activated hard-locations to the center of 1,000 random circles. The code is available in the "Distances of activated hard-locations" notebook [3].

In figure 11, we can notice that the theoretical model and the simulation matches.

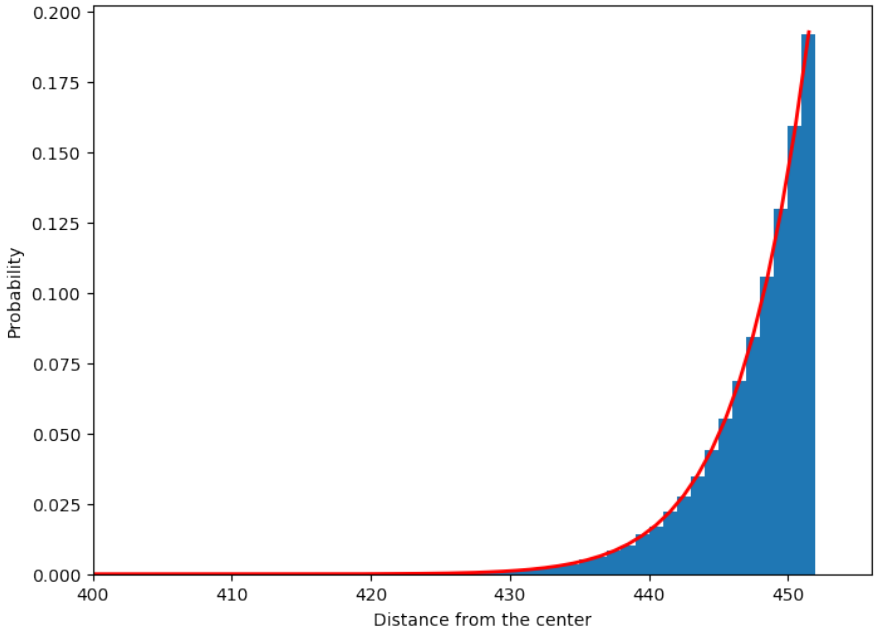


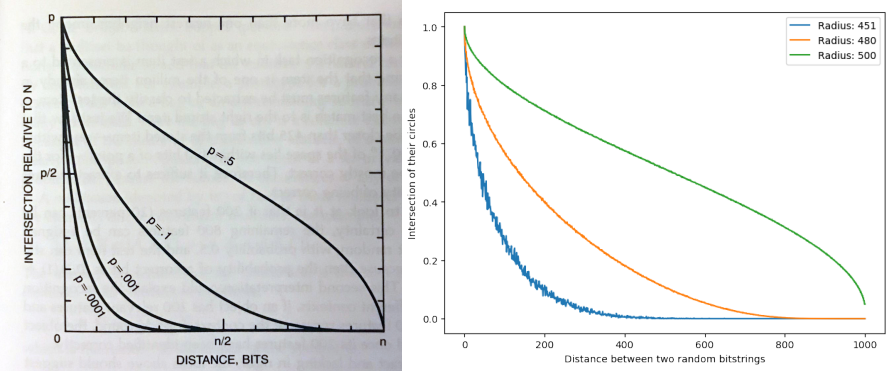
Figure 11: Histogram of the distances of activated hard-locations to the center of the circles. The curve in red is the theoretical distribution of Equation 2

5.3 INTERSECTION OF TWO CIRCLES

Kanerva has calculated the intersection of two circles according to the distance between their centers. The intersection is important to understand how SDM works, because it affects directly the critical distance. When η_d is inside the critical distance, then it will converge to η. In fact, it converges because they share a sufficient amount of hard-locations, i.e., the intersection of the circle around η_d and η is enough to converge. For further information about the relation between the critical distance and the intersection, see Brogliato et al. [4].

We have calculated the intersection between a random bitstring (bs1) and another bitstrings (bs2) exactly d bits away. The former (bs1) is just a random bitstring. The latter (bs2) was generated randomly flipping d bits of bs1. The code is available in the "Kanerva's Figure 1.2" notebook [3].

In Figure 12, we can notice that we have obtained the same results as Kanerva. It seems that the random flipping bits algorithm and the scan algorithm work properly.



(a) Kanerva [15, Figure 1.2, p.25] (b) Generated by SDM-Frameworkm with $n = 1,000$

Figure 12: Number of hard-locations in the intersection of circles around two bitstrings x bits away.

5.4 STORAGE AND RETRIEVAL OF SEQUENCES

Kanerva [15, Ch.8] presented an approach to store and retrieve sequences using k different SDMs, namely sdm_1, sdm_2, dots, sdm_k.

Let $a_0, a_1, a_2, \ldots, a_n$ be a sequence to be stored in a k-fold memory. So, all pointers of the form $a_i \rightarrow a_{i+k}$ will be written to sdm_k memory, i.e., in sdm_1, the following pointers will be written: $a_0 \rightarrow a_1$, $a_1 \rightarrow a_2$, $\ldots$, $a_{n-1} \rightarrow a_n$; while in sdm_2, the following pointers will be written: $a_0 \rightarrow a_2$, $a_2 \rightarrow a_3$, $\ldots$, $a_{n-2} \rightarrow a_n$; and so forth.

We have tested exactly the same example presented in Kanerva [15], p.85. We wrote two sequences to a 3-fold memory: $< A, B, C, D >$ and $< E, B, C, F >$. Then, after reading the sequences $< A, B, C >$ and $< E, B, C >$, we have obtained D and F, respectively.

Each reading operation was performed summing the counters of all activated hard-locations from all three memories. For instance, to read the sequence $< A, B, C >$, we have activated the hard-locations around C in sdm_1, we have also activated the hard-locations around D in sdm_2, and, finally, we have also activated the hard-locations around A in sdm_3. After summing the counters of all those hard-locations, we evaluate the resulting bitstring just as in the original read operation.

The code is available in the "Sequences (Kanerva Ch 8)" notebook [3].

The logic behind how it works is that, when reading the sequence $< A, B, C >$, we have A pointing to D, while both B and C point to D and F. Thus, D appears more often than F and ended up being the result.

Hence, as we have replicated the theoretical results from Kanerva, we have one more evidence that our framework works properly.

5.4.1 k-*fold memory using only one SDM*

We have extended Kanerva's ideas to be able to store and retrieve sequences in k-fold memories using only one SDM (instead of k SDMs).

Our idea was to create k random bitstrings, one for each fold. We have performed writing and reading exactly as Kanerva's original idea, but, instead of writing to sdm_k, we have written a_{i+k} into the address $a_i \oplus tag_k$, and, instead of reading from sdm_k, we have read from address $a_i \oplus tag_k$, where $\oplus$ is the exclusive or (XOR) operator.

It worked as if we had splitted SDM into k regions with low intersection between two of them. So, as the interference is minimal, they work like independent SDMs. The major disadvantage of this approach is that memory capacity may be reached faster.

Splitting the memory into regions may be an interesting strategy to other sorts of problems, mostly the ones which would need many SDMs and, consequently, would use a lot of RAM.

6

RESULTS (II): CRITICAL DISTANCE

One particular analysis of Kanerva's interest is given by the limits of recovery. That is, given an item read at a distance x from a previously stored η, does this reading at a η_x recover the original? Suppose an SDM is trying to read an item written at η, but the cues received so far lead to a point of distance x from η. As one reads at η_x, a new bitstring β is obtained, leading to Kanerva's question: what is the new distance from η to β? Is it smaller or larger than x? That, of course, depends on the ratio between x and the number of dimensions of the memory.

Kanerva [15, p.70] originally predicted a ~500-bit distance after a point (Figure 13). The original prediction considered that the read distance would decline when inside the critical distance and increase afterwards, converging to a ~500-bit distance. At this point, each read would lead to a different, orthogonal, ~500-bit distance bitstring. He analyzed specifically an SDM with 1,000 bits and 10,000 random bitstrings written into it.

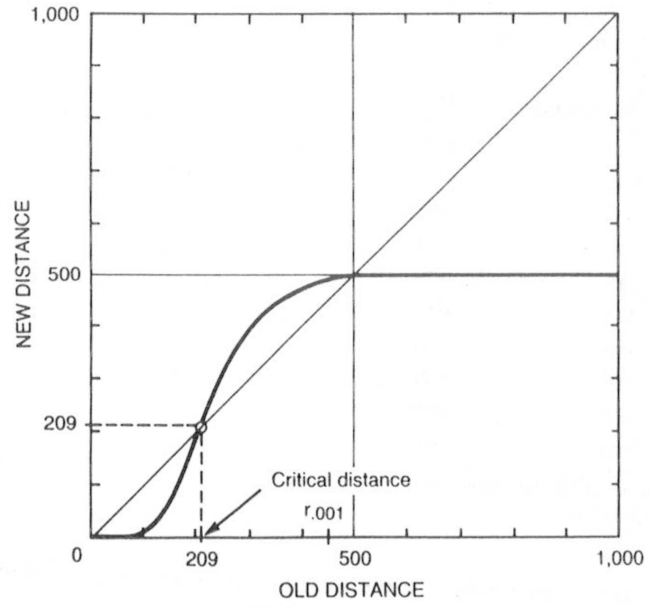
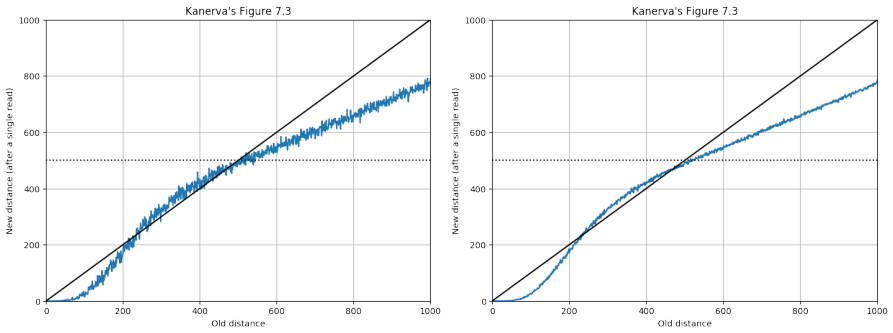


Figure 7.3
New distance to target as a function of old distance.

Figure 13: Kanerva's original Figure 7.3 (p. 70) predicting a ~500-bit distance after a point.

As we ran the simulations, this one in particular struck our attention: The new distances obtained after a read operation were not perfectly predicted by the theoretical model. We have strictly followed Kanerva's configuration and, even so, we have found out some deviations from Kanerva's original theoretical analysis and the results obtained by simulation.

In details, we have created a SDM with $n = 1,000$, $H = 1,000,000$, and $r = 451$. Then, we have generated 10,000 random bitstrings and written them into the memory. Then, we have generated a reference bitstring (bs_ref) and written it into the memory. Then, we have executed the following steps with x from 0 to 1,000: (i) copy bs_ref into a new bitstring; (ii) randomly flipped x bits of the copy; (iii) read from the memory in the copy address; and (iv) stored the distance between the returned bitstring and bs_ref. Finally, we have plotted Figure 14.



(a) 1 sample for each distance x (b) 6 samples for each distance x

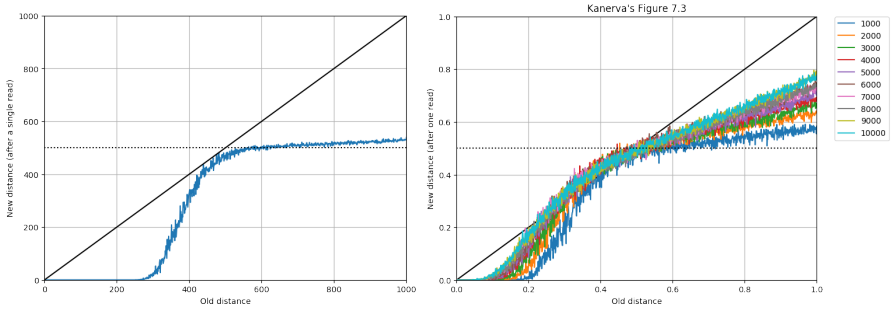
Figure 14: Results generated by the framework diverging from Kanerva's original Table 7.2. Here we had a 1,000 bit, 1,000,000 hard-location SDM with exactly 10,000 random bitstrings written into it, which was also Kanerva's configuration.

Figure 14a has a lot of noise because we have read only once for each distance x and Kanerva has predicted the average distance. So, we have changed the steps to run k reads and store the average new distance. We run with $k = 6$, and the results can be seen in Figure 14b, which has much lower noise and still holds the divergence.

Our results show that the theoretical prediction is not accurate. There are interaction effects from one or more of the attractors created by the 10,000 writes, and these attractors seem to raise the distance beyond ~500 bits (Figure 14).

Obviously, these small deviations from Kanerva's original theoretical predictions deserve a qualification. Kanerva was working in the 1980s and the 1990s, and had no access to the immense computational power that we do today. It is no surprise that some small interaction effects should exist as machines allow us to explore the ideas of his monumental work.

But, when we reduced the number of random bitstrings written in the SDM from 10,000 to only 100, the results reflected very well the Kanerva's theoretical expectation (Figure 15a). This result strengthens our hypothesis that the disparities in the computational results are due to the interaction effect of high numbers of different attractors. In Figure 15b we can notice that, the more random bitstrings are written, the stronger the attractors.



(a) 100 writes (b) Steps of 1,000 writes

Figure 15: Results generated by the framework similar from Kanerva's original Table 7.2. Here we have a 1,000 bit, 1,000,000 hard-location SDM with (a) exactly 100 random bitstrings written into it and (b) steps of 1,000 random bitstrings written into it.

To obtain the results from Figures 14 and 15, we had to write 10,000 random bitstrings to an SDM, and then randomly choose one of those bitstrings to be our origin. Finally, we randomly flipped some bits from the origin bitstring and executed a reading operation in the SDM. Thereby, in order to show the interaction effects more clearly, we changed the single read for an 15-iterative read. As we can see in Figure 16, after a distance of 500 bits, all bitstrings converged to 500-bit distance bitstrings, just as described by Kanerva.

Hence, our understanding is that the attractors are just preventing the bitstrings to converge directly to 500-bit distance bitstrings, requiring more reading steps to do so. They are in other orthogonal bitstrings' critical distance, but sufficiently far not to converge in a single read.

Going further in the analysis, we calculated the probability of missing a bit when reading from SDM. After all, that's how Kanerva has originally found the curve. To do this, we used the following equations from our previous work [4]. Let d be the distance to the target, h be the number of hard-locations activated during read and write operations, s be the number of total stored bitstrings, H be the number of total hard-locations, w be the number of times the target was written into SDM, θ be the total random bitstrings in all h hard-locations activated by read operation, and $\phi(d)$ be the average number of shared hard-locations activated two bitstrings d bits away.

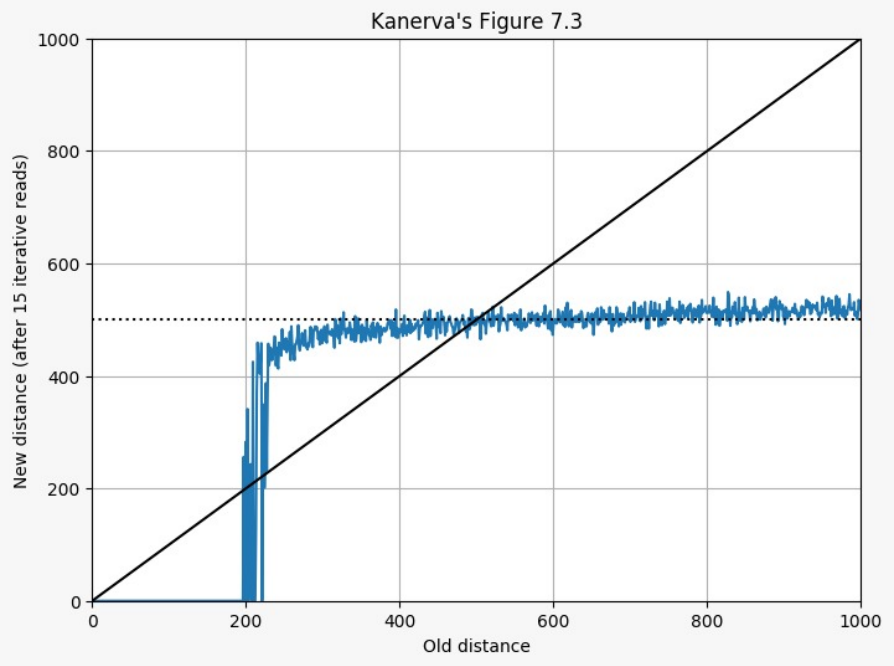


Figure 16: This graph shows the interaction effects more clearly. As we change the single read to a 6-iterative read, the effect has vanished and all bitstrings above $x = 500$ have converged to 500-bit distance bitstrings. Here we have the exact same configuration of Figure 14, except for the iterative read.

$$\theta = \frac{sh^2}{H} - w \cdot \phi(d) \tag{3}$$

$$P(\text{miss}|\text{bit} = 0) = 1 - P\left(\sum_{i=1}^{\theta} X_i < \frac{sh^2}{2H}\right) \tag{4}$$

$$P(\text{miss}|\text{bit} = 1) = P\left(\sum_{i=1}^{\theta} X_i < \frac{sh^2}{2H} - w \cdot \phi(d)\right) \tag{5}$$

$$P(\text{miss}) = \frac{1}{2} \cdot [P(\text{miss}|\text{bit} = 0) + P(\text{miss}|\text{bit} = 1)] \tag{6}$$

For details and the proof of this equation, see Brogliato et al. [4]. Although Kanerva has found a formula for $\phi(d)$ through an unsolved integral, and de Pádua Braga and Aleksander [9] have proposed another way to calculate $\phi(d)$, we have used our framework to estimate the values of d. In order to do that, we used a Monte Carlo approach, generating many pair of random bitstrings d bits away from them and calculating the average number of shared hard-locations between them. The code is available in the "Calculate critical distance" notebook [3].

Kanerva's settings according to the parameters of the equation were: $s = 10,000$, $H = 1,000,000$, and $w = 1$. We have calculated $\phi(d)$ as explained, and $h = H \cdot 2^{-n} \sum_{i=0}^{r} \binom{n}{i}$, where $n = 1,000$ and

$r = 451$. Finally, $h = 1,071.85$ and changing d from 0 to 1000, we got Figure 17.

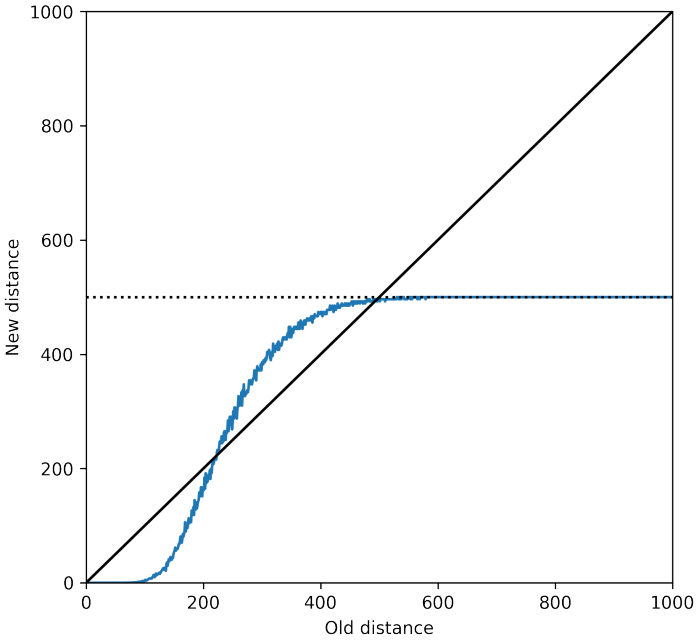


Figure 17: Kanerva's original Figure 7.3 generated using the equations from Brogliato et al. [4].

As we can easily notice, we got exactly the same curve as Kanerva. This question has been intriguing us since then and we are still looking for a more analytic explanation then merely interference from the other written attractors.

6.1 CRITICAL DISTANCE OF 209

The critical distance is defined as d where $P(miss) = d/n$, or, in Figure 17, the point where the curve meets with the identity function (the black diagonal line). Thus, we plot a zoom-in of Figure 17 around $d = 209$ in Figure 18 using the same equations [4]. It was surprising that the meeting does not happen at $d = 209$, but around $d = 221$.

To confirm that the critical distance is not around 209, but around 221, we also plot a zoom-in of Figure 14 around $d = 209$ in Figure 19. In order to reduce the noise, we increased the samples to $k = 180$.

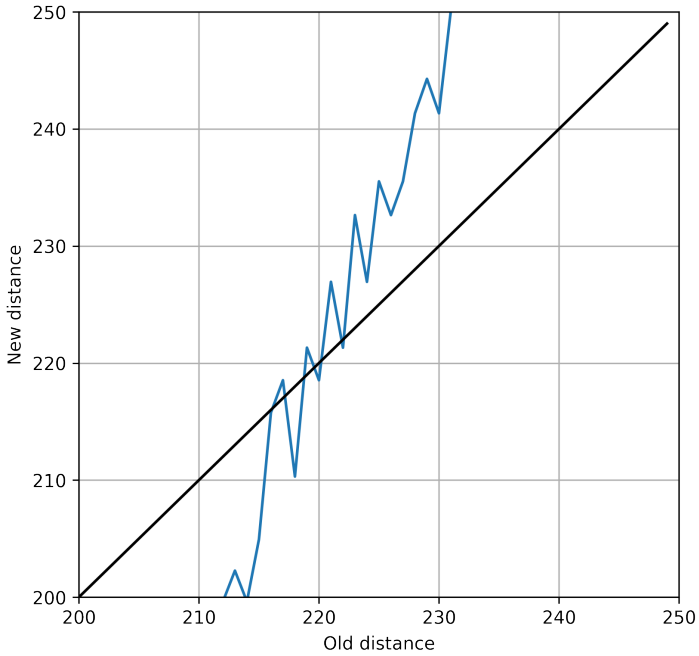


Figure 18: Zoom-in around $d = 209$ of Figure 17.

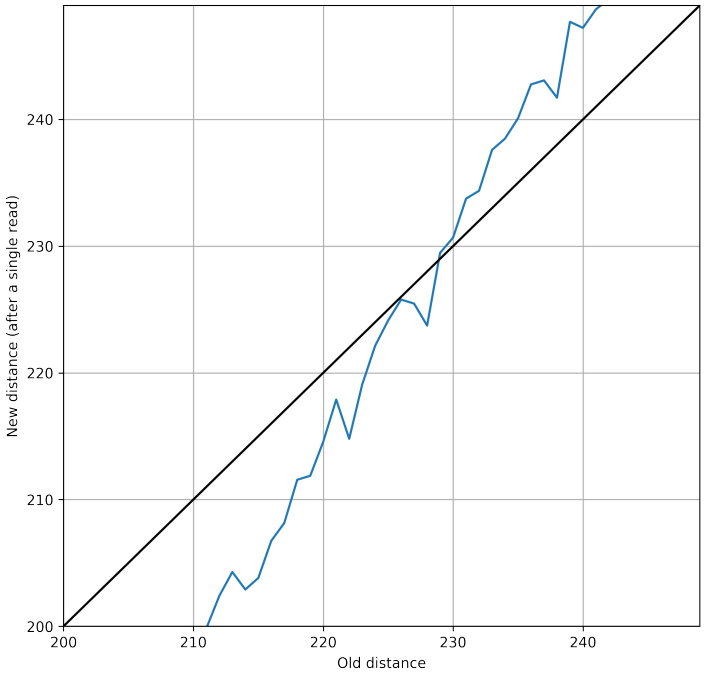


Figure 19: Zoom-in around $d = 209$ of Figure 14.

7

RESULTS (III): LOSS OF NEURONS

In SDM, the data is written distributed among millions of hard-locations, which theoretically gives SDM robustness against loss of neurons. In other words, SDM should keep convering correctly even when some neurons are dead. The question is: how robust it really is? How many neurons may die before it starts to forget things. These questions have never been addresses before.

Looking for answers to these questions, we run simulations in which we kept killing some neurons and checking whether SDM remained converging to a given bitstring or not. In these simulations, 10,000 random bitstrings were written to a 1,000-bit SDM with 1,000,000 hard-locations, and we choose one of them as our target. As the bitstrings were all written exactly once, we may generalize the results. The code is available in the "Reseting hard-locations" notebook [3].

As neurons are hard-locations in SDM, when we say that a neuron has been killed, we mean that its counters have been zeroed and a new random bitstring address has been assigned. During our simulations, no other bitstring has been written after the 10,000. Consequently, as their counters willl remain zeroed, it is exactly like ignoring the dead hard-locations in the subsequent reading operations.

In Figure 20, we can notice that SDM is robust up to 200,000 neuron deaths which is 20% of all hard-locations. Its robustness is astonishing. In fact, SDM begins to be significantly affected by loss of neurons after 600,000 neuron deaths (Figure 21), and obviously forgets everything when all neurons are dead.

It is interesting that 500,000 neuron deaths has a minor effect in SDM's recall capability (see Figure 22). It is analogous to do an hemispherectomy in a person and, after the procedure, the person being able to recall and learn almost just like before. In fact, there are clinical reports of children submitted to hemispherectomy who live an almost normal life with minor function problems.

An important observation is that around 800,000 neuron deaths (80% of all neurons) the critical distance becomes small, i.e., SDM recall capacity is very diminished. After 900,000 neuron deaths the critical distance is zero. In this case, everything has been lost.

Although there is some decrease in SDM recall after 600,000 neuron deaths, it is curious that there is a sudden change between 900,000 (90%) and 1,000,000 (100%). In Figure 23 we can see the details of this

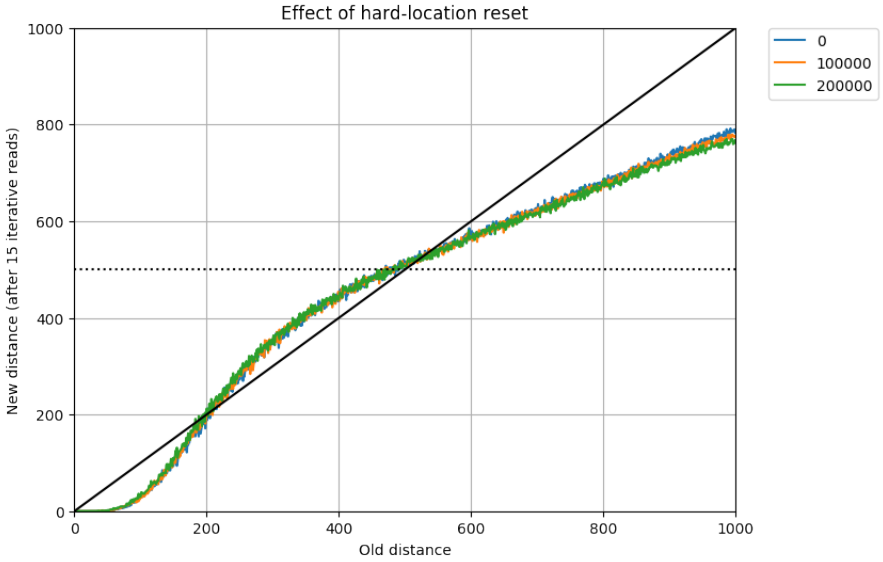


Figure 20: This graph shows the SDM's robustness against loss of neurons in a SDM with $n = 1,000$ and $H = 1,000,000$. It shows that a loss of 200,000 neurons, 20% of the total, does not seem to affect SDM whatsoever.

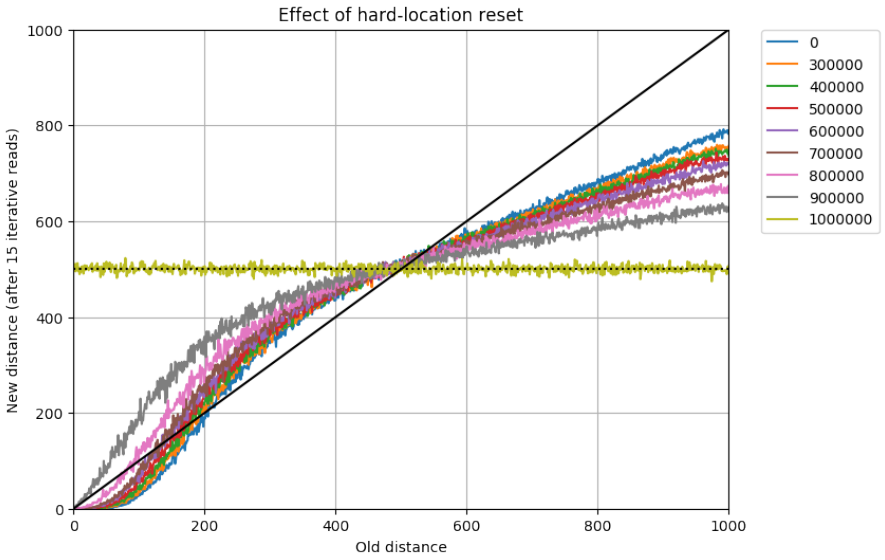


Figure 21: This graph shows the SDM's robustness against loss of neurons in a SDM with $n = 1,000$ and $H = 1,000,000$. The more neurons are lost, the smaller the critical distance, i.e., the worse the SDM recall.

non-linear change. Notice that after 950,000 even the exact clue η_0 does not converge to η.

We run exactly the same simulation for a 256-bit SDM with 1,000,000 hard-locations. The results were even more surprising, as the 256-bit SDM seems to be more robust to loss of neurons than the 1,000-bit SDM (see Figure 24). Notice that the loss of 50% of neurons barely affected the 256-bit SDM which remained functional even facing an enormous loss of 90% of neurons.

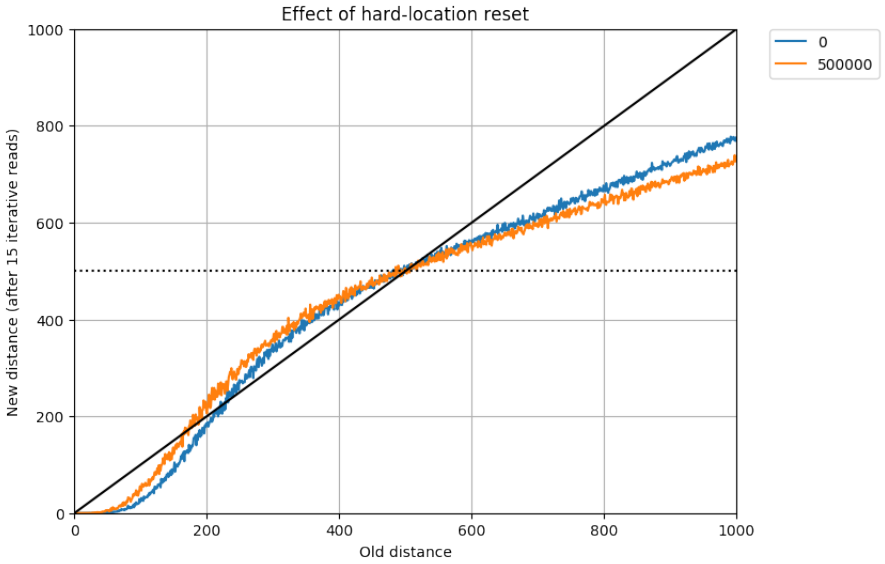


Figure 22: This graph shows the SDM's robustness against loss of neurons in a SDM with $n = 1,000$ and $H = 1,000,000$. Even when 50% of neurons are dead, SDM recall is barely affected, which is an impressive result and matches with some clinical results of children submitted to hemispherectomy.

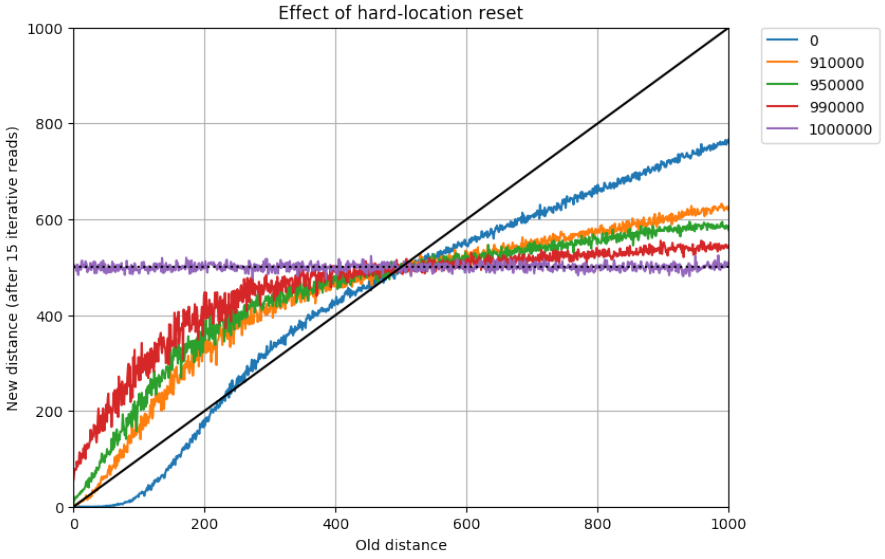
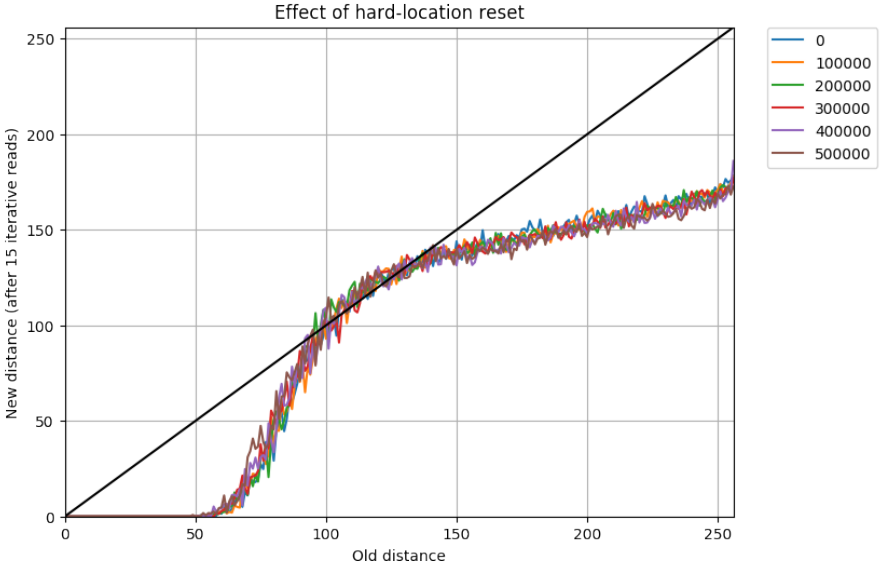


Figure 23: This graph shows the SDM's robustness against loss of neurons in a SDM with $n = 1,000$ and $H = 1,000,000$.



(a) Up to 500,000 neuron deaths

(b) From 600,000 to 1,000,000 neuron deaths

Figure 24: This graph shows the SDM's robustness against loss of neurons in a SDM with $n = 256$ and $H = 1,000,000$.

8

RESULTS (IV): GENERALIZED READ OPERATION

Murilo observed that the models of Kanerva's read ($z = 1$) and Chada's read ($z = 0$) were simple variations of a generalized read with an exponent z, which suggests experimenting with different values. Mathematically, let A be the set of the counters of the activated hardlocation, and c_i be the counter of the i-th bit. Then,

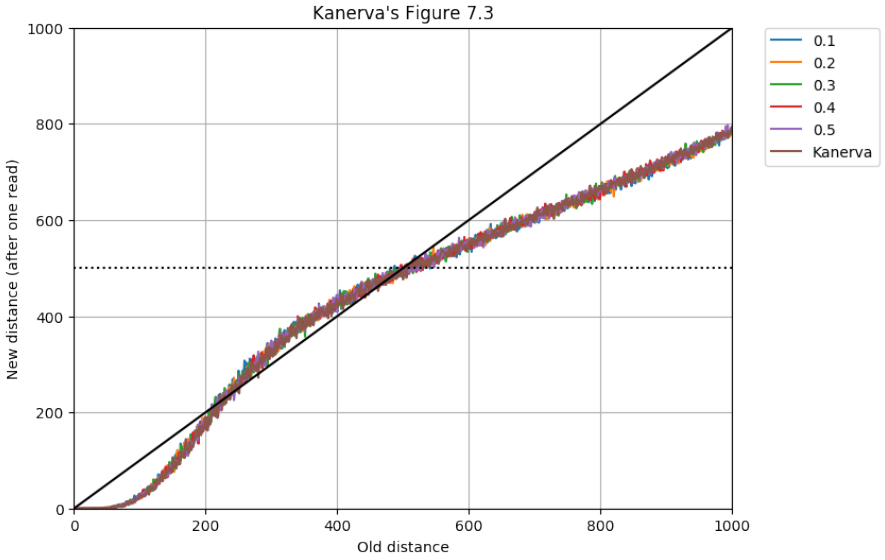
$$s_i = \sum_{c \in A} \frac{c_i}{|c_i|} |c_i|^z$$

The sum of $|c_i|^z$ turns the intermediate values from integers to float point numbers. Thus, we have developed a specific read operation which stored the intermediate values in double variables.

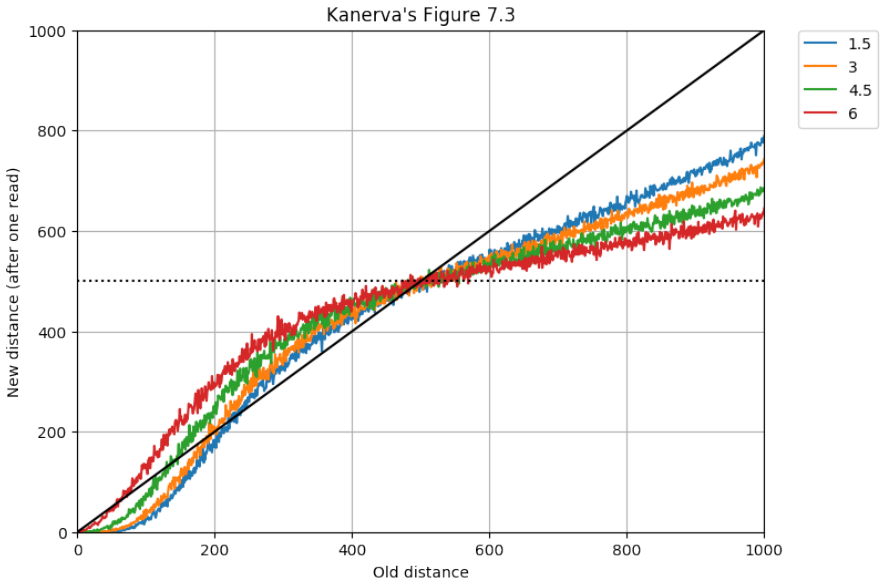
The results, however, have not yielded performance improvements. Though for $z \leqslant 1$ results are comparable to $z = 1$, for $z > 1$, the system shows a clear deterioration, with a smaller critical distance and faster divergence at large-distance reads. This is shown in Figures 25 and 26.

We understand that the critical distant is an important parameter of SDM. The bigger the critical distance, the best, because SDM is able to converge even with farther clues. For $z > 1$, the bigger the z, the smaller the critical distance. For $z = 6$, the critical distance almost reaches zero.

It is interesting that Kanerva has proposed $z = 1$ without realizing the generalized reading. Even so, he proposed the z with the highest critical distance.

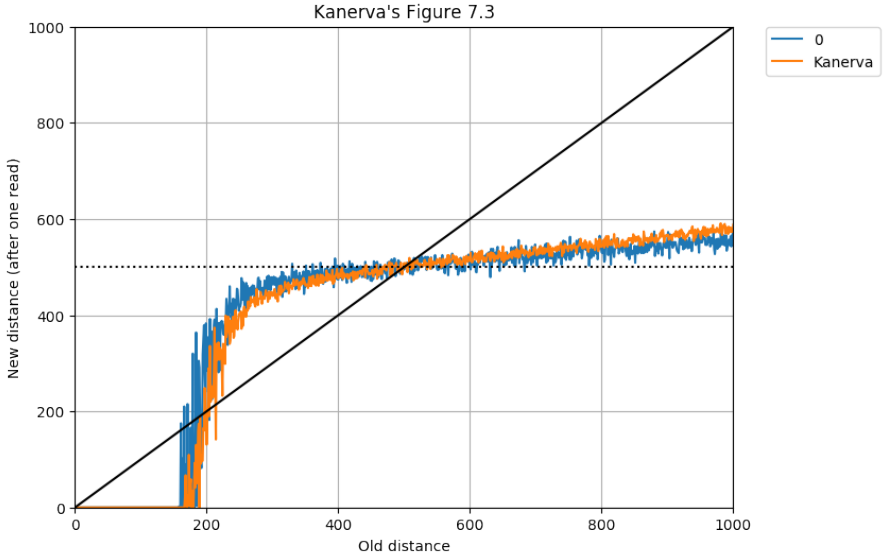


(a) SDM behavior when $z \in \{0.1, 0.2, 0.3, 0.4, 0.5, 1\}$

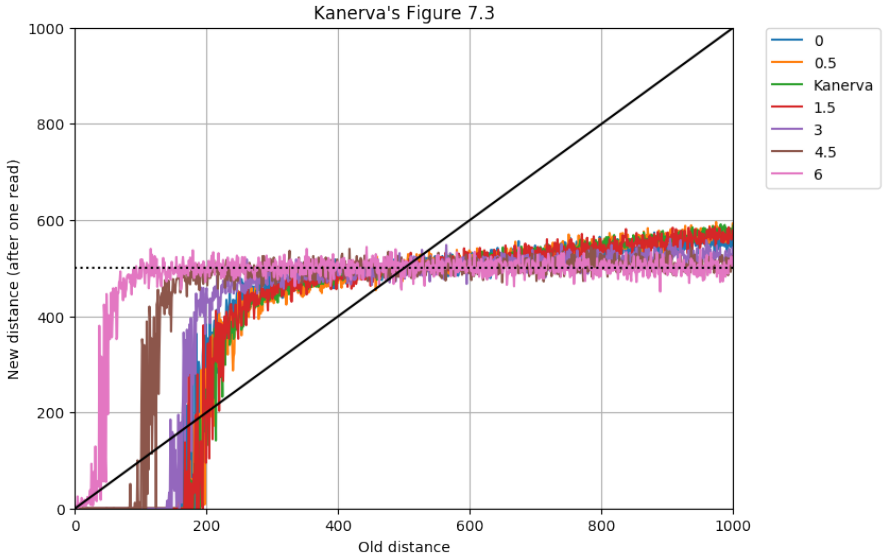


(b) SDM behavior when $z \in \{1.5, 3, 4.5, 6\}$

Figure 25: (a) and (b) show the behavior of a single read. As stated previously, we can see a deterioration of convergence, with lower critical distance as $z > 1$. Another observation can be made here, concerning the discrepancy of Kanerva's Fig 7.3 and our data. It seems that Kanerva may not have considered that a single read would only 'clean' a small number of dimensions *after the critical distance*. What we observe clearly is that with a single read, as the distance grows, the system only 'cleans' towards the orthoghonal distance 500 after a number of iterative readings.



(a) $z \in \{0, 1\}$



(b) $z \in \{0, 0.5, 1, 1.5, 3, 4.5, 6\}$

Figure 26: (a) and (b) show the behavior of Figure 25, now executed with 6-iterative reads. What we observe clearly is that with a single read, as the distance grows, the system only 'cleans' towards the orthoghonal distance 500 after a number of iterative readings.

9

RESULTS (V): PERFORMANCE

Performance matters — and has always mattered. If an experiment takes a few seconds, there is no point arguing whether we should try it. If it takes a few hours, maybe we should think first. If it takes a few days — or more —, it is important to devise a good plan. As SDM consumes large processor and memory resources, some experiments may take a long time.

Each scan on a 1,000-bits SDM with 1,000,000 hard-locations executes 10^9 bit compares through $10^9/64 = 15,625,000$ XORs and calls to the built-in popcount. So, 10,000 writes execute 10^{13} bit compares, while a 6-iterative reading executes $6 \cdot 10^{12}$ bit compares. The heatmap of Figure 7a executed $3.05 \cdot 10^{15}$ bit compares. For comparison, the number of seconds since Jesus's birth is 63,639,648,000 = $6.36 \cdot 10^{10}$. The number of people who have ever lived on earth is estimated to be $1,08 \cdot 10^{11}$. There are approximtely $1.8 \cdot 10^9$ websites in the internet. A modern laptop can increment a counter $5 \cdot 10^8$ times per second. Hence, a naive implementation of SDM may take several hours — or days — to simply write 10,000 random bitstrings.

Amazon EC2 p3.2xlarge has generated the heatmap of Figure 7a in 15 minutes and 3 seconds. It has compared $3.37 \cdot 10^{12}$ bits per second through $52.6 \cdot 10^9 = 52.6$ billion XORs and popcounts per second. It is a 60-fold improvement over the first versions of the code which took 16 hours to generate the same heatmap (and its memory use was already optimized and the computations were distributed in threads).

We have created a benchmark to be able to compare the performance in different devices. So, the same performance test was executable in our devices, with results reported in tables and figures. The benchmark has 3 parts: (i) the first part consists of comparing the available OpenCL kernels to find which works best for that device; than (ii) the second part consists of comparing the linear scanner, the thread scanner, and the OpenCL scanners with the best kernel found in part one; finally, (iii) the third part consists of comparing read and write operations using the thread scanner and the OpenCL scanner with the best kernel. Each part was run for three SDM setups: (i) $n = 1,000$, $r = 451$, and $H = 1,000,000$; (ii) $n = 256$, $r = 103$, and $H = 1,000,000$; and (iii) $n = 10,000$, $r = 4850$, and $H = 1,000,000$. The whole source code is available in the "Performance test" notebook [3]. We would like to invite the reader to run this benchmark and, if possible, share the results.

Our first device was a personal MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU. It was not possible to run the 10,000-bits in this device because there was no memory available — it needs 37.25 GB of memory. For its results, see Table 3

Our second device was an iMac Retina 5K 27-inch 2017 with a 3.8GHz Intel core i5 processor, 8GB DDR4 RAM, and a Radeon Pro 580 8G CPU. For its results, see Table **??**.

Beyond that, we were also running in state-of-the-art devices: (i) an Amazon EC2 p2.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU (see Table **??**), and (ii) an Amazon EC2 p3.2xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and NVIDIA Tesla V100 GPU (see Table 6).

9.1 KERNELS COMPARISON

OpenCL is a framework for writting software that executes in *heterogeneous* devices [20], like CPUs, GPUs, FPGA and other co-processor for hardware acceleration. Because they are heterogeneous, they may differ a lot in architecture and performance, which means there is no one-size-fits-all kernel. A *kernel* is generally a small function on the code that runs in thousands of parallel threads, executing the same steps on different parts of a large vector or matrix. The slower kernel for one device may be the fastest for another device, as we will see happening in our case.

A total of 8 kernels have been developed for benchmarking in our framework: single_scan0, single_scan1, single_scan2, single_scan3, single_scan4, single_scan5, single_scan5_unroll, and single_scan6. Each scan uses a different algorithm to do exactly the same thing: calculate which hard-locations are inside the circle of the given bitstring. They differ in how they split the work between work-groups and how they explore the GPU architecture to obtain performance gains.

The difference in which kernel is the best depends also in the SDM setup. The best kernel for the 1,000-bits SDM in the iMac 2017 was single_scan5_unroll with average scan time of 3.61ms; but, for the 256-bits SDM in the same device, it was single_scan0 with average scan time of 3.03ms; while, for the 10,000-bits SDM in the same device, it was single_scan6 with 10.96ms (see Table 4).

We recommend users to run a specific kernel comparison test for their own GPU and SDM settings. This is available in the jupyter notebooks.

9.2 SCANNERS COMPARISON

In this section, we are comparing the OpenCL scanner (with the best kernel) with the linear scanner and the threads scanner. Again, which one is faster depends on the SDM settings. In the iMac 2017, the faster scanner for a 1,000-bits SDM was the OpenCL scanner with , but for a 256-bit SDM was the threads scanner.

What happened here is that the OpenCL kernel chosen was a generic one which performs the scan for any SDM. It is always possible to optimize the OpenCL kernel to a specific setting, and it would be faster than the threads. By default, the framework chooses a generic kernel which we believe would be reasonable for the most common setups.

We can notice that Amazon EC2 p3.2xlarge and p2.xlarge's linear and thread scanners, both running on CPU, were much slower than the CPU of both the personal MacBook Pro and the iMac 2017. As Amazon EC2 are virtual machines with GPU devices, their CPU is shared with other virtual machines which reduces CPU power significantly. Hence, for both virtual machines we have tested, using the GPU seriously boosts performance, but using the CPU should be avoided. See Tables 4

9.3 READ AND WRITE OPERATIONS

Even though scaning is an important part of the operations, we are really interested in the performance of the enterity of operations themselves. Comparing the times of the thread and OpenCL scanners with the times of their respective operations (either read or write), we can notice that their difference remains almost constant, which means the operation time itself is negligible when compared to the scan time. In other words, in order to gain even more performance, we have to pursue ways to improve the scan.

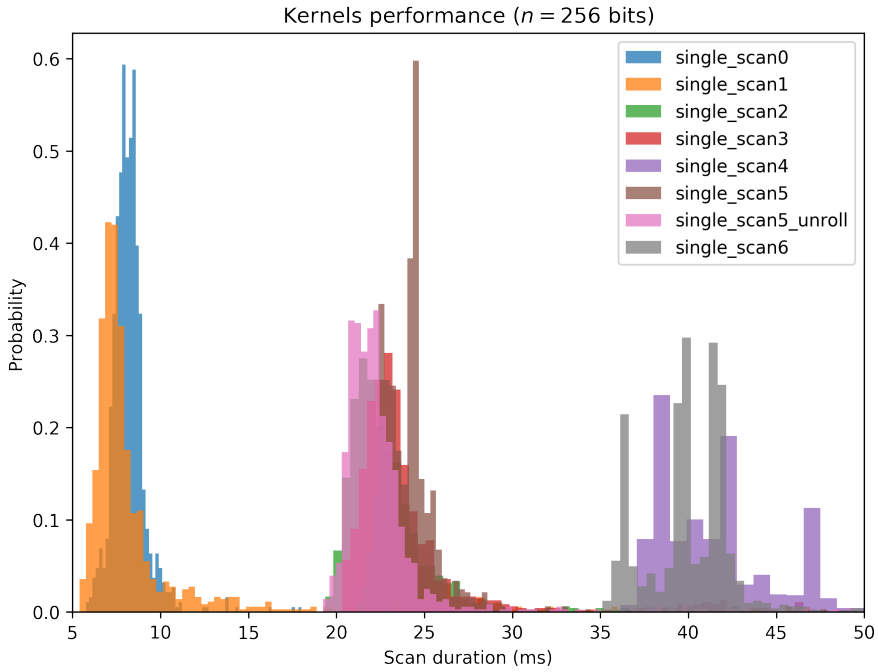
9.3.1 *Summary of results*

The results, beyond showing the obvious fact that consumer grade hardware is not comparable to the Amazon instances, indicate a non-trivial issue: The chosen kernel for scanning the memory is of crucial importance to performance, and this kernel speed is a function of both the hardware used and the particular parameters used in the SDM settings.

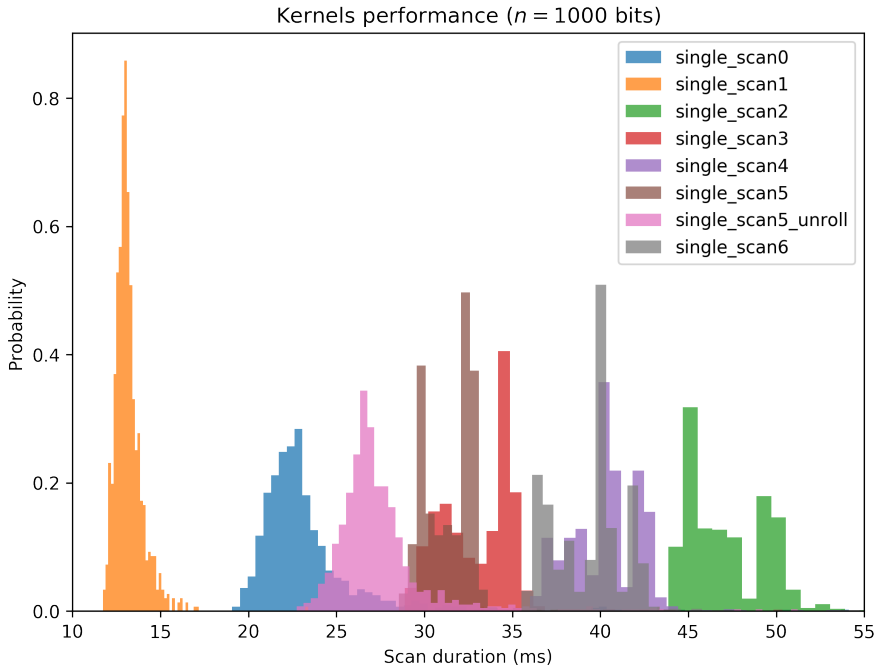
It is reasonable to consider that the performance results obtained are of particular merit, and one particular fact stands out: The scanning of 1,000,000 hard locations has become, in the desired professional-grade machines, *faster* than the updating of the 1,000 active locations.

	256 bits	**1,000 bits**	**10,000 bits**
Kernel	Duration (ms)	Duration (ms)	Duration (ms)
single_scan0	8.36	23.60	
single_scan1	10.43	13.22	
single_scan2	23.48	47.28	
single_scan3	25.51	33.06	
single_scan4	42.39	40.32	
single_scan5	24.42	31.51	
single_scan5_unroll	22.77	27.55	
single_scan6	42.18	39.48	
Scanner	Duration (ms)	Duration (ms)	Duration (ms)
Linear scan	9.07	17.98	
Thread scan	5.14	10.17	
OpenCL scan	8.05	12.35	
Operation	Duration (ms)	Duration (ms)	Duration (ms)
Thread write	6.72	14.13	
Thread single read	5.88	11.12	
OpenCL write	6.39	18.55	
OpenCL single read	5.26	13.06	

Table 3: MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU. The SDM settings were: (i) $n = 256$, $r = 103$, and $H = 1,000,000$; (ii) $n = 1,000$, $r = 451$, and $H = 1,000,000$; and (iii) $n = 10,000$, $r = 4850$, and $H = 1,000,000$. There is no benchmark for $n = 10,000$ because memory is not enough on either RAM or GPU—it would consume 37.25 GB of RAM and 1.2GB of memory in the GPU. For the histogram of durations, see Figures 27, 28, 29, and 30.

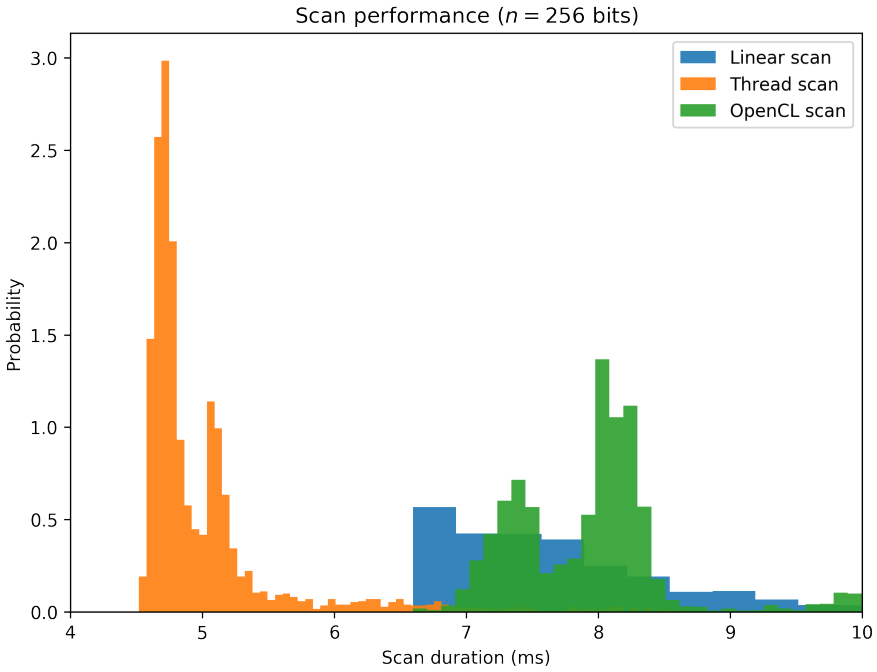


(a) $n = 256$, $r = 103$, and $H = 1,000,000$

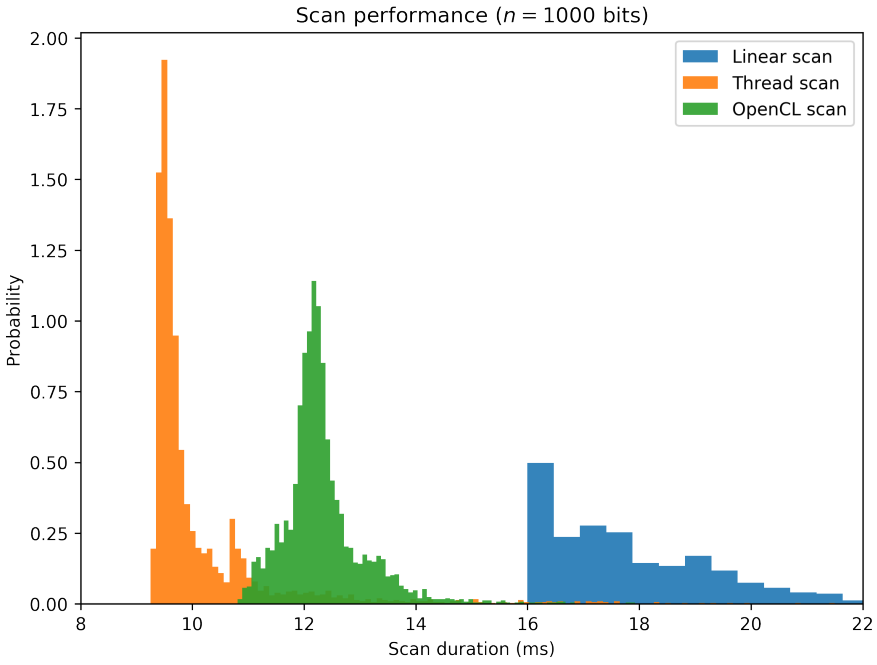


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

Figure 27: Kernel comparisons for MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU.

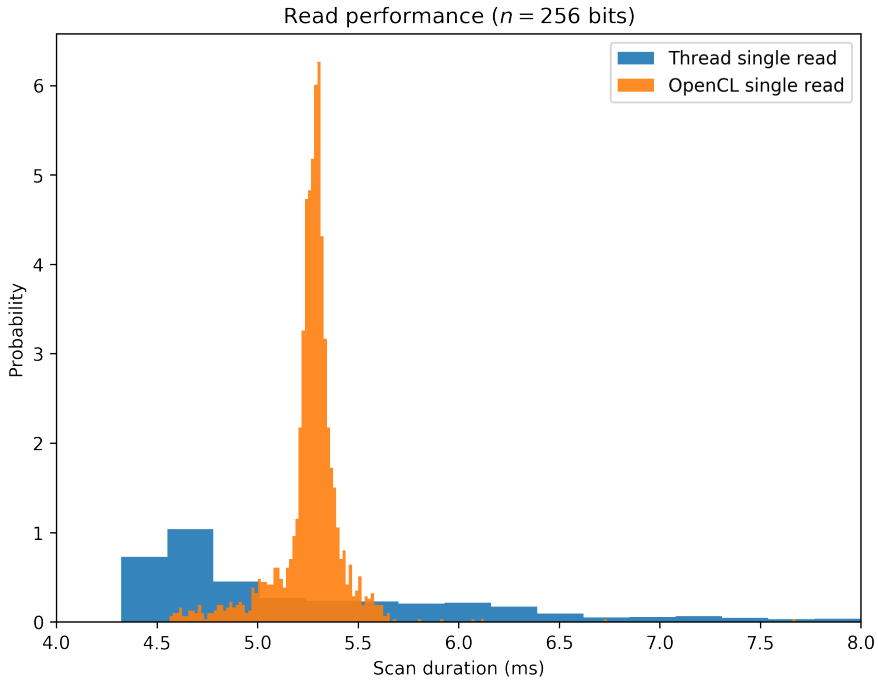


(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

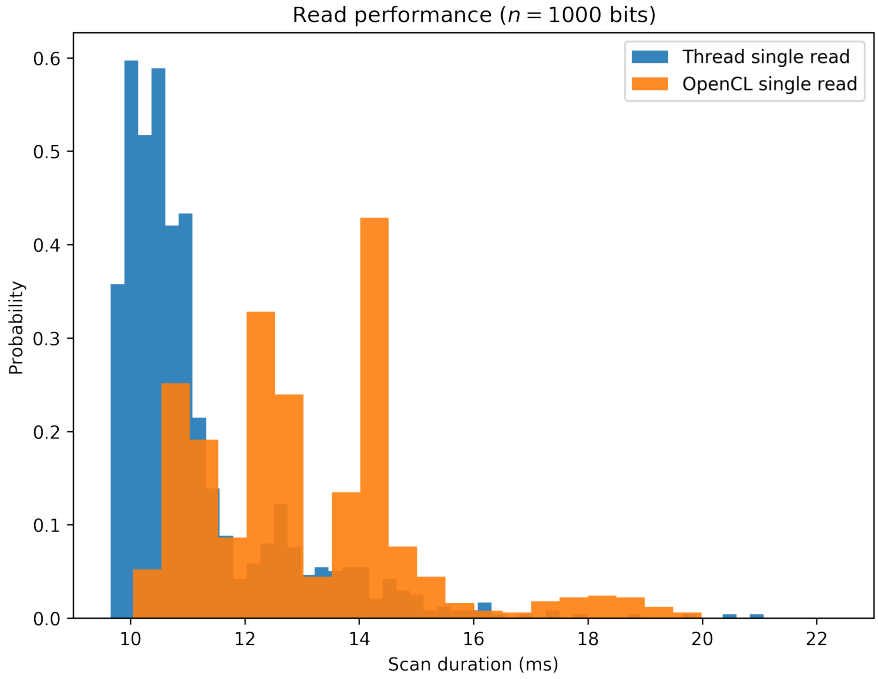


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

Figure 28: Scanner comparisons for MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU.

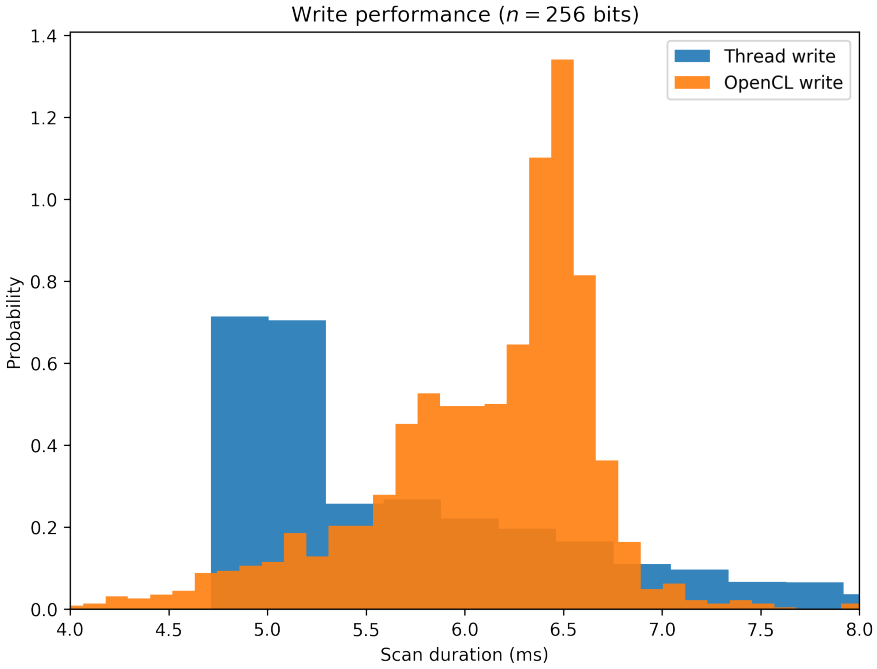


(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

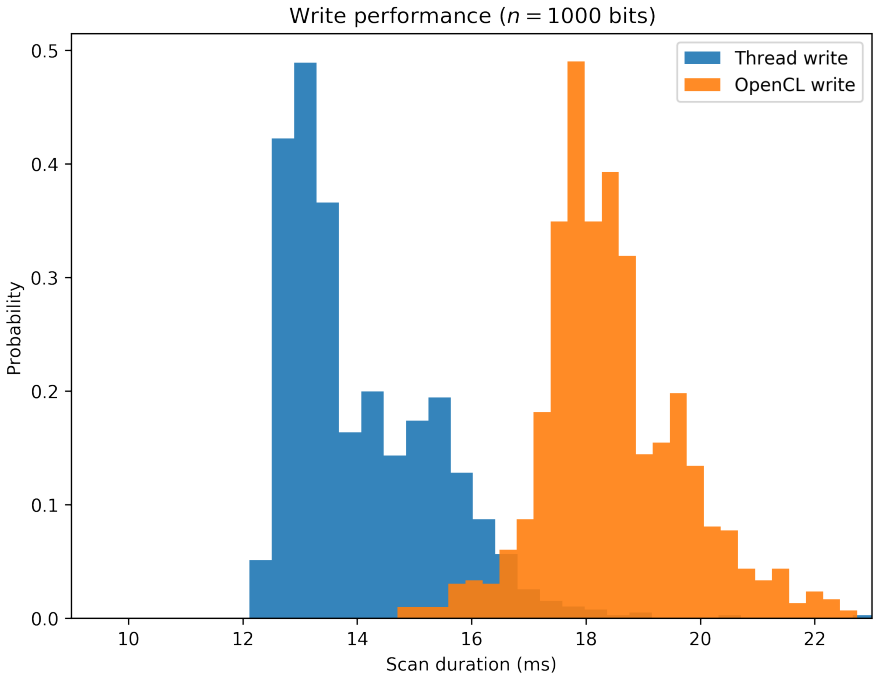


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

Figure 29: Read operation comparisons for MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

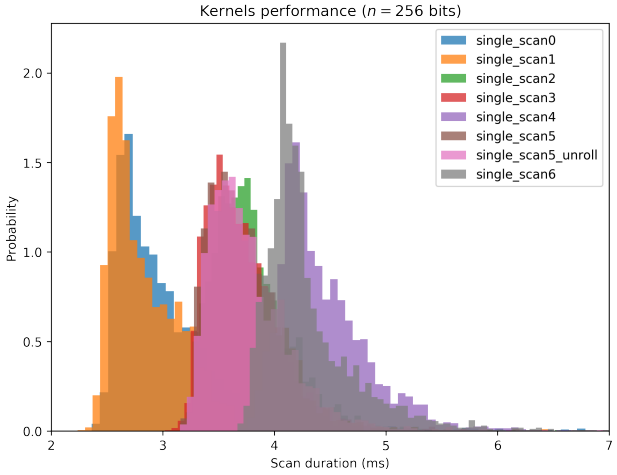


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

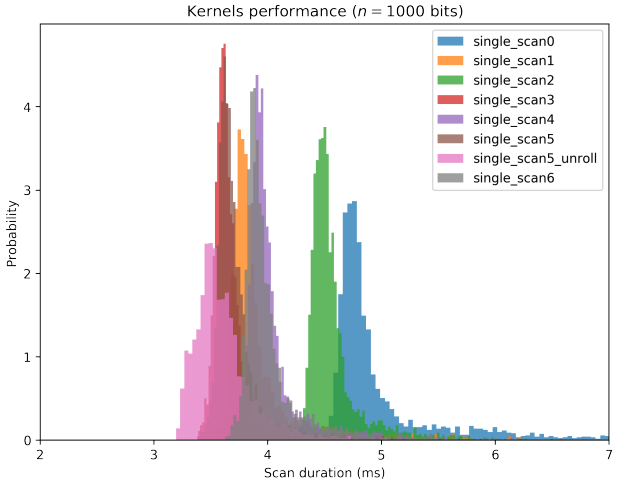
Figure 30: Write operation comparisons for MacBook Pro Retina 13-inch Late 2013 with a 2.6GHz Intel core i5 processor, 6GB DDR3 RAM, and Intel Iris GPU.

	256 bits	**1,000 bits**	**10,000 bits**
Kernel	Duration (ms)	Duration (ms)	Duration (ms)
single_scan0	3.03	5.00	61.06
single_scan1	2.87	3.95	44.96
single_scan2	3.82	4.57	44.98
single_scan3	3.72	3.68	12.67
single_scan4	4.48	4.04	11.45
single_scan5	3.76	3.72	12.58
single_scan5_unroll	3.79	3.61	11.37
single_scan6	4.36	4.02	10.96
Scanner	Duration (ms)	Duration (ms)	Duration (ms)
Linear scan	5.04	12.25	116.38
Thread scan	2.92	6.95	53.56
OpenCL scan	2.81	4.20	12.95
Operation	Duration (ms)	Duration (ms)	Duration (ms)
Thread write	3.28	13.34	
Thread single read	2.55	10.39	
OpenCL write	2.64	7.90	
OpenCL single read	2.14	5.25	

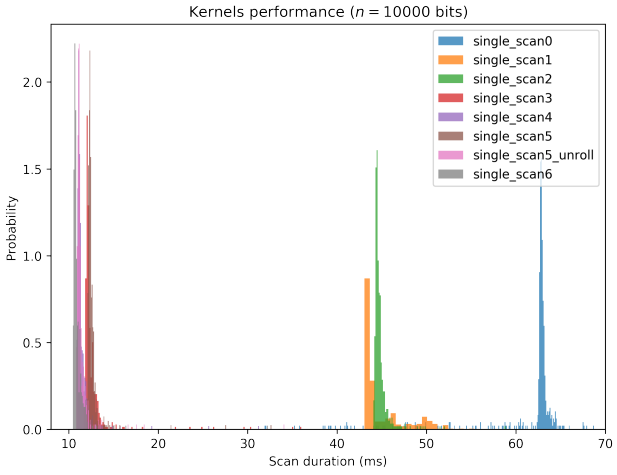
Table 4: iMac Retina 5K 27-inch 2017 with a 3.8GHz Intel core i5 processor, 8GB DDR4 RAM, and a Radeon Pro 580 8G GPU. The SDM settings were: (i) $n = 256$, $r = 103$, and $H = 1,000,000$; (ii) $n = 1,000$, $r = 451$, and $H = 1,000,000$; and (iii) $n = 10,000$, $r = 4850$, and $H = 1,000,000$. There is no benchmark for read and write operations with $n = 10,000$ because RAM is not enough to allocate the counters—it would consume 37.25 GB of RAM. For the histogram of durations, see Figures 31, 32, 33, and 34.



(a) $n = 256$, $r = 103$, and $H = 1,000,000$

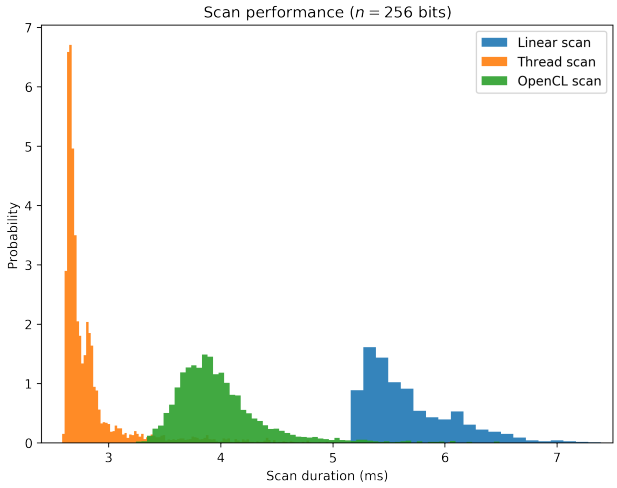


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

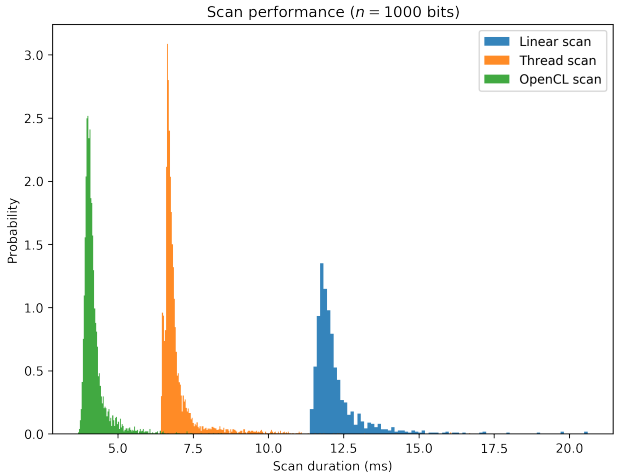


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

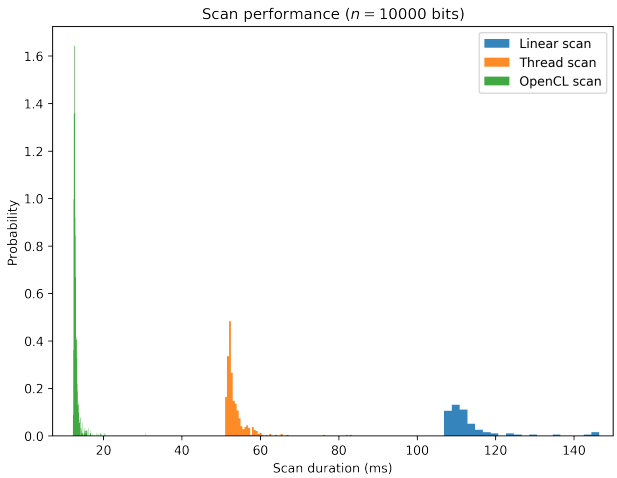
Figure 31: Kernel comparisons for iMac Retina 5K 27-inch 2017 with a 3.8GHz Intel core i5 processor, 8GB DDR4 RAM, and a Radeon Pro 580 8G GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

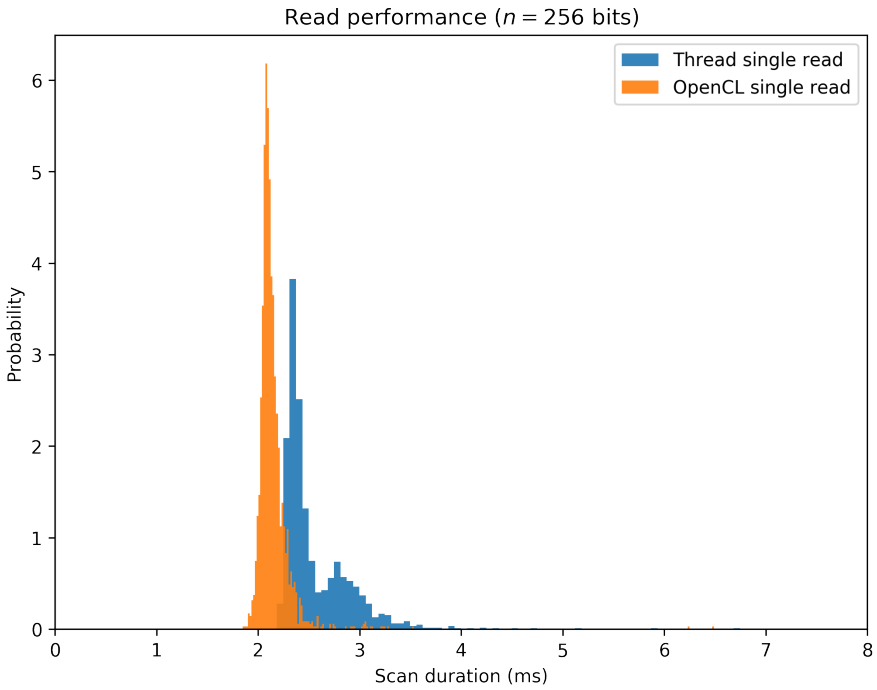


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

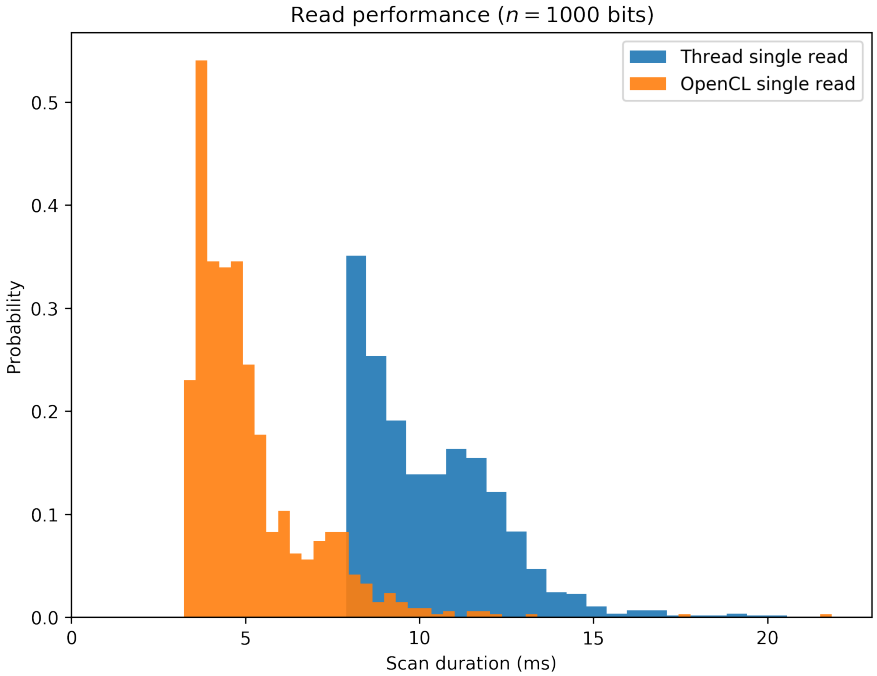


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

Figure 32: Scanner comparisons for iMac Retina 5K 27-inch 2017 with a 3.8GHz Intel core i5 processor, 8GB DDR4 RAM, and a Radeon Pro 580 8G GPU.

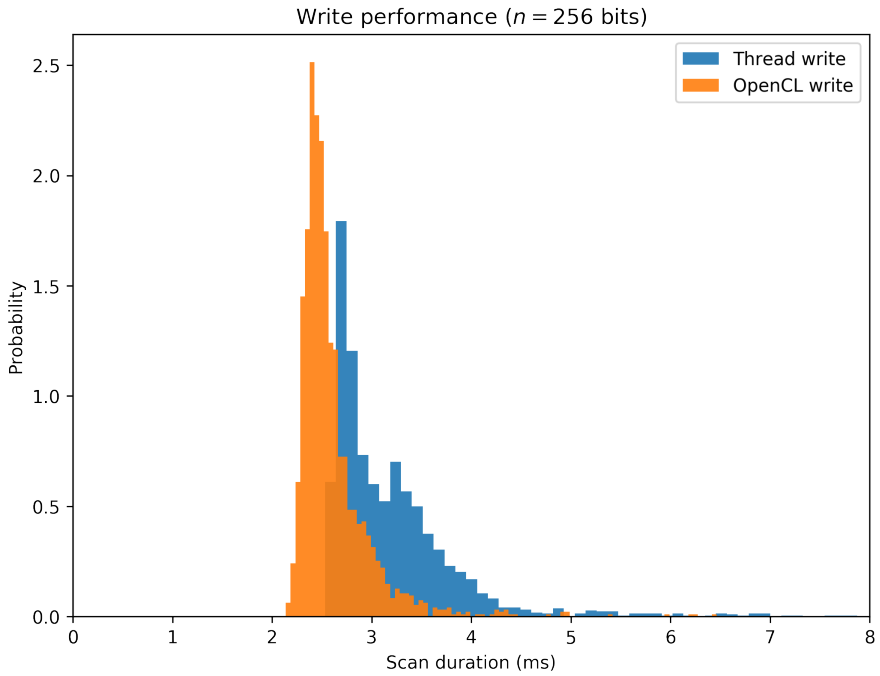


(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

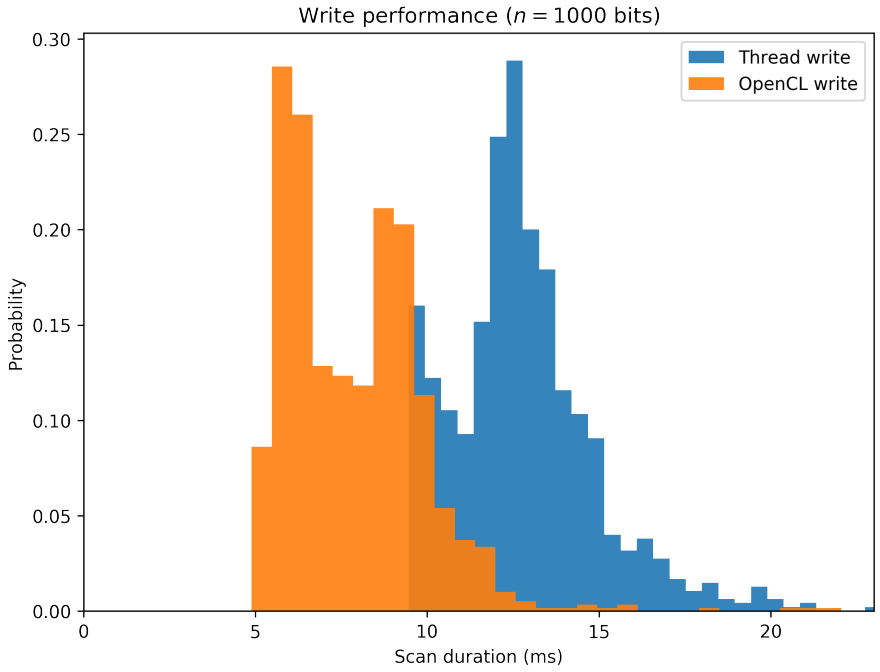


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

Figure 33: Read operation comparisons for iMac Retina 5K 27-inch 2017 with a 3.8GHz Intel core i5 processor, 8GB DDR4 RAM, and a Radeon Pro 580 8G GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

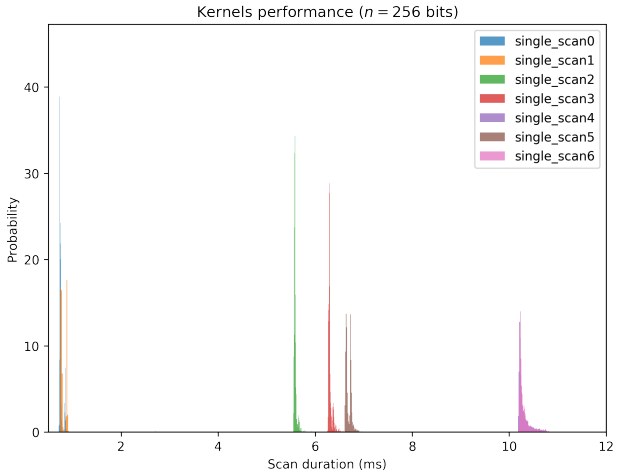


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

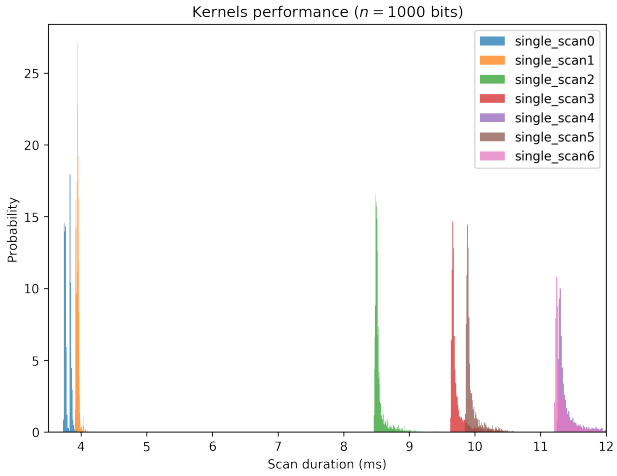
Figure 34: Write operation comparisons for iMac Retina 5K 27-inch 2017 with a 3.8GHz Intel core i5 processor, 8GB DDR4 RAM, and a Radeon Pro 580 8G GPU.

	256 bits	**1,000 bits**	**10,000 bits**
Kernels	Duration (ms)	Duration (ms)	Duration (ms)
single_scan0	0.76	3.79	35.45
single_scan1	0.80	3.94	57.80
single_scan2	5.59	8.54	63.71
single_scan3	6.31	9.73	39.92
single_scan4	10.29	11.38	45.49
single_scan5	6.69	9.95	43.51
single_scan5_unroll			
single_scan6	10.29	11.33	41.42
Scanners	Duration (ms)	Duration (ms)	Duration (ms)
Linear scan	19.09	64.73	600.75
Thread scan	9.95	32.81	296.56
OpenCL scan	6.88	10.67	46.73
Operations	Duration (ms)	Duration (ms)	Duration (ms)
Thread write	11.80	42.64	383.50
Thread single read	10.49	35.37	307.97
OpenCL write	8.84	19.31	122.17
OpenCL single read	7.50	11.72	55.47

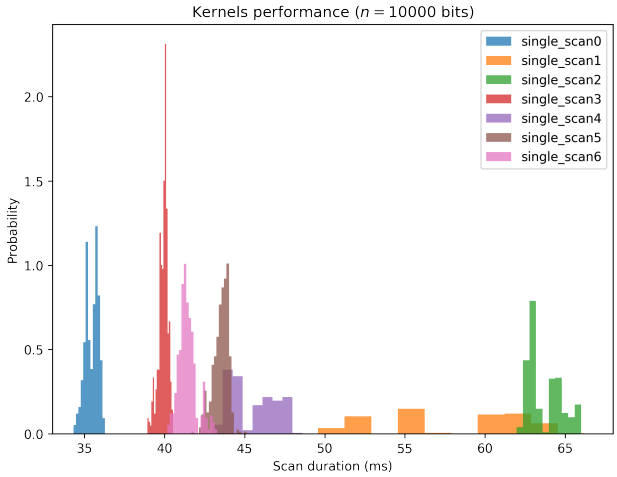
Table 5: Amazon EC2 p2.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU. Running an SDM with $n = 256$ bits, $H = 1,000,000$, and $r = 103$. The SDM settings were: (i) $n = 256$, $r = 103$, and $H = 1,000,000$; (ii) $n = 1,000$, $r = 451$, and $H = 1,000,000$; and (iii) $n = 10,000$, $r = 4850$, and $H = 1,000,000$. There is no benchmark for kernel single_scan5_unroll because it returns the wrong result in this GPU. The problem is related with the premises of the optimization used by this kernel, which are not true for this GPU. For the histogram of durations, see Figures 35, 32, 33, and 34.



(a) $n = 256$, $r = 103$, and $H = 1,000,000$

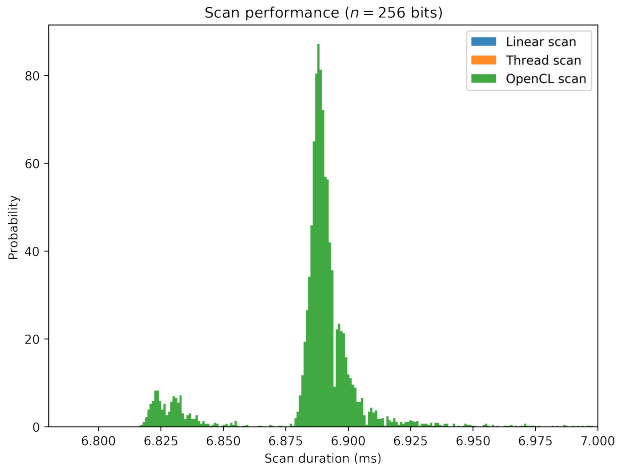


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

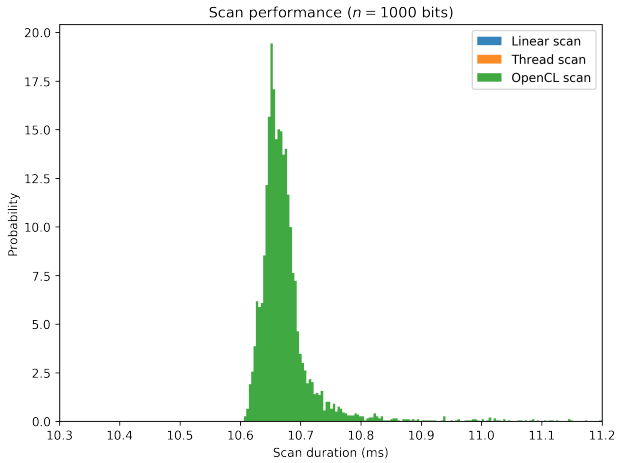


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

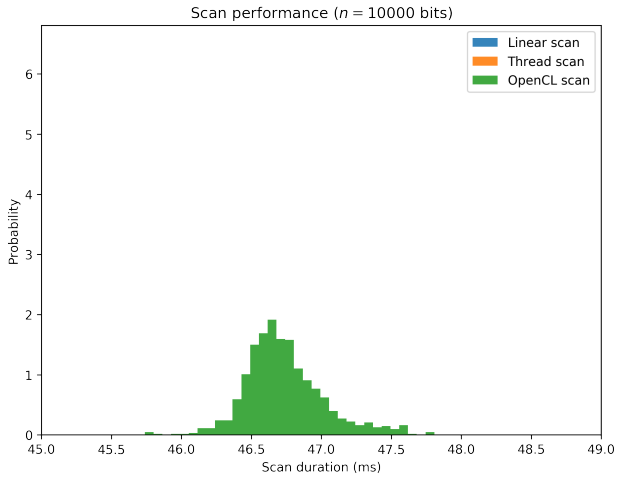
Figure 35: Kernel comparisons for Amazon EC2 p2.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

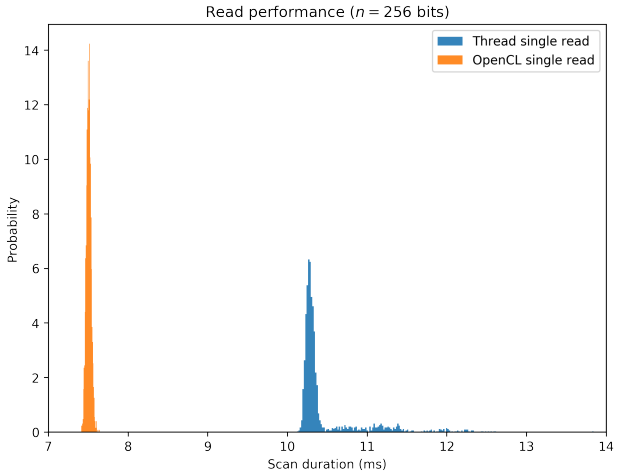


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

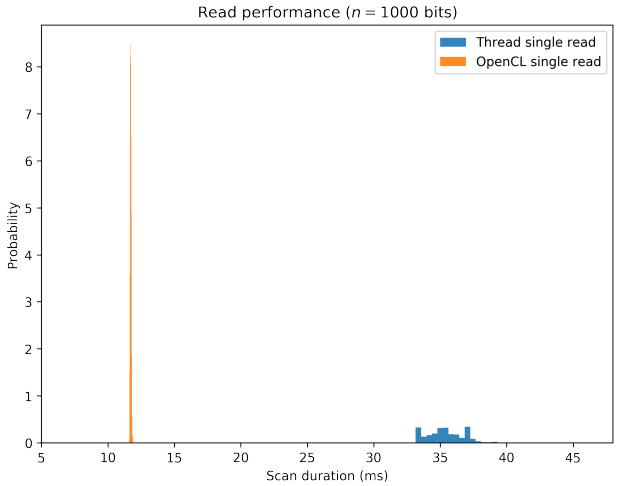


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

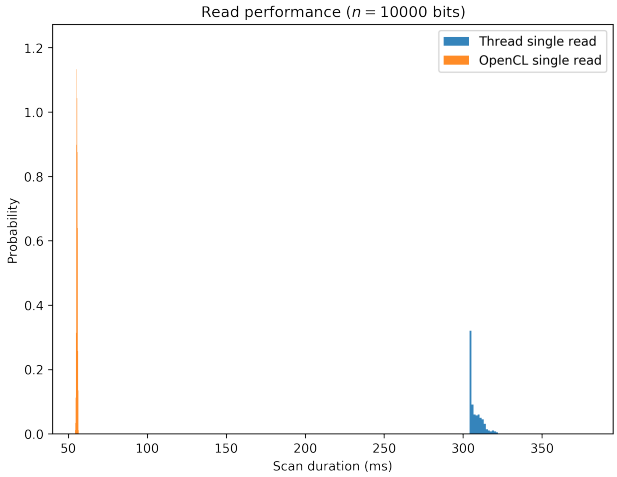
Figure 36: Scanner comparisons for Amazon EC2 p2.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

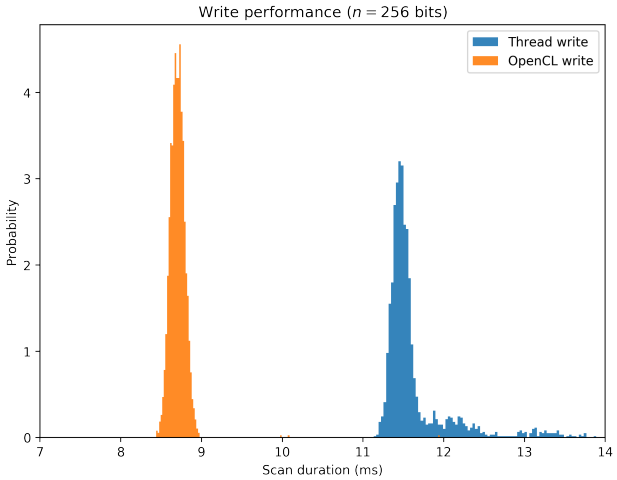


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

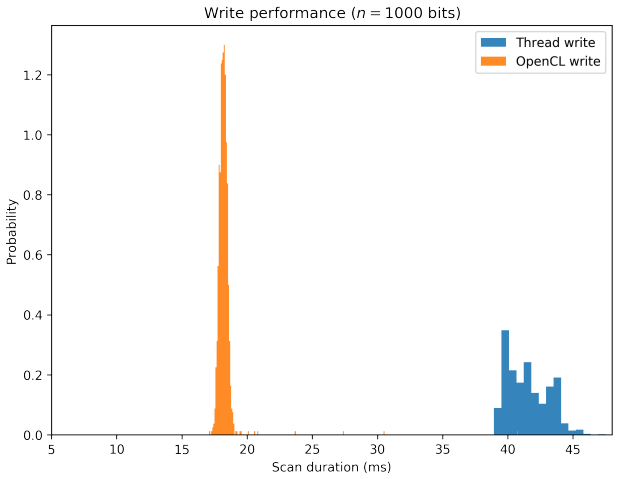


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

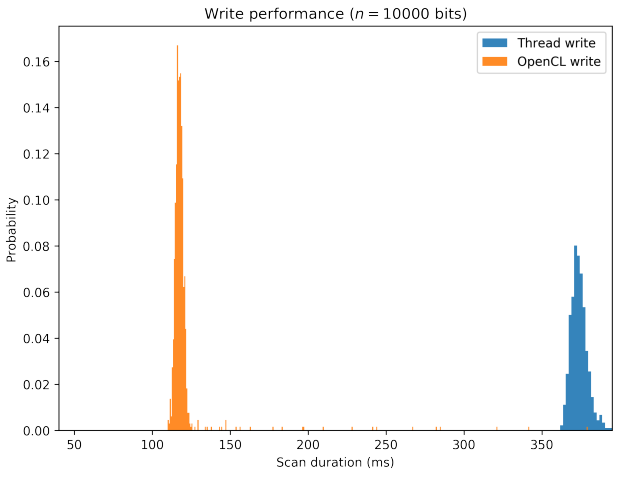
Figure 37: Read operation comparisons for Amazon EC2 p2.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$



(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

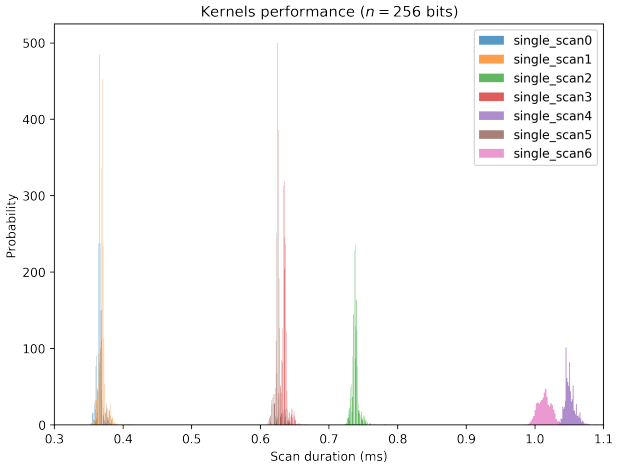


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

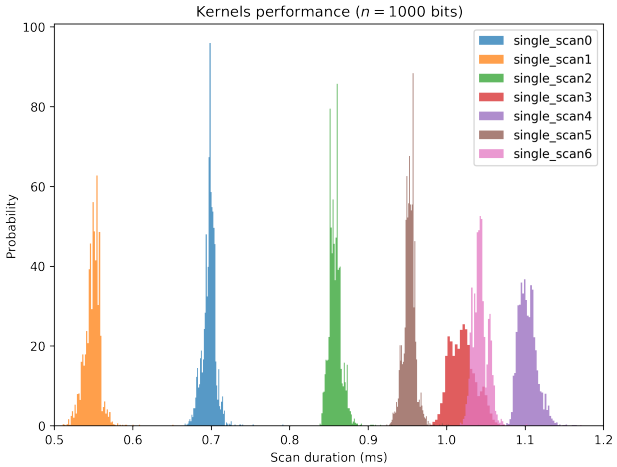
Figure 38: Write operation comparisons for Amazon EC2 p2.xlarge with Intel Xeon E5-2686v4 processor, 61GB DDR3 RAM, and NVIDIA K80 GPU.

	256 bits	**1,000 bits**	**10,000 bits**
Kernel	Duration (ms)	Duration (ms)	Duration (ms)
single_scan0	0.36	0.69	20.60
single_scan1	0.36	0.54	4.94
single_scan2	0.73	0.85	5.01
single_scan3	0.63	1.02	6.07
single_scan4	1.05	1.10	5.99
single_scan5	0.62	0.95	5.36
single_scan5_unroll			
single_scan6	1.01	1.04	5.96
Scanner	Duration (ms)	Duration (ms)	Duration (ms)
Linear scan	17.60	58.34	540.97
Thread scan	5.19	16.39	198.74
OpenCL scan	0.63	0.85	5.74
Operation	Duration (ms)	Duration (ms)	Duration (ms)
Thread write	7.59	28.47	222.08
Thread single read	6.17	17.44	145.01
OpenCL write	2.33	8.77	80.48
OpenCL single read	1.01	1.82	13.99

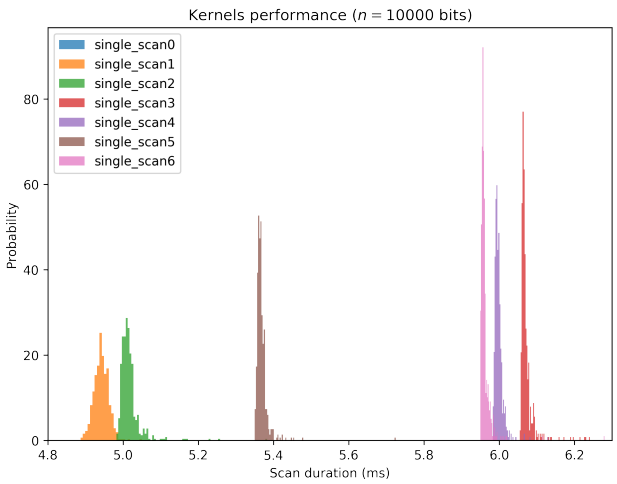
Table 6: Amazon EC2 p3.2xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and NVIDIA Tesla V100 GPU. The SDM settings were: (i) $n = 256$, $r = 103$, and $H = 1,000,000$; (ii) $n = 1,000$, $r = 451$, and $H = 1,000,000$; and (iii) $n = 10,000$, $r = 4850$, and $H = 1,000,000$. There is no benchmark for kernel single_scan5_unroll because it returns the wrong result in this GPU. The problem is related with the premises of the optimization used by this kernel, which are not true for this GPU. For the histogram of durations, see Figures 39, 40, 41, and 42.



(a) $n = 256$, $r = 103$, and $H = 1,000,000$

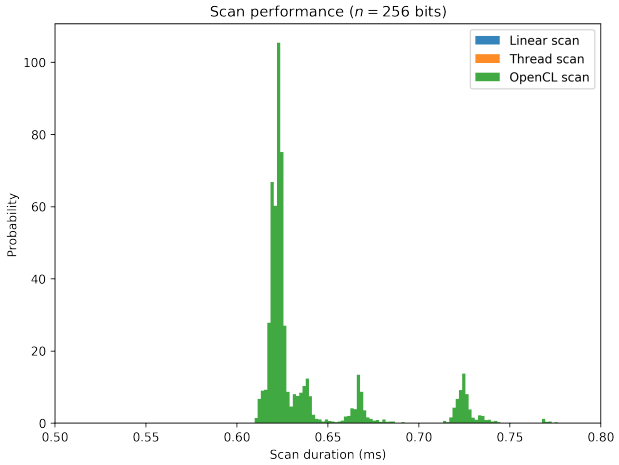


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

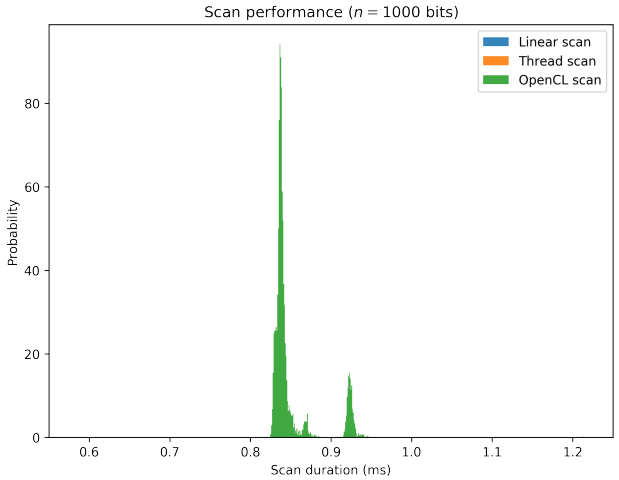


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

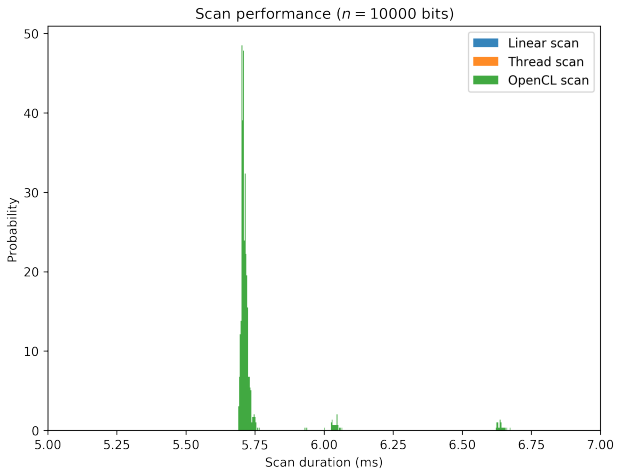
Figure 39: Kernel comparisons for Amazon EC2 p3.2xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and NVIDIA Tesla V100 GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

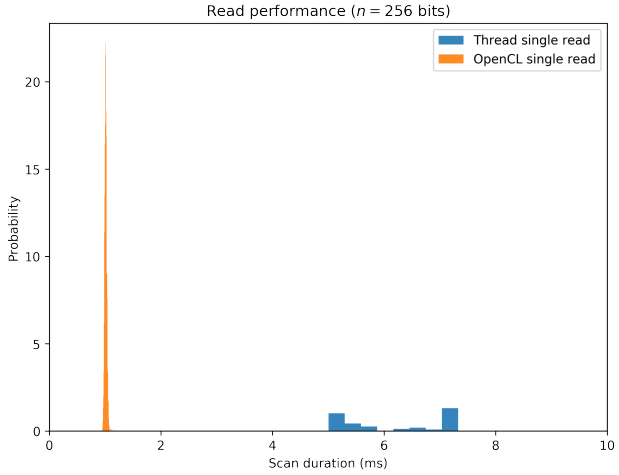


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

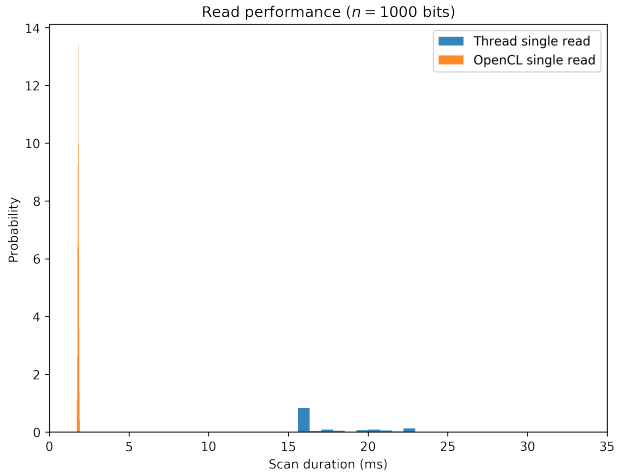


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

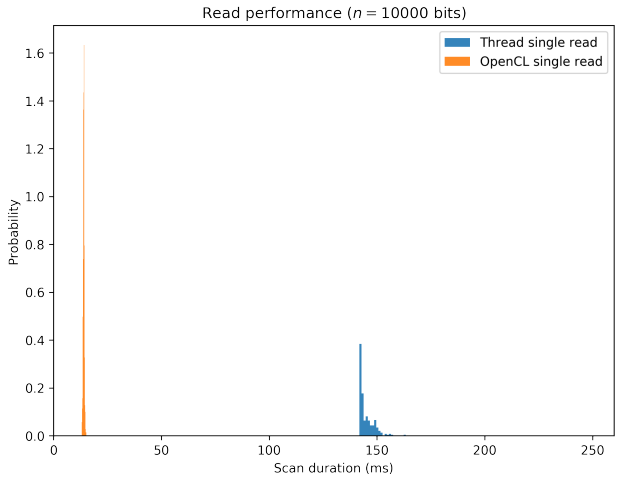
Figure 40: Scanner comparisons for Amazon EC2 p3.2xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and NVIDIA Tesla V100 GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$

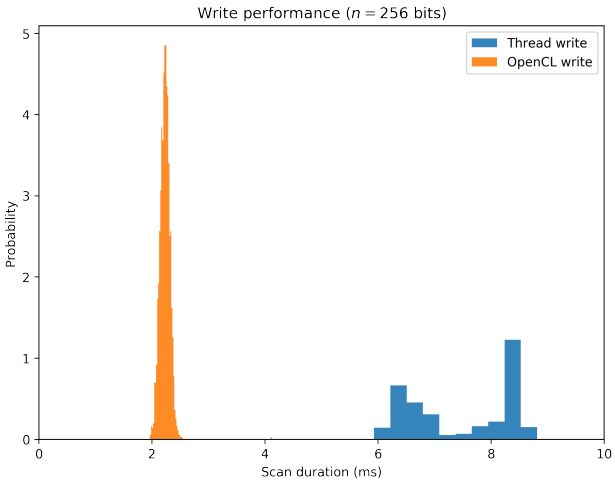


(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$

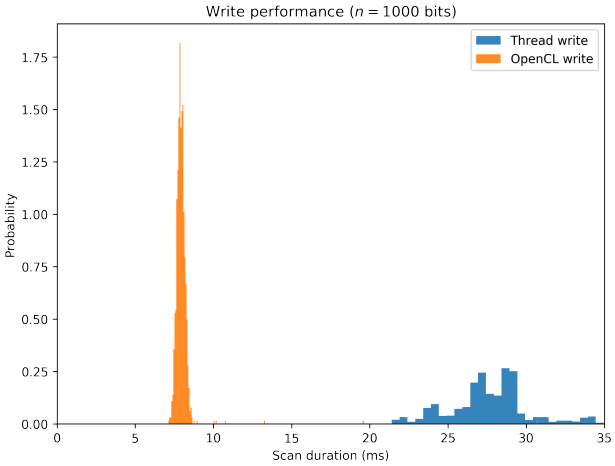


(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

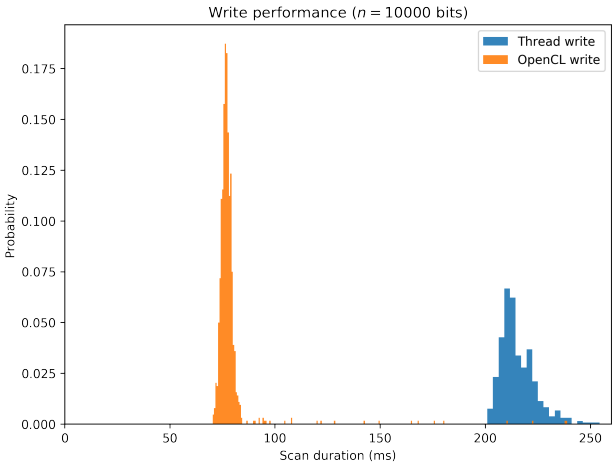
Figure 41: Read operation comparisons for Amazon EC2 p3.2xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and NVIDIA Tesla V100 GPU.



(a) $n = 1,000$, $r = 451$, and $H = 1,000,000$



(b) $n = 1,000$, $r = 451$, and $H = 1,000,000$



(c) $n = 10,000$, $r = 4805$, and $H = 1,000,000$

Figure 42: Write operation comparisons for Amazon EC2 p3.2xlarge with Intel Xeon E5-2686v4 processor, 488GB DDR3 RAM, and NVIDIA Tesla V100 GPU.

10

RESULTS (VI): SUPERVISED CLASSIFICATION APPLICATION

Supervised classification problem consists of categorize data into groups after seeing some samples from each group. First, it is presented pieces of data with their categories. The algorithm learns from these data, which is known as learning phase. Then, new pieces of data are presented and the algorithm must classify them into the already known groups. It is named supervised because the algorithm will not create the groups itself. It will learn the groups from during the learning phase, in which the groups have already been defined and the pieces of data have already been classified into them.

Although this problem has already been studied (REF), our intention here is to show that a pure SDM may also be used to classify data. Fan and Wang [10] has used SDM to solve a classification problem, recognizing handwriting letters from images, but he used a mix of genetic algorithm with SDM, which is very different from the original SDM described by [15]. Even though his algorithm has classified properly, we were intrigued whether a pure SDM would also classify successfully.

Hence, we have developed a supervised classification algorithm based on a pure SDM as our main memory. Our goal was to classify noisy images into their respective letters (case sensitive) and numbers. For some examples, see Figure 43.

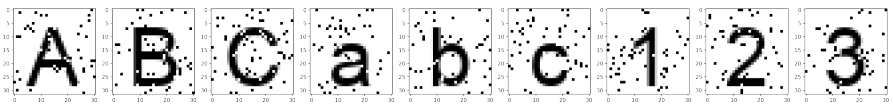


Figure 43: Examples of noisy images with uppercase letters, lowercase letters and numbers.

The images had 31 pixels of width and 32 pixels of height, totaling 992 pixels per image. Each image was mapped into a 1,000 bit bitstring in which the bits were set according to the color of each pixel of the image. So, white pixels were equal to bit 0, and black pixels to bit 1. The 8 remaining bits were all set to zero. This was a bijective mapping (or one-to-one mapping), i.e., there was only one bitstring for each image, and there was only one image for each bitstring.

A total of 62 classification groups have been trained in the SDM. For each of them, it was generated a random bitstring. Thus, the groups' bitstrings were orthogonal between any two of them. There is one

image for each of the 62 groups in Figure 44. Notice that the SDM has never seen a single image with no noise.

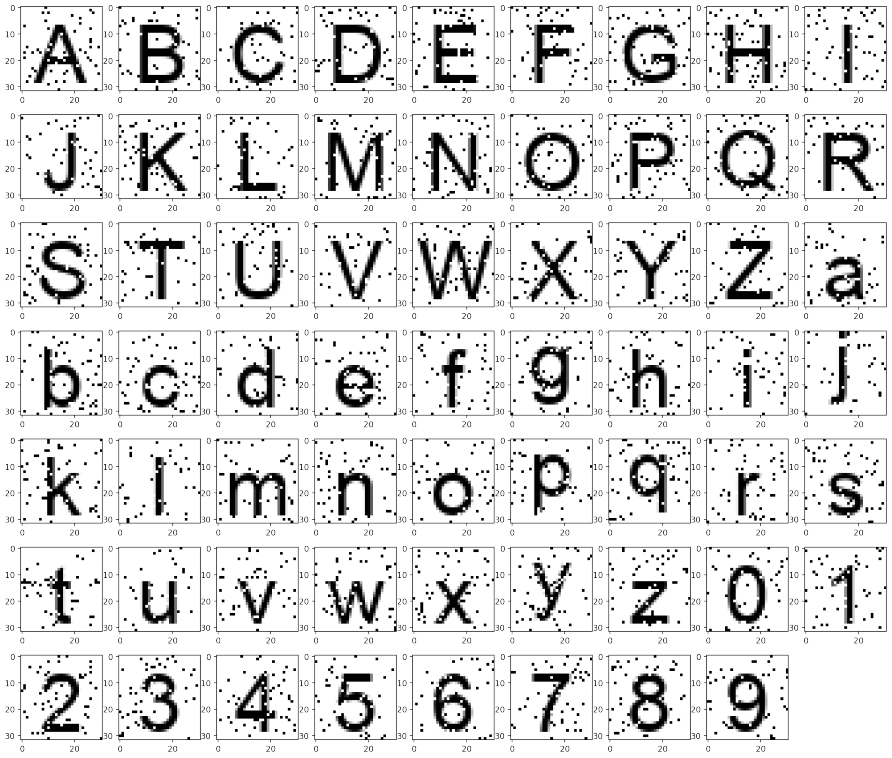


Figure 44: One noisy image for each of the 62 classification groups.

The association of images to groups was stored as sequences in SDM, as detailed by Kanerva [15] in Chapter 8. During the learning phase, the image bitstrings were stored pointing to their groups bitstrings, i.e., write(addr=bs_image, datum=bs_label). Thus, in order to classify an unknown image, we only had to read from its address and check which group has been found.

During the learning phase, we have generated 100 noisy images for each character. The images had 5% of noise, i.e., 5% of their pixels have been randomly flipped. For example, see the generated images for letter A in Figure 45. Then, we have wrote the classification group bitstring into the bitstring associated to each noisy image, i.e., write(bs_image, bs_label). For a complete image training set, see Appendix XYZ.

Finally, we have assess the performance of our classifier. We had done it in three different scenarios: high noise (20%), low noise (5%) and no noise. See Figures 46 and 47 for images with 20% noise and no noise. The low noise scenario had the same noise as the training set. For each scenario, we had classified 620 unknown images with 10 images per group.

The performance was calculated as the percentage of hits for each group. We did not expected the same performance for all groups

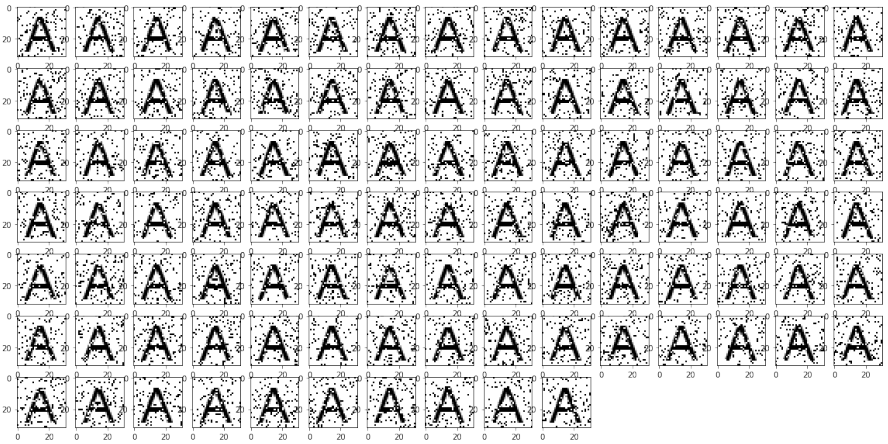


Figure 45: 100 noisy images generated to train label A.

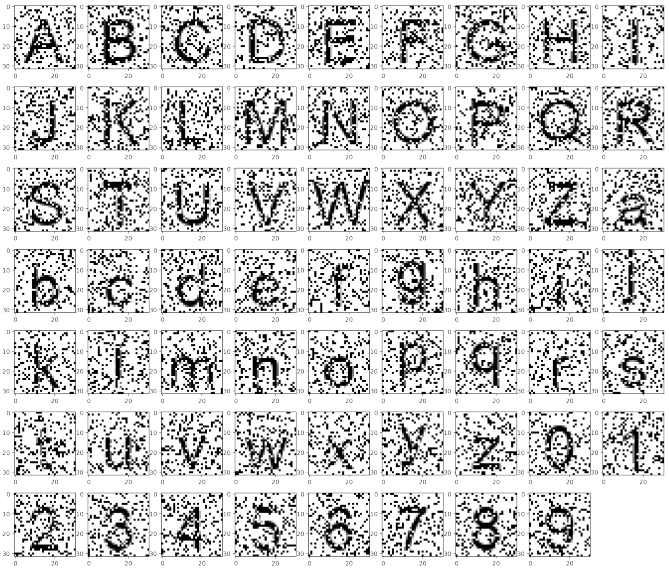


Figure 46: Images generated using a 20% noise for the high noise scenario.

because some groups become very similar to other depending on the noise level, and this similarity may even confuse a person (see Figure 48).

In the no noise scenario, the classifier has hit all characters, except letter "l" which was wrongly associated to the group of "i". We believe that it happened because the classifier had never seen an image with no noise and the difference between the images of "l" and "i" is smaller than the critical distance. So, both groups have been merged and it would converge to only one of them. In our simulation, it happened to be the group of "i".

In the low noise scenario, it has made few mistakes. It correctly classified all images but some from characters "b", "e", "f", "l", "t", and "9". It completely classified "l" images to the "i" group. In the

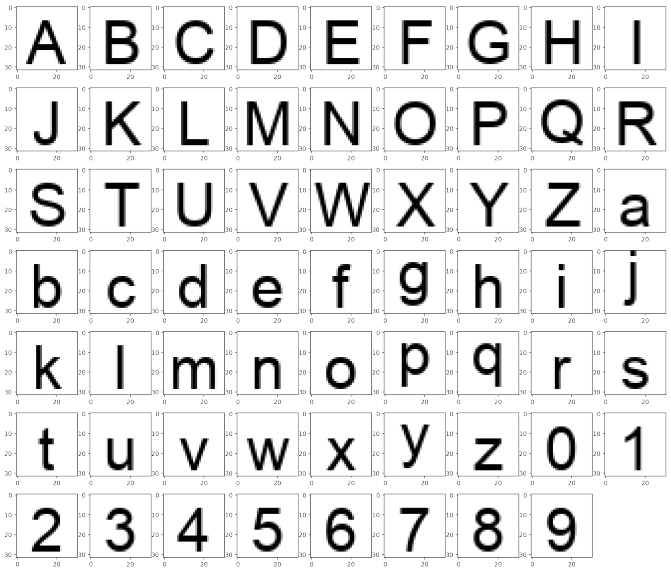


Figure 47: Images generated for the no noise scenario.

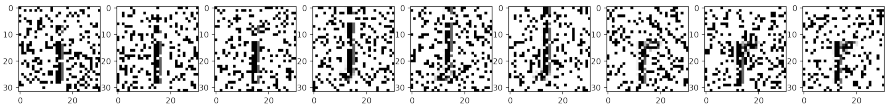
other cases, it made just a few mistakes. See Figure 49 to check the images and their classification.

The high noise scenario is the most interesting, because, even in a high noise level, the classifier has hit most of the characters. It has hit all images for 44 out of 62 groups, and made at least one miss for the other 18 groups. The misses may be seen in details in Figure **??**.

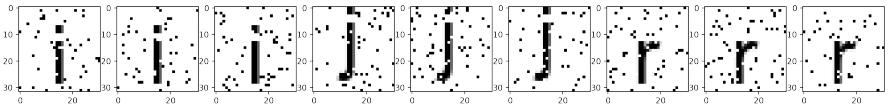
The critical distance plays an important role in the classification error. As we have 62 groups and each have been trained with 100 images, there were 6,200 writes to the memory. When an image is being classified, it will have to converge to a group, and the convergence depends on the distance between this image and the images from the training set, i.e, in the noise level.

In our simplified scenario, there is neither translation nor rotation. Future work may explore how sensible this classification algorithm is to these operations. We expect that, with proper training, the algorithm will remain classifying the images with a good hit rate.

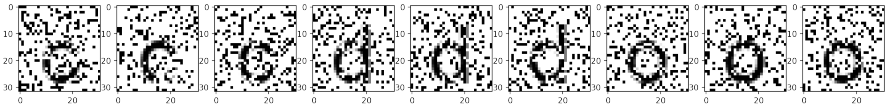
These results show that the SDM may be used as a supervised classification algorithm. Although we do not believe that the mapping between images and bitstrings are even close to the way human cognition deals with images, we believe the results are interesting and useful to many possible real world problems.



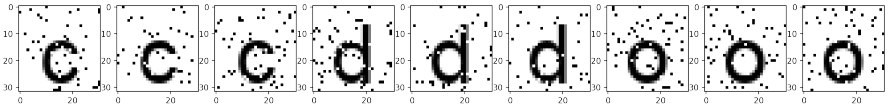
(a) “i”, “l”, and “r” with 20% noise.



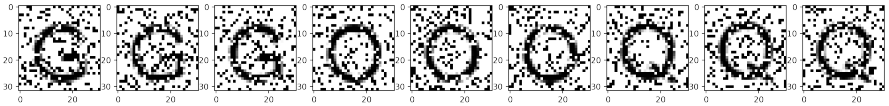
(b) “i”, “l”, and “r” with 5% noise.



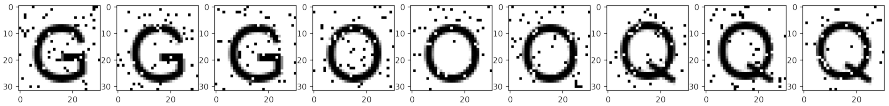
(c) “c”, “d”, and “o” with 20% noise.



(d) “c”, “d”, and “o” with 5% noise.

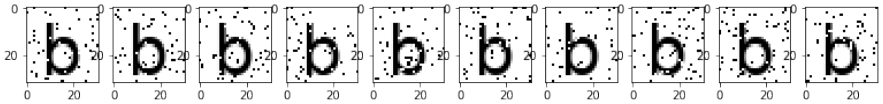


(e) “G”, “O”, and “Q” with 20% noise.

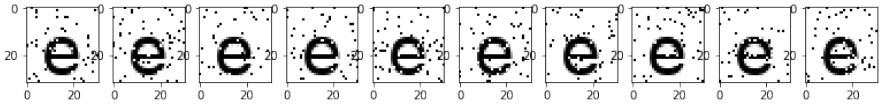


(f) “G”, “O”, and “Q” with 5% noise.

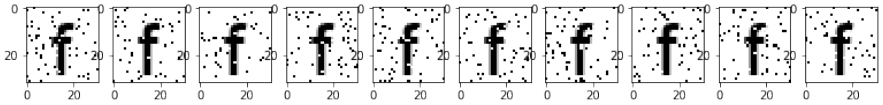
Figure 48: Images of different characters which may be confusing depending on the noise level.



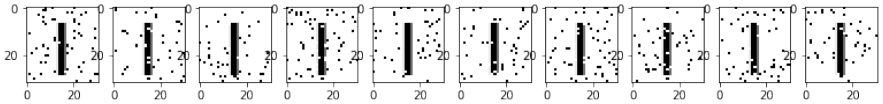
(a) Images from character "b which were classified as [b, b, b, h, b, o, b, h, b, b], respectively. It has made 3 misses.



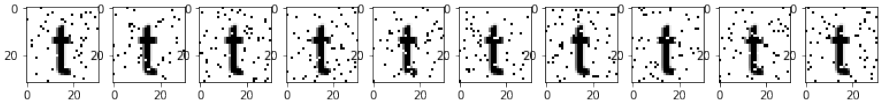
(b) Images from character "e which were classified as [e, e, e, e, e, e, e, e, o, e], respectively. It has made 1 miss.



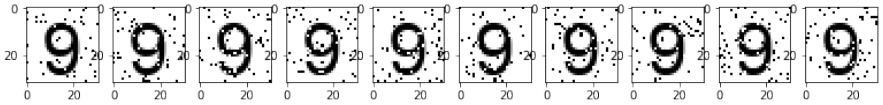
(c) Images from character "f which were classified as [i, f, f, I, I, I, f, f, f, f], respectively. It has made 4 misses.



(d) Images from character "l" which were classified as [i, i, i, i, i, i, i, i, i, i], respectively. It has missed them all, as if both groups have been merged.

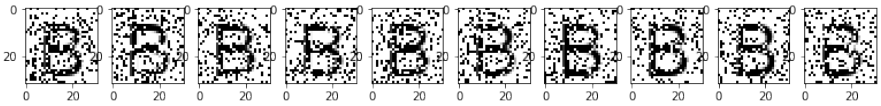


(e) Images from character "t" which were classified as [t, t, t, t, t, t, t, i, t, t], respectively. It has made 1 miss.

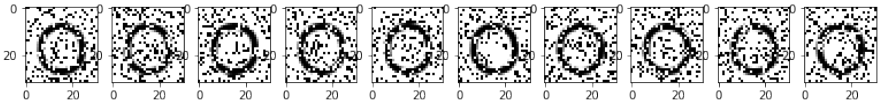


(f) Images from character "9" which were classified as [9, 9, o, 9, 9, 9, o, o, 9, 9], respectively. It has made 3 misses.

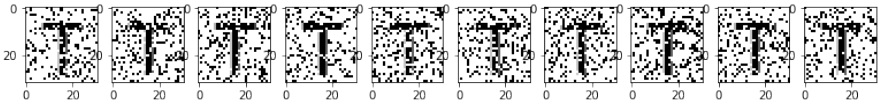
Figure 49: Characters in the low noise scenario in which the classifier has made at least one mistake. In all the other cases, it correctly classified the images. We may notice that the groups of "i" and "l" have been completely merged by the classifier, because it cannot distinguish them, not even with no noise.



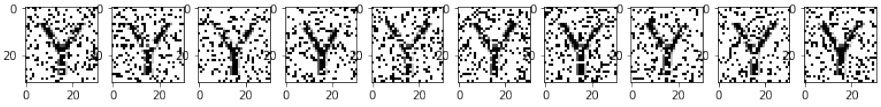
(a) Images from character “B” which were classified as [S, B, B, B, B, B, B, B, B, B]. It has made 1 mistake.



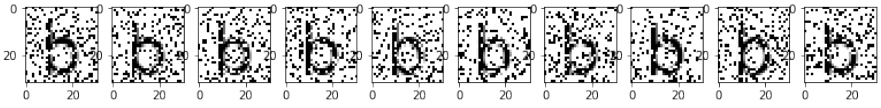
(b) Images from character “O” which were classified as [G, G, O, O, O, O, O, O, O, O]. It has made 2 mistakes.



(c) Images from character “T” which were classified as [T, T, T, T, T, I, T, T, T, T]. It has made 1 mistake.



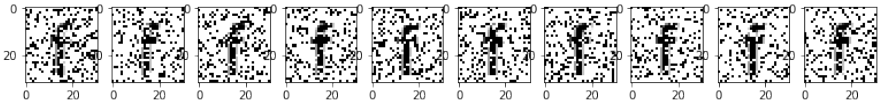
(d) Images from character “Y” which were classified as [Y, I, Y, Y, Y, Y, Y, Y, Y, Y]. It has made 1 mistake.



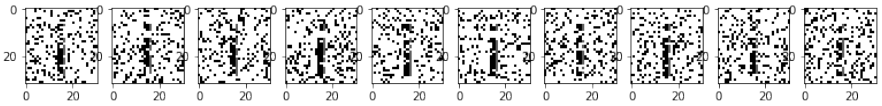
(e) Images from character “b” which were classified as [o, o, o, b, o, h, h, b, b, o]. It has made 7 mistakes.

(f) Images from character “c” which were classified as [c, c, c, c, c, o, c, c, c, o]. It has made 2 mistakes.

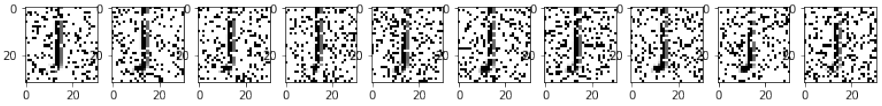
(g) Images from character “e” which were classified as [e, o, e, o, o, o, e, o, o, e]. It has made 6 mistakes.



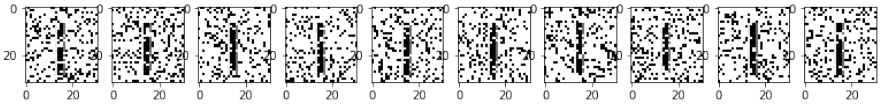
(h) Images from character “f” which were classified as [I, I, I, I, i, I, I, I, I, I]. It has missed them all.



(i) Images from character "i" which were classified as [i, i, i, I, i, i, i, i, I, i]. It has made 2 mistakes.

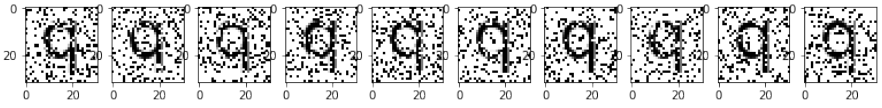


(j) Images from character "j" which were classified as [j, j, j, I, I, j, j, j, I]. It has made 3 mistakes.

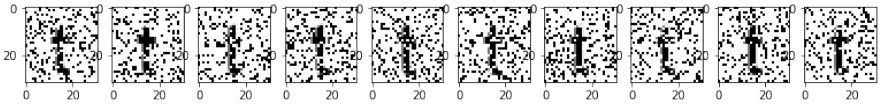


(k) Images from character "l" which were classified as [I, i, I, I, I, I, i, I, I, i]. It has missed them all.

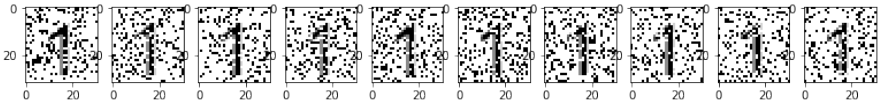
(l) Images from character "n" which were classified as [u, n, n, n, n, n, u, u, u, h]. It has made 5 mistakes.



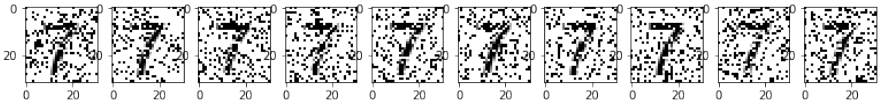
(m) Images from character "q" which were classified as [q, q, q, q, q, q, q, q, q, g]. It has made 1 mistake.



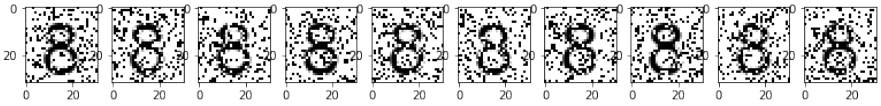
(n) Images from character "t" which were classified as [I, r, I, i, I, i, i, I, i]. It has missed them all.



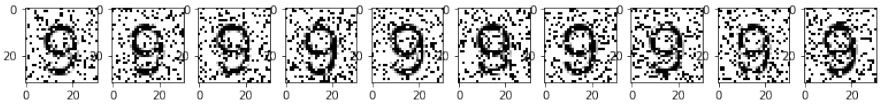
(o) Images from character "1" which were classified as [1, I, 1, I, 1, 1, I, I, 1, I]. It has made 5 mistakes.



(p) Images from character "7" which were classified as [7, 7, 7, I, 7, I, I, 7, 7, 7]. It has made 3 mistakes.



(q) Images from character "8" which were classified as [8, 6, 6, 6, 8, d, 8, 8, d, 6]. It has made 6 mistakes.



(r) Images from character "9" which were classified as [9, 0, 6, 0, 9, 0, 0, 9, 0, 0]. It has made 7 mistakes.

Figure 48: Characters in the high noise scenario in which the classifier has made at least one mistake. In all the other cases, it correctly classified the images.

11

RESULTS (VII): IMAGE NOISE FILTERING APPLICATION

Image noise filtering consists in removing the noise from an input, in out case an image. Our images are black & white images and the noise is generated randomly flipping some of their pixels from black to while and vice versa. In Figure 49, we may see an image with different levels of noise, from 0% to 45% in steps of 5%. It makes no sense to apply 50% of noise because it would absolutely randomize the image.

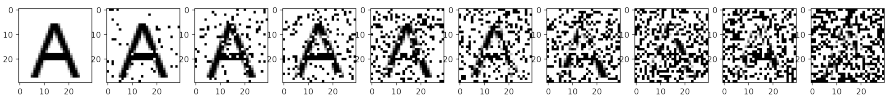


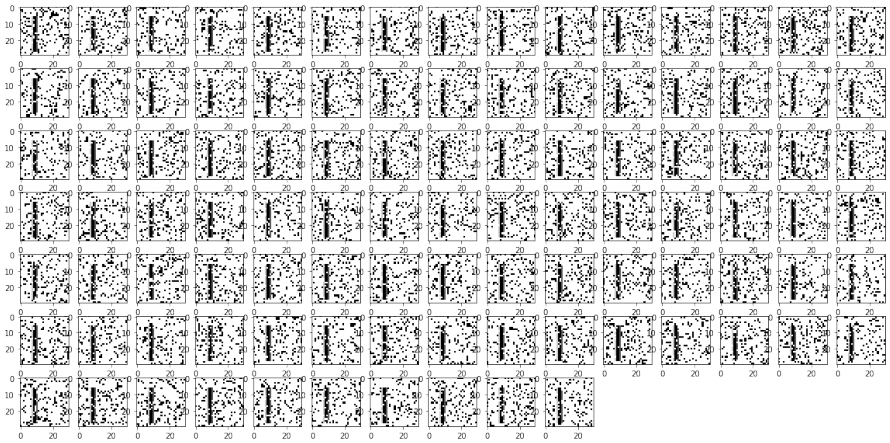
Figure 49: Progressive noise into letter "A", from 0% to 45% in steps of 5%.

The images have 30 x 30 pixels, totaling 900 pixels per image. Each image is mapped into a bistrings of 1,000 bit in which the bits are set according to the color of each pixel of the image. White pixels are mapped to bit 0, and black pixels to bit 1. The 100 remaining bits are all set to zero. This is a bijective mapping (or one-to-one) from images and bitstrings, i.e., there is one, and only one, bitstring for each image, and vice versa.

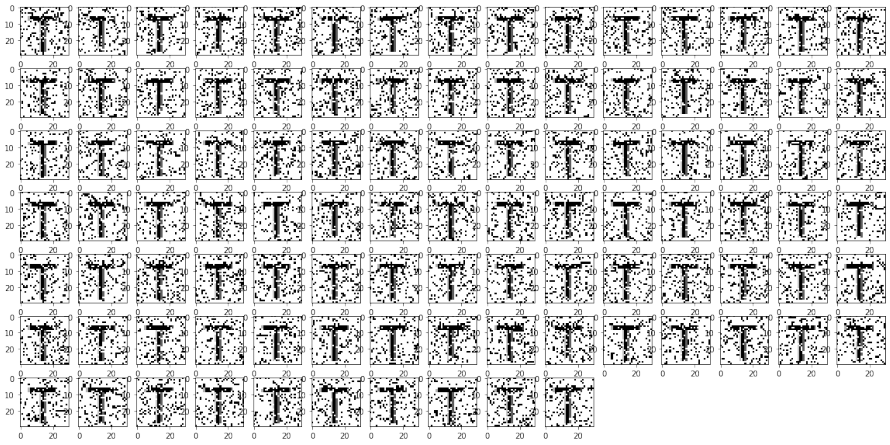
In the learning phase, 200 noisy images were generated and written into SDM. Half of the letter "I" and half of the letter "T" (see Figure 50). They were written into their own addresses, i.e., `write(address=bs_image, datum=bs_image)`.

Then, in order to test the filtering, we just have read from noisy images, and the results were really impressive. We were able to clean images up to 42% of noise (see Figure 51). While SDM has never seen a clean version of the letters, it just learned from the learning phase which pixels has appeared more frequently and chose them.

A simplified mathematical analysis would be: During the learning phase, 200 images with 15% of noise were written to SDM, so, the average distance between them and the clean image was 150 bits. Thus they shared, on average, 175 hard-locations with the clean image. In these 175 hard-locations, the counter's value for a black pixel mapped to bit 1 was $(1 - 0.15) \cdot 200 - 0.15 \cdot 200 = 140$. Finally, let's analyse the reading. When reading from a noise image with 42% of noise, the average distance between the noisy image and its clean image is 420, which means they share, on average, 6 hard-locations. As the average number of activated hard-locations is 1,072, the sum of their counters will be, on average, $Y = 6 \cdot 140 - \sum_{i=1}^{1072-6} X_i$,

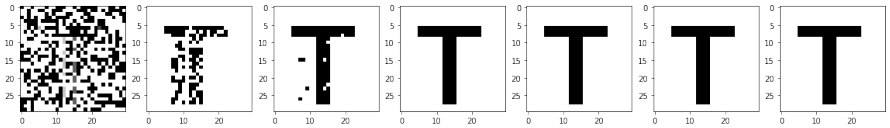


(a) Letter "I" with 15% of noise.

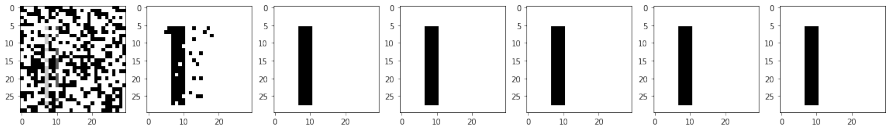


(b) Letter "T" with 15% of noise.

Figure 50: Training images written into the SDM. They were written in their own addresses — write(address=bs_image, datum=bs_image).



(a) Steps of reading from letter "T" with 42% of noise



(b) Steps of reading from letter "I" with 42% of noise

Figure 51: In order to test the SDM as a noise filter, we read from noisy images expecting to get a clean image. It is interesting to highlight that SDM has never seen a clean version letters "T" and "I".

where X_i is a Bernoulli trial with probability 0.5. Hence, $P(\text{black pixel}) = P(Y > 0) = P(6 \cdot 140 - \sum_{i=1}^{1072-6} X_i > 0) =$

$P(\sum_{i=1}^{1066} X_i < 840) = 99.99\%$. But, when reading from a noise image with 45% of noise, the average number of hard-locations shared with the clean image is only 3. Thus the sum of the activated hardlocations' counters will be, on average, $3 \cdot 140 - \sum_{i=1}^{1072-3} X_i$, and $P(\text{black pixel}) = P(\sum_{i=1}^{1069} X_i < 420) = 1.28 \cdot 10^{-12}$. The probability abruptly drops from 100% to 0% when the noise goes from 42% to 45% (see Figure 52). The analysis for while pixels is exactly the same, but with opposite signs. The code to calculate this probability is available in "Noise filter - Math analysis" notebook [3].

This is a simplified analysis because it does not take into consideration the hardlocations shared by the different letters. It works fine for our example because letter "I" and "T" are almost orthogonal and share, on average, only one hard location.

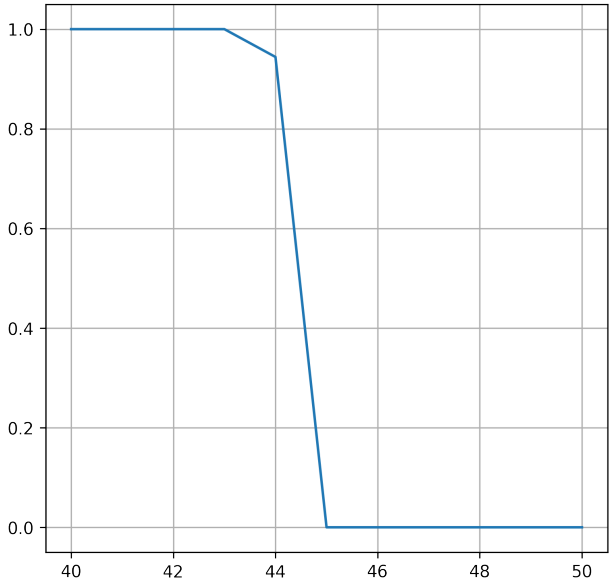
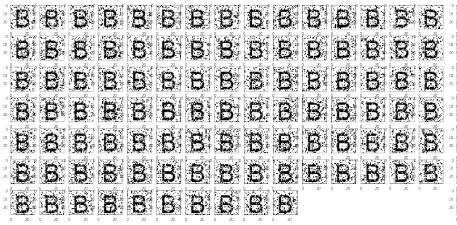


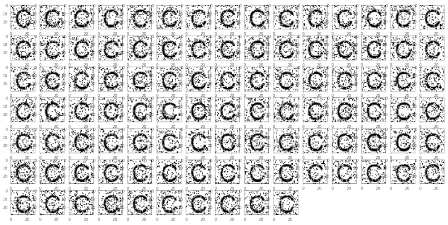
Figure 52: Probability of getting the right pixel when reading from an image with noise p. It assumes that SDM was trained with 200 images with 15% noise.

If the intersection between images becomes too high, the noise filter stops working properly. We have confirmed it writing letters "B", "C", "D" and numeral "8". They share a great amount of hard-locations and our noise filter could not filter their noise correctly. The training sets can be seen in Figure 53 and the results in Figure 54.

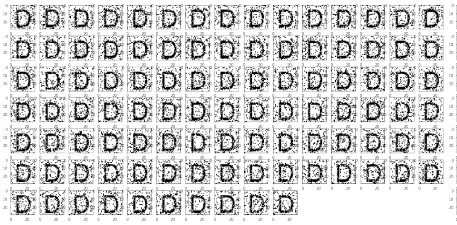
A possible solution to this interference problem is to use labels. Each label has a random bitstring, which will be chuncked with the images before writing into SDM. Hence, before reading, we also have to chunck the image with the label — which means we need to know the label of each image. The chunk was done using the exclusive OR (XOR) operator, i.e., $\texttt{bs\_chunck} = \texttt{bs\_image} \oplus \texttt{bs\_label}$. In other words, we run `write(address=bs_chunk, datum=bs_label)` during the



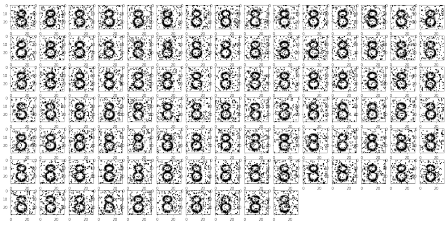
(a) Letter “B” with 15% of noise.



(b) Letter “C” with 15% of noise.

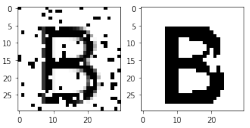


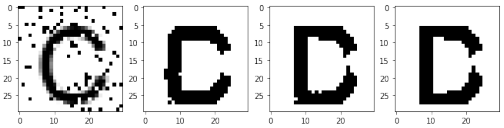
(c) Letter “D” with 15% of noise.

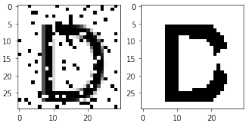


(d) Letter “8” with 15% of noise.

Figure 53: Training images in which the intersection between images is too high. They were written in their own addresses — write(address=bs_image, datum=bs_image).







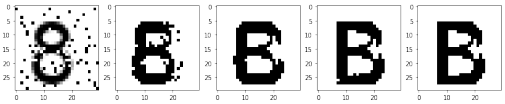


Figure 54: When the intersection between images becomes too high, there appears some interference in the resulting image. All cases have 10% noise. We can notice that the empty space in the right side of the “C” letter generates some white pixels in the right side of both “B” and “D” letters.

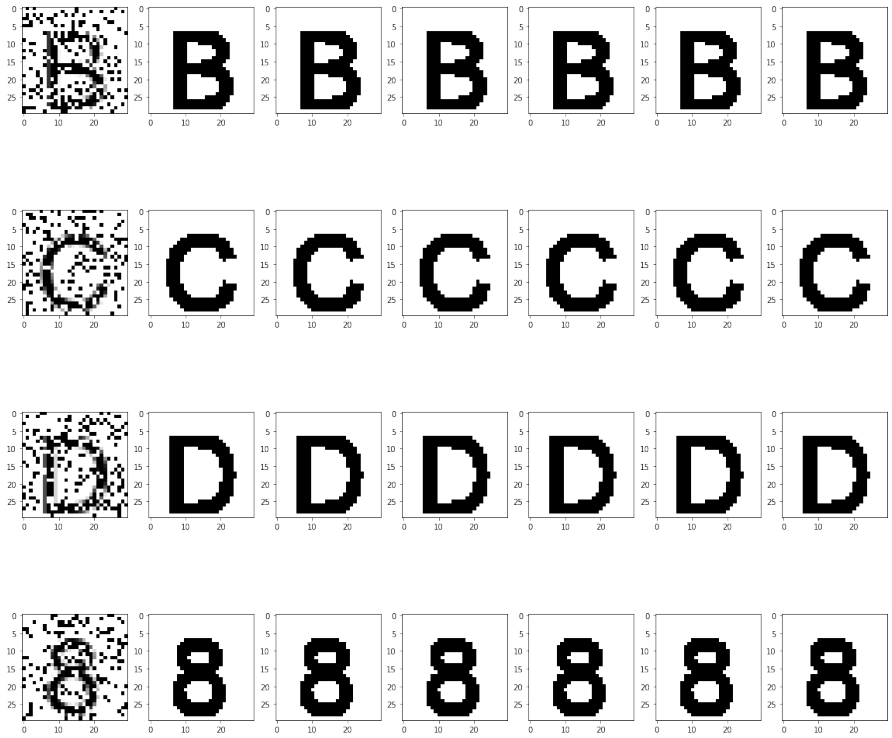


Figure 55: Using labels solves the interference problem when the intersection between images becomes too high. All cases have 20% noise.

training, and `read(address=bs_chunk)` during the testing. We used the same training set as before, and the results can be seen in Figure 55.

The chunck through exclusive OR (XOR) works because of Theorem 1, which says that chuncking the images with labels will generate, on average, orthogonal bitstrings. Thus, these orthogonal bitstrings will not interfere with each other because they share, on average, only one hardlocation.

The disadvantage of using labels is that it requires classification of the images. In our example, we just used the correct label with each image, but we could have used our classification algorithm as a pre-processing step, and only then run the noise filter.

Theorem 1. *If* ν_1 *and* ν_2 *are random bitstrings, then* $\forall a, b, \mathbf{E}\left[d(a \oplus \nu_1, b \oplus \nu_2)\right] = n/2$.

Proof. Let $\mathbb{A} = \{i | a^i = b^i\}$ be the indexes in which the bits of a are equal to the bits of b, and $\mathbb{B} = \{i | a^i \neq b^i\}$ be indexes in which the bits of a are different from the bits of b. Thus,

$$\begin{aligned} d(a \oplus v_1, b \oplus v_2) &= \sum_{i=1}^{n} d(a^i \oplus v_1^i, b^i \oplus v_2^i) \\ &= \sum_{i=1}^{n} (a^i \oplus v_1^i) \oplus (b^i \oplus v_2^i) \\ &= \sum_{i=1}^{n} (a^i \oplus b^i) \oplus (v_1^i \oplus v_2^i) \\ &= \sum_{i \in \mathbb{A}} (a^i \oplus b^i) \oplus (v_1^i \oplus v_2^i) + \sum_{i \in \mathbb{B}} (a^i \oplus b^i) \oplus (v_1^i \oplus v_2^i) \end{aligned}$$

For $i \in \mathbb{A}$, $a^i \oplus b^i = 0$, and follows:

$$\begin{aligned} (a^i \oplus b^i) \oplus (v_1^i \oplus v_2^i) &= 0 \oplus (v_1^i \oplus v_2^i) \\ &= v_1^i \oplus v_2^i \\ &= d(v_1^i, v_2^i) \end{aligned}$$

Hence, $\mathbf{E}\left[\sum_{i \in \mathbb{A}} d(a^i \oplus v_1^i, b^i \oplus v_2^i)\right] = \mathbf{E}\left[\sum_{i \in \mathbb{A}} d(v_1^i, v_2^i)\right] = |\mathbb{A}|/2$, because v_1 and v_2 are random bitstrings and their average distance is half the number of bits.

For $i \in \mathbb{B}$, $a^i \oplus b^i = 1$, and follows:

$$\begin{aligned} (a^i \oplus b^i) \oplus (v_1^i \oplus v_2^i) &= 1 \oplus (v_1^i \oplus v_2^i) \\ &= d(1, v_1^i \oplus v_2^i) \end{aligned}$$

Hence,
$\mathbf{E}\left[\sum_{i \in \mathbb{B}} d(a^i \oplus v_1^i, b^i \oplus v_2^i)\right] = \mathbf{E}\left[\sum_{i \in \mathbb{B}} d(1, v_1^i \oplus v_2^i)\right] = |\mathbb{B}|/2$.

Finally, $\mathbf{E}\left[d(a \oplus v_1, b \oplus v_2)\right] = |\mathbb{A}|/2 + |\mathbb{B}|/2 = n/2$, since $|\mathbb{A}| + |\mathbb{B}| = n$ □

A final note is in order. In no way is it claimed here that these bitstrings should form a plausible representation of letters in the human mind (See, for instance, Hofstadter [14] for a marvelous discussion of the subtleties and fluidity involved in that process). These letters should merely be seen as *invariant patterns* that are processed by the memory, which is able to capture their underlying invariant structure even when presented only under heavy noise.

12

RESULTS (VIII): THE POSSIBILITY OF UNSUPERVISED REINFORCEMENT LEARNING

Reinforcement learning has increasing prominence in the media after AlphaZero has won all games from both the best chess grandmasters in the world and the best chess engines. What is incredible about these victories is that AlphaZero has almost no knowledge about chess game and has learned all its movement playing against itself for 4 hours. Basically, it knowns only the valid movements and had to learn everything from scratch, which it did using a reinforcement learning algorithm.

Reinforcement learning is a machine learning algorithm which learns from the rewards of its actions. So, it receives the game state as input, it decides which action will be taken, and then it learns from the rewards of all the actions it has chosen. In theory, it learns after each reward feedback it receives, improving its decision over time and presenting intelligent behavior. A positive reward would indicate that the chosen action should be encouraged, while a negative reward would indicate the opposite. In some algorithms, there may be a neutral reward which would indicate that the chosen action was neither positive nor negative. How each type of reward should be handled depends on each algorithm.

We have done some experiments with an SDM as a memory for a TicTacToe player. Basically, it receives the current board state and returns which action should be played. In the end of the game, it receives the sequences of boards and the winner, and is supposed to learn from them.

In our approach, there were 9 possible actions: one for each cell of the TicTacToe game. Action 1 means playing in the first cell of the first row; action 2 means playing in the second cell of the first row; and so one. Figure 56 shows the TicTacToe board numbering and the link between each cell and its respective action.

Our algorithm to decide what should be played is very simple: it reads the current board from SDM and then it chooses the valid action with highest score. To calculate the scores, the bitstring is split into 9 parts, one per action (Figure 56). The number of bits 1 in each part indicates the score of its respective action.

After a game has finished, it is time to learn from its decisions. Our algorithm has three rewards: positive, negative, and neutral. It always learn from both players, no matter who wins or if it was a draw. The winner's sequence of actions feed our positive reward learning. The

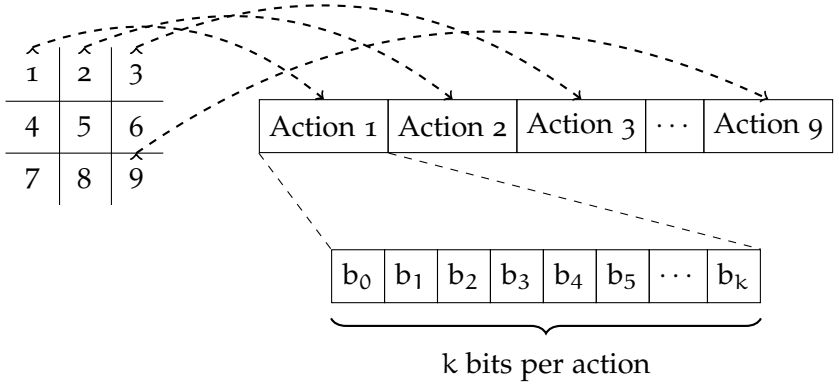


Figure 56: Each action is a cell in the TicTacToe board and is mapped to slice of the bitstring.

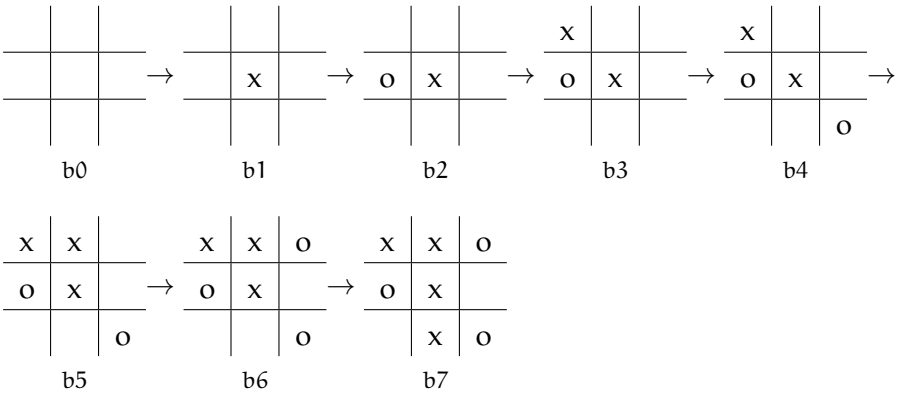


Figure 57: Example of a game with 7 movements in which X wins.

loser's sequence of actions feed our negative reward learning. If it is a draw, both sequences feed our neutral reward learning.

Each board state has a unique random bitstring. For instance, each board in Figure 57 has its own unique random bitstring. Thus, if a specific board state is reached again, SDM will return the scores of each action.

Let $b_0, b_1, b_2, \ldots, b_n$ be the board sequence of the game (see Figure 57). In order to learn, our algorithm will map each action which goes from b_k to b_{k+1} to a reward bitstring and then will write a pointer from b_k to this reward bitstring, i.e., `write(bs_board, bs_reward)`.

The reward bitstring may be a positive, negative, or neutral bitstring. The positive reward bitstring is randomly generated and then only the bits related to the action will be set to 1 (see Figure **??**). The negative reward bitstring is also randomly generated and then only the bits related to the action will be set to 0. Finally, the neutral reward bitstring is only a random bitstring.

The idea behind these reward bitstrings is to increase the score of positive rewards (all bits set to 1), and to decrease the score of

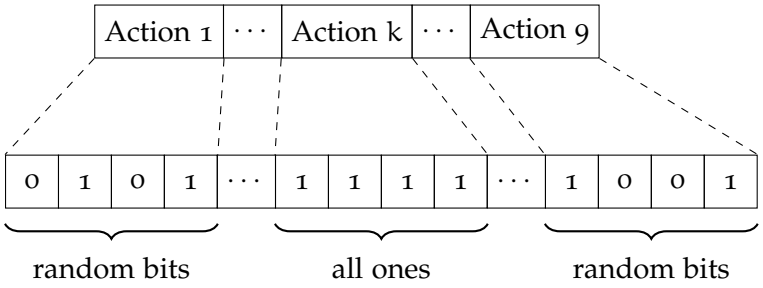


Figure 58: Positive reward bitstrings used in our reinforcement learning algorithm.

negative rewards (all bits set to 0). The neutral reward will have, on average, half of its bits 1 and the other half 0. Thus, it is in the middle between a positive reward and a negative reward.

Let the b be the sequence of boards and a be the sequence of actions, then $b_0 \xrightarrow{a_0} b_1 \xrightarrow{a_1} b_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} b_n$. Suppose there is a winner, thus the winner's actions will be the sequence $a_{n-1}, a_{n-3}, a_{n-5}, \ldots$, while the loser's actions will be the sequence $a_{n-2}, a_{n-4}, a_{n-6}, \ldots$.

The positive reward learning will be writing the following pointers in SDM: $b_{n-1} \to a_{n-1}$, $b_{n-3} \to a_{n-3}$, and so on. The reward bitstring of a_{n-1} will have all bits set to one in a_{n-1}'s slice. All other bits will be random.

The negative reward learning will be writing the following pointers in SDM: $b_{n-2} \to a_{n-2}$, $b_{n-3} \to a_{n-3}$, and so on. The reward bitstring of a_{n-2} will have all bits set to zero in a_{n-2}'s slice. All other bits will be random.

If it is a draw, all actions will be mapped to the neutral reward bitstring, which is simply a random bitstring.

There is also weights associated to positive, negative, and neutral reward. They are used to indicate what goal is more important. For instance, if SDM should try to win in the first place, no matter if it may lead to losing, the weight of the positive reward should be higher than the others. But, if it is more important to not lose, then the weight of the negative reward should be higher.

In the very beginning, SDM is empty and its counters are zeroed. So, any reading will result in a random bitstring, because it will flip a coin for all counters. Thus, the chosen action will be random, as their scores will follow a binomial distribution. It is exactly the desired behavior — we play randomly until we learn. In fact, it will happen every time an unknown board is seen.

Internally, every board is mapped into a random bitstring and passed to SDM. As every two boards are, on average, orthogonal, SDM knows nothing about the boards themselves, neither whether they are consecutives or not. It knowns only the score of the actions

according to the games it has seen and learned from. Hopefully, the actions will lead to a victory or a draw.

In more details, the next movement decision consists in one read from SDM, resulting in a bitstring. Then the scores of the actions are calculated counting the number of 1 in each part of the bitstring. The chosen action is the one which is valid and has the highest score.

12.1 TRAINING

Our algorithm learned playing games against opponents. We had four types of opponents: (i) another SDM player, (ii) a random player whose actions are always random; (iii) a smart player whose actions wins when it can, block the opponent when it can, or are random; and (iv) a human player.

The weights of the rewards were chosen to prevent losing. Thus the weight of the negative reward was 5, while the weight of the positive reward was 2 and the neutral was 1.

Every learn cycle had two parts: (i) 100 games learning and (ii) 100 games testing. So, it has never learned during the testing phase and has not affected the measure of the statistics.

12.2 RESULTS

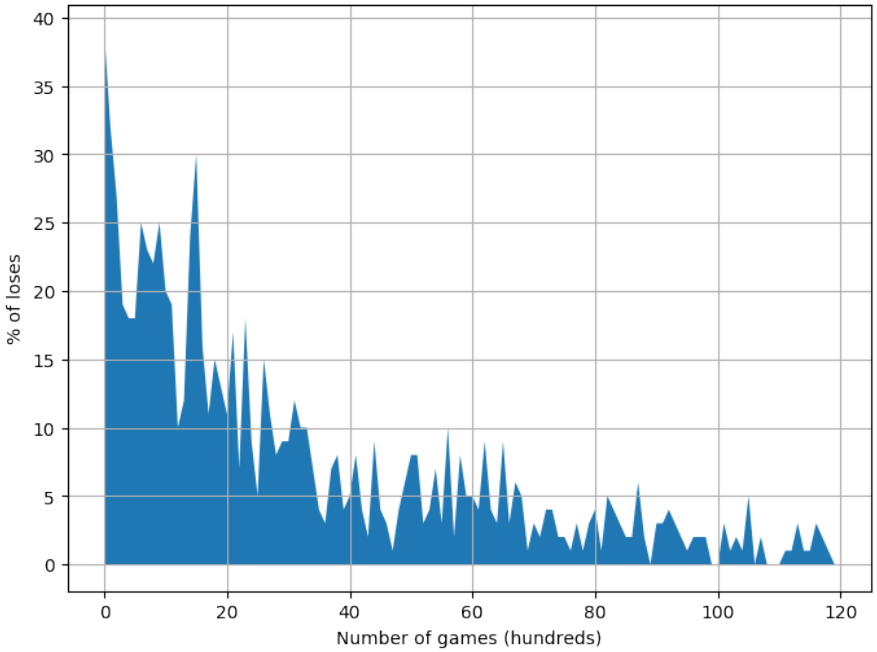
When playing against the random player, SDM has already started improving after the first 100 training games. Its winning rate converges quickly to around 80%, while the drawing rate starts to grow after 2,000 games and keeps growing until the end. The losing rate keeps decreasing until it reaches cycles of 100 games without any loss. See Figure 59.

When playing against smart player, SDM has started learning how to not lose during the first training cycles. The drawing rate grows quickly in the first 500 games and slowly since then. The winning rate grew to around 20% and remained there until 6,000 games, while the drawing rate kept growing. Then, after learning how to not lose, SDM started learning how to win, since after 7,000 games, the winning rate started to increase. See Figure 60.

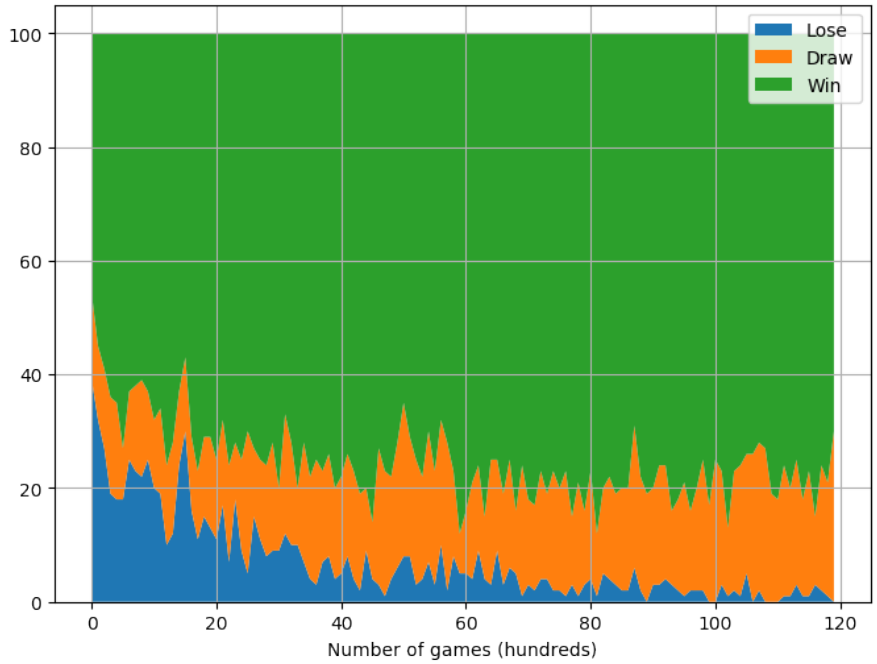
When playing against another SDM player, both player quickly learns how to not lose. During the first 100 testing games, without any learning game, they have behaved like two random players. After the 100 training games, they have learned how to not lose and the drawing rate grows quickly, reaching 100%. See Figure 61.

When playing against mixed players, SDM has also adapted. In Figure 62, it has played 6,000 games against random player, smart player, and another SDM player. In each cycle, one of them was randomly chosen. The number of losses over time is decreasing, whereas the number of wins and draws change a lot. It is easy to

notice when the other SDM player was chosen, because all games in that cycle have draw.

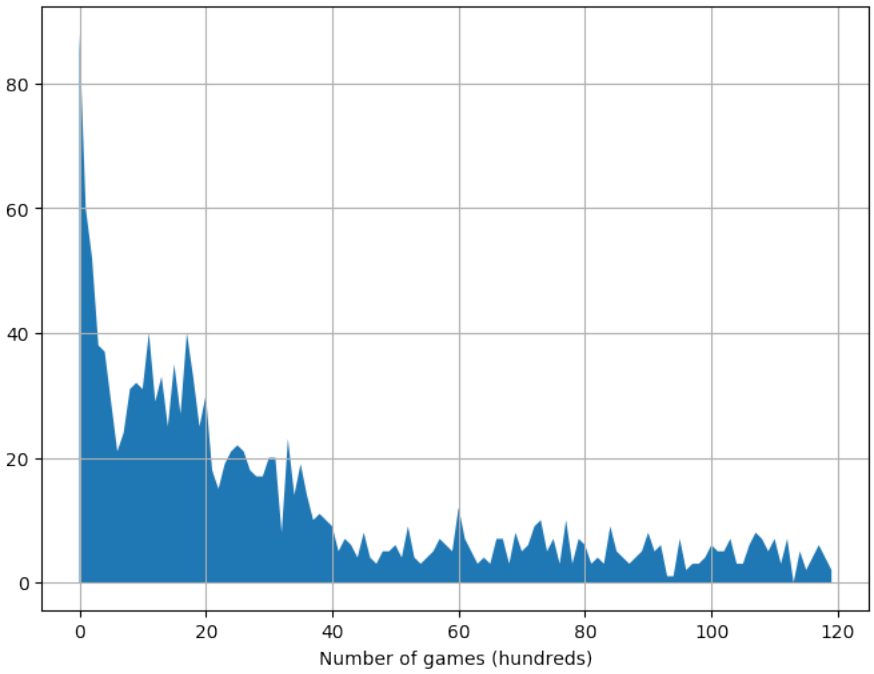


(a) Losses over time

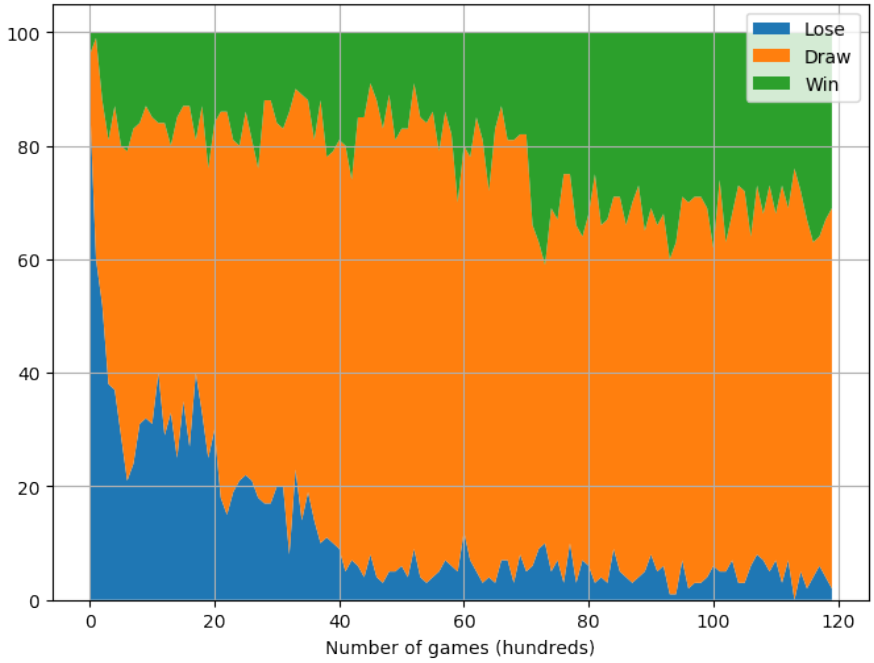


(b) Results over time

Figure 59: Results playing against random player. Each cyle was made of 100 games for training, and then 100 games for measuring statistics.

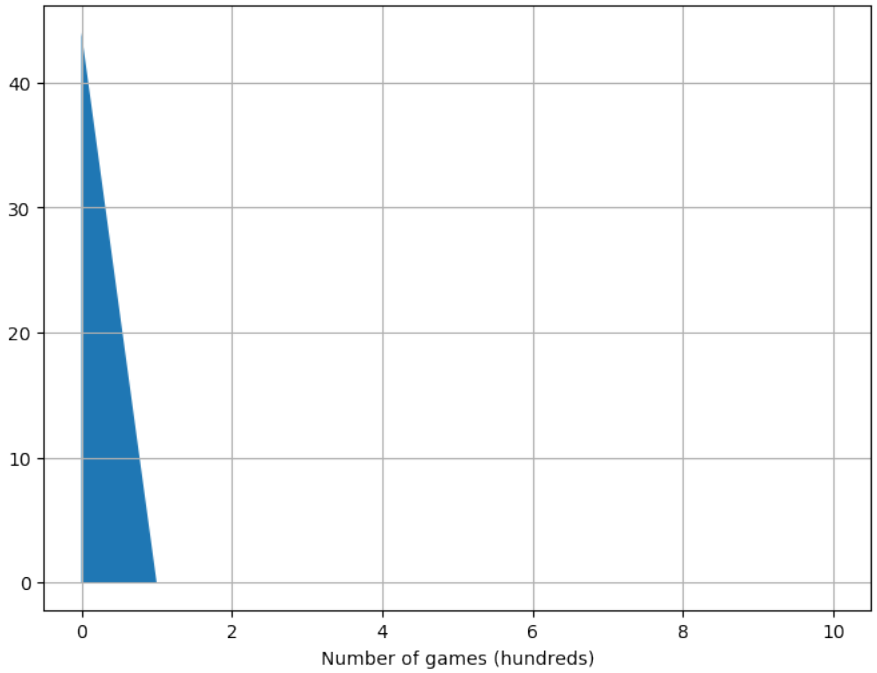


(a) Losses over time

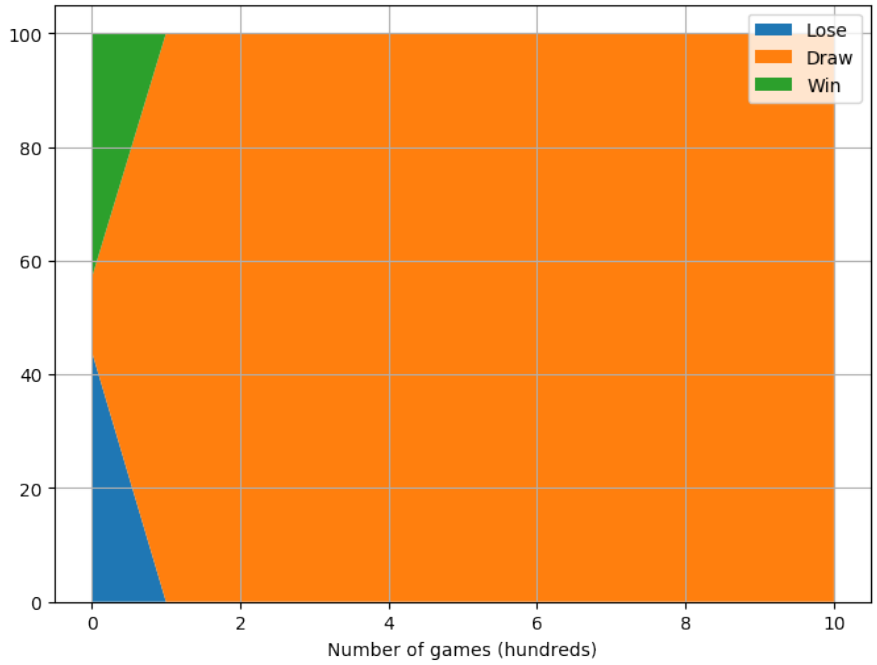


(b) Results over time

Figure 60: Results playing against smart player. Each cycle was made of 100 games for training, and then 100 games for measuring statistics.

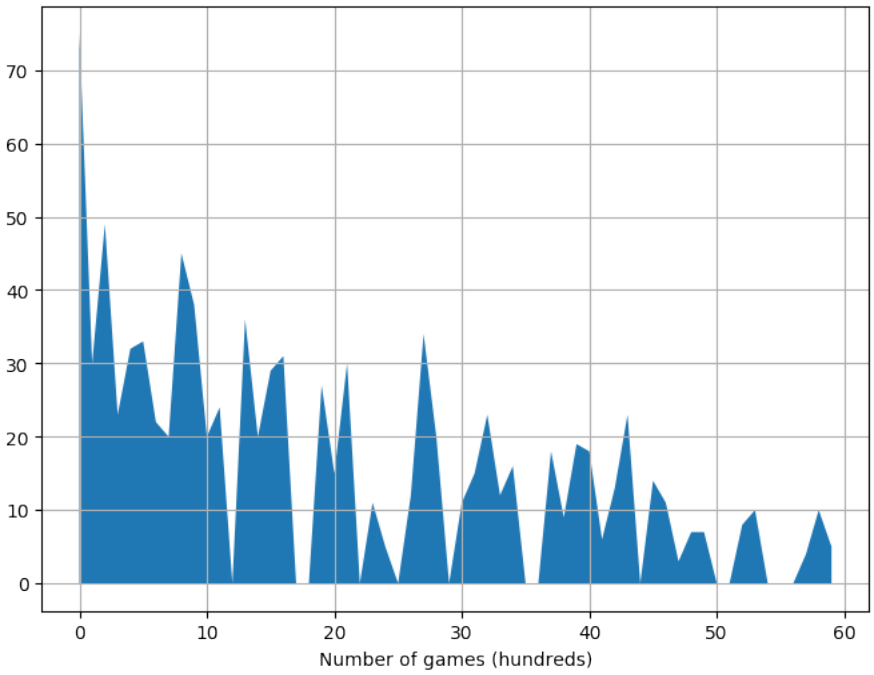


(a) Losses over time

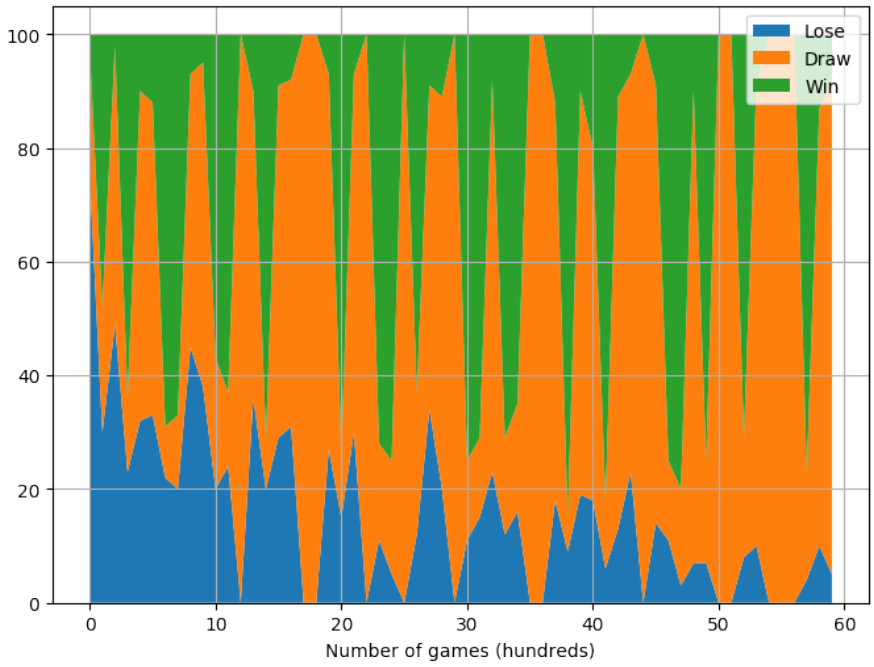


(b) Results over time

Figure 61: Results playing against another SDM player. Each cycle was made of 100 games for training, and then 100 games for measuring statistics.



(a) Losses over time



(b) Results over time

Figure 62: Results playing against a randomly chosen player between random player, smart player, and another SDM player. Each cycle was made of 100 games for training, and then 100 games for measuring statistics.

13

RESULTS (IX): INFORMATION-THEORETICAL WRITE OPERATION

My advisor, Alexandre Linhares, has proposed another write operation: an information-theoretical weighted write. In it, the sum of the counter's value is weighted based on the distance between each hard-location's address and the reading address. The logic behind it is to vary the importance of each hard-location inside the circle. It is only natural that one encodes an item in closer hard locations with a stronger signal, and a natural candidate for this signal function is the amount of information contained in the distance between the item and each hard location. Closer hard locations have lower probabilies and therefore should encode more information.

Consider the following. Information Theory [8] let us compute the precise amount of information in an event, when given its probability p, through the measure of *self-information*:

$$I(p) = -\log_2(p).$$

Now, given any two n-sized bitstrings, the probability of their Hamming distance being d is given by,

$p(H = d) = 2^{-n}\binom{n}{d}$

And the probability of it being at most d is

$$p(H \leqslant d) = 2^{-n}\sum_{i=0}^{d}\binom{n}{i},$$

and, consequently,

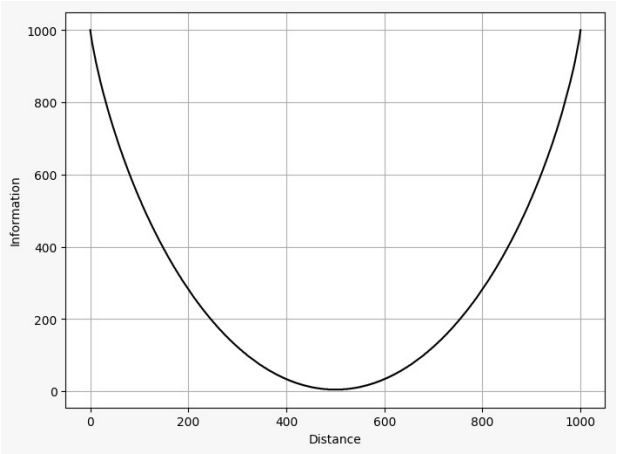
$$p(H \geqslant n-d) = 2^{-n}\sum_{i=n-d}^{n}\binom{n}{i},$$

$$p(d+1 \leqslant H \leqslant n-d-1) = 2^{n} - 2^{1-n}\sum_{i=0}^{d}\binom{n}{i}, \forall d < n/2.$$

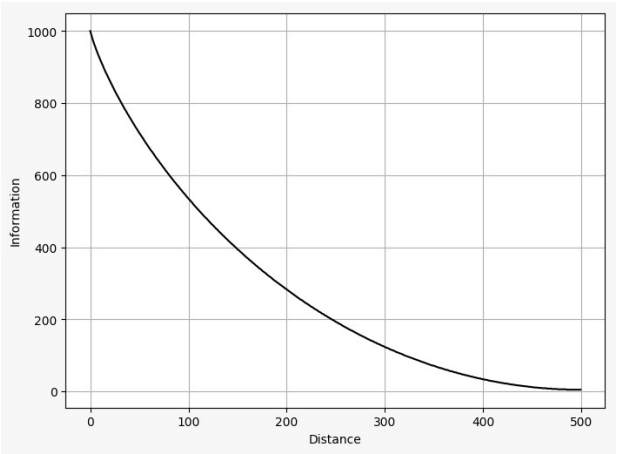
Hence the weighted write would, on each hard location, sum (or subtract) the following:

$w(d) = -\log_2\left(2^{-n}\binom{n}{d}\right) = n - \log_2\binom{n}{d}$, as seen in Figure 66.

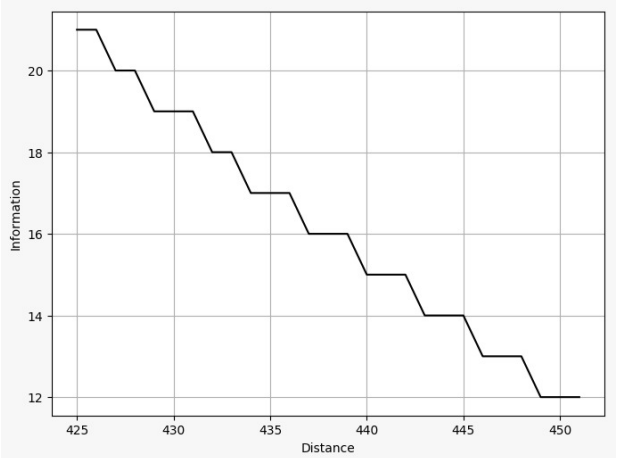
It is easy to interpret this data though a binary tree approach. How many binary questions would be needed to precisely define a bitstring?



(a) $w_1(d), d \in \{1, 2, ..., n\}$.



(b) $w_1(d)$ for the desired range.



(c) stepwise $\lfloor w_1(d) \rfloor$ for fast integer computation.

Figure 63: Shannon write operation: Computing the amount of information of a signal to each hard location in its access radius. (a) entirety of the space; (b) region of interest; (c) Fast integer computation is possible through a stepwise function.

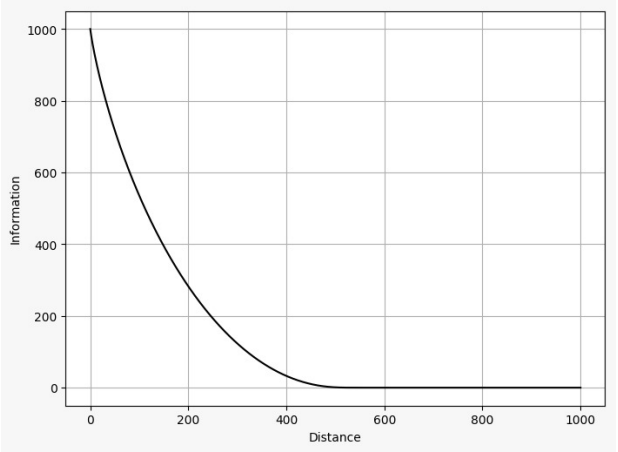
Another possibility would be to use the sum of all distances closer (and less likely) locations within the weighting function $w(d)$,

$$w(d) = -\log_2 \left(2^{-n} \sum_{i=0}^{d} \binom{n}{i} \right) = n - \log_2 \sum_{i=0}^{d} \binom{n}{i}.$$

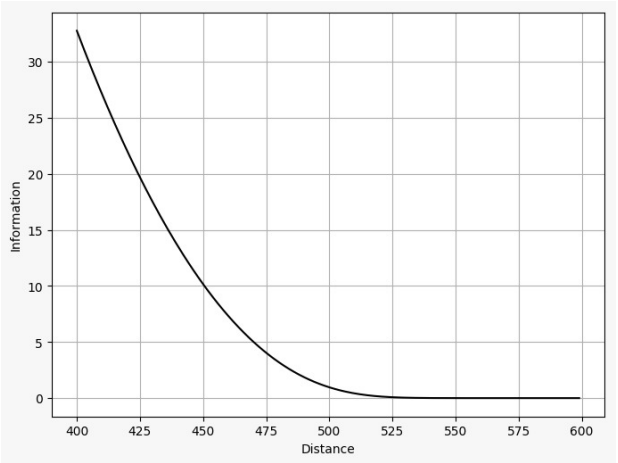
This can be seen in 64.

The initial results of this *Shannon write* operation can be seen in Figure 65 and seem promising. It seems that the critical distance increases by a number of bits. Note that 10 additional bits imply an attractor 2^{10} of the size of the original. Another point to keep in mind is that, since the modulus of the vectors are not uniform in this approach, that the shape of the attractor may have asymmetries.

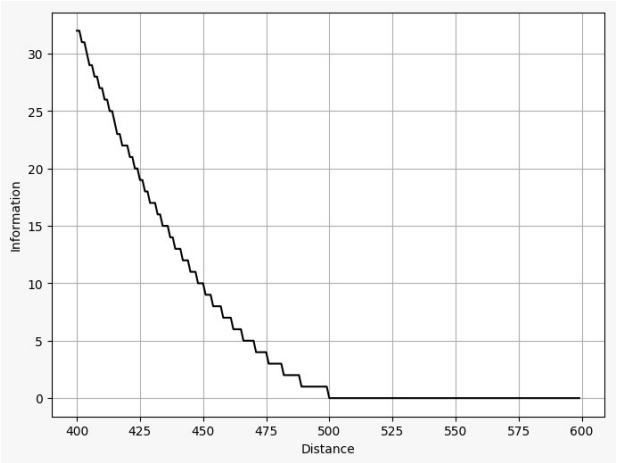
Note, finally, that this is not the first time in which a weighted function has been applied to writing in SDM — Hely et al. [13] suggest a rather complex spreading model based on floating point signals in the interval [0.05, 1.0] — they were, however, only able to test their model with 1,000 hard locations.



(a) $w_2(d), d \in \{1, 2, ..., n\}$.

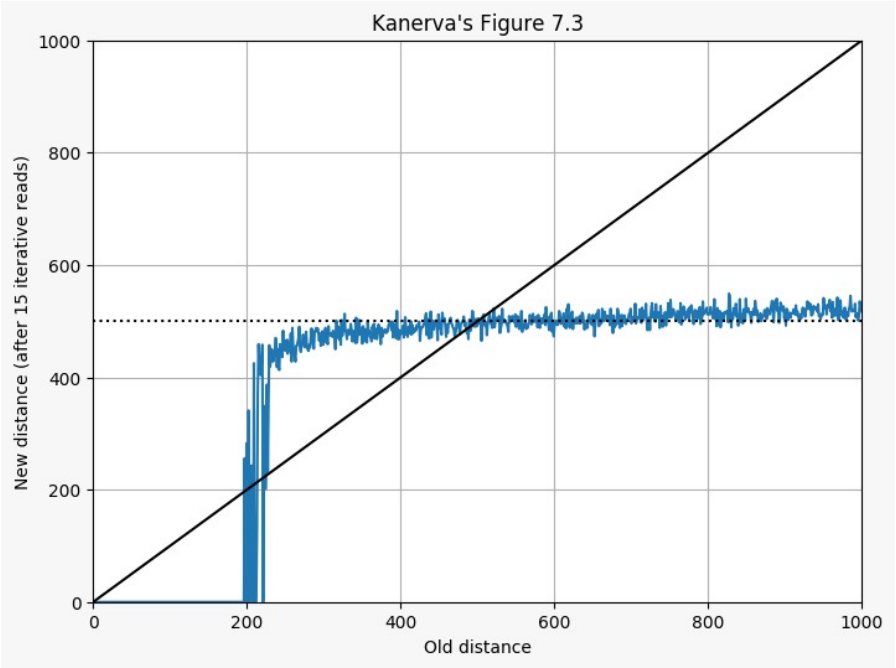


(b) $w_2(d)$ for the desired range.

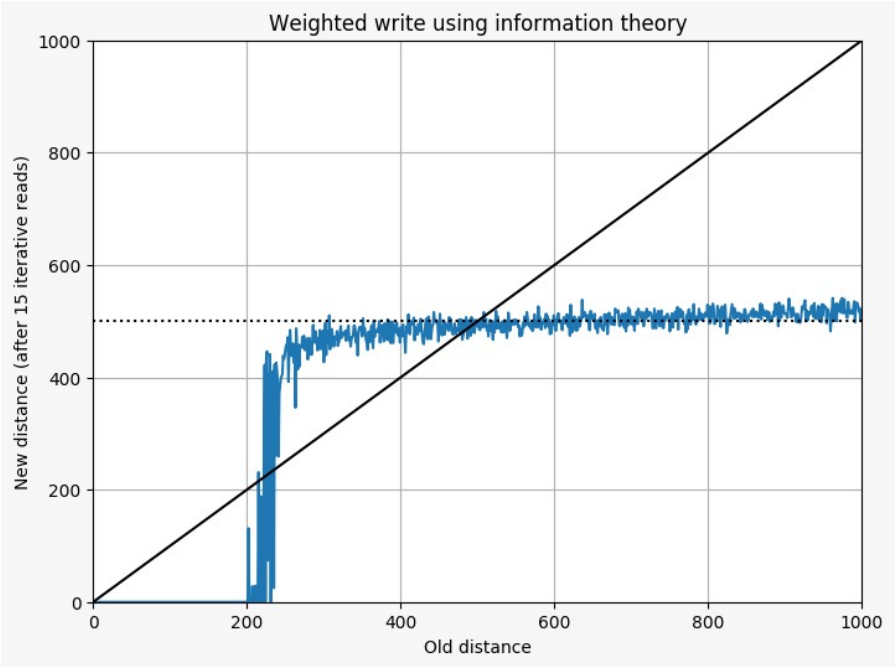


(c) stepwise $\lfloor w_2(d) \rfloor$ for fast integer computation.

Figure 64: SOON TO BE DEPRECATED. Shannon write operation: Computing the sum of low-likelihood signals. (a) entirety of the space; (b) region of interest; (c) Fast integer computation through a stepwise function.

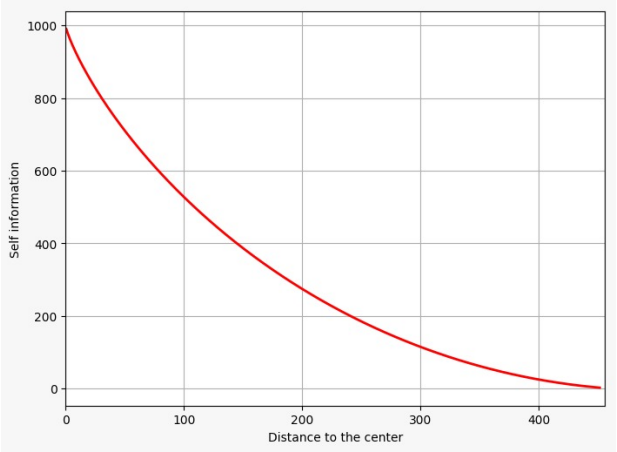


(a) Kanerva's model

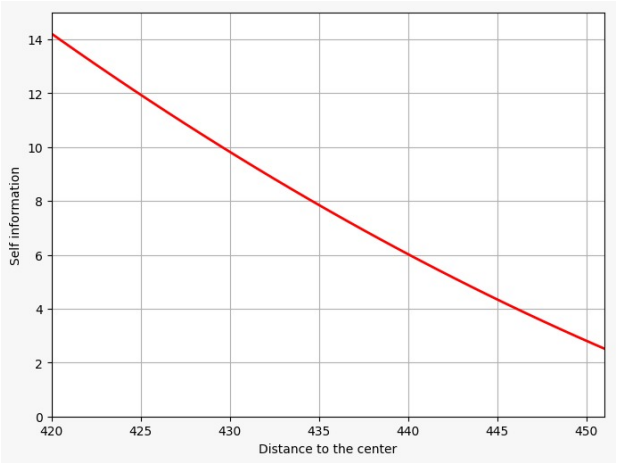


(b) Write process weighted by the amount of information contained in the distance between the written bitstring and each hard location

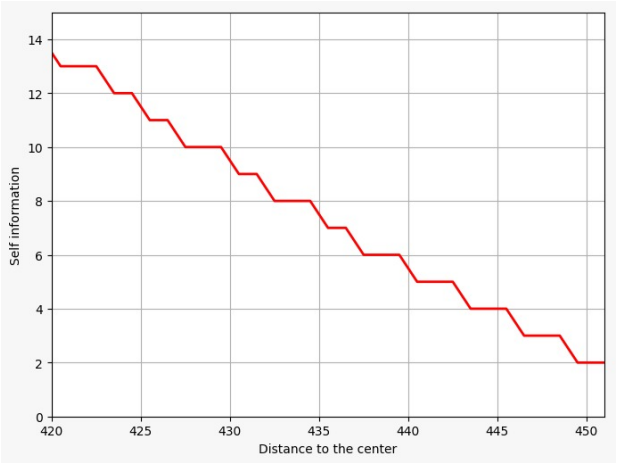
Figure 65: (a) and (b) show the behavior of the critical distance under Kanerva's model and the information-theoretic one, respectively.



(a) $w_2(d), d \in \{1, 2, ..., n\}$.



(b) $w_2(d)$ for the desired range.



(c) stepwise $\lfloor w_1(d) \rfloor$ for fast integer computation.

Figure 66: Shannon write operation: Computing the amount of information of a signal to each hard location in its access radius. (a) entirety of the space; (b) region of interest; (c) Fast integer computation is possible through a stepwise function.

14 CONCLUSION

Sparse Distributed Memory is a viable model of human memory, yet it does require researchers to (re-)implement a number of parallel algorithms in different architectures.

We provide a new, open-source, cross-platform, highly parallel framework in which researchers may be able to create hypotheses and test them computationally through minimal effort. The framework is well-documented for public release at this time (http://sdm-framework.readthedocs.io), it has already served as the backbone of Chada's Ph.D. thesis [5]. The single-line command "pip install sdm" will install the framework on posix-like systems, and single-line commands will let users test the framework, generate some of the figures from Kanerva's theoretical predictions in their own machines, and — if interested enough —, test their own theories and improve the framework, and the benchmarks used to evaluate the framework, in open-source fashion. It is our belief that such work is a necessary component towards accelerating research in this promising field.

Here are interesting questions that have been considered during this work, but have had to be left for future research.

14.1 MAGIC NUMBERS

Kanerva suggests, in his book, the use of 1,000 dimensions and 1,000,000 hard locations. More recently, he suggested the use of 10,000 dimensions, and on personal discussions suggested that this should be a minimum; as he has been concerned in latent semantic analysis and seems to be the proper scale in that application.

Each parameter set choice like this will lead to particular numbers — many of them emergent—, such as the access radius size, critical distance, and so forth.

One intriguing question here is: is there a 'better' number of dimensions and of hard locations? If so, can such numbers better studied analitically, or numerically?

How should these parameters be compared? What are the tradeoffs that should be considered? What are the 'best' benchmarks possible?

14.2 SYMMETRICAL, RAPIDLY ACCESSIBLE, HARD LOCATIONS

A hypercube with n dimensions can be divided by two hypercubes with $n-1$ dimensions. Is there an algorithm that separates the area

of each hard-location in such a form that there exists a function mapping each bitstring in $\{0,1\}^n$ to the set of hard locations it 'belongs to'? Though this would break Kanerva's assumption of a randomly yet uniformly distributed set of hard locations — for a perfectly symmetrical set of hard locations —, there could be large performance gains if such a mapping function from a bitstring to its corresponding set of nearest hard locations exists.

Consider the hypercube with n dimensions. We want to select a subset of its vertices with cardinality 2^{20} that is symmetrically distributed over the space. Afterwards, $\forall b \in \{0,1\}^n$, we want an algorithm A that yields the particular list of hard locations for b and all hard locations respect the desired properties of the memory.

A reduction from measuring the distance to 2^{20} hard locations to a computation of 2^{10} hard locations might yield astonishing performance gains, depending, of course, on our optimistic assumptions concerning existence and complexity of such algorithm. At large scales of computing, the very ability to perform some experiments is a function of sheer performance. The horizon of experiments — and possibly of knowledge — expands *as a function of computational demands*. A little more on this in my closing words.

14.3 PAVING AND ILLUMINATING THE PATHWAY

Let us revisit, in these concluding thoughts, the emphasis employed over speed of computation. At first sight, that might seem like a typical objective of efficiency in computer science. But we are not only interested in the computer science effects here — the ambition is different. More important than this 'computer-sciency' goal, i.e., a beautiful, clean, efficient algorithm with the primary effect of enhanced speed, however, is the secondary effect on the sociology of science: *We can see farther*.

Beyond speed, I have also strived for *ease of use*. All the simulations and graphics generated in this thesis are promptly available to be re-executed and explored by those interested. I have generated a Docker image, which makes it even easier to explore the framework. After running the container, a Jupyter Notebook is available with sdm-framework and other tools already installed. We invite the reader to take a look and explore a little bit.

The overarching intention here is to not only provide a starting point, but to provide a documented Framework in which SDM research can be conducted. Consider having the ability to compare the results of a new ('forked') model to the previous 'best' (under a particular benchmark set). For example, some of the benchmarks that we plan to develop in future research is: how fast is convergence through iterative reading? How large is the attractor of the critical distance? How well does the system filter noise? How well does the

system work under the supervised learning task? And other authors may be able to improve this benchmark set themselves, as is usual in open source development. It is perhaps this facility of ease to build on top of previous work that seems most exciting at this stage.

Consider the misunderstanding concerning the SDM read operation: Dr Stan Franklin describes Kanerva's read operation in a way that each hard location, at each dimension, provides only a single bit of information to the read operation (instead of Kanerva's full counter). We have referred to this modified read operation as Chada read[1]. Having an open, testable, codebase reduces the possibilities of such misunderstandings in the long run. Indeed, a high-quality codebase seems to have become a scientific community's form of unequivocally standing behind a consensus. For example, the journal Nature analysed the top-100 cited papers in history, to find:

> ... some surprises, not least that it takes a staggering 12,119 citations to rank in the top 100 — and that many of the world's most famous papers do not make the cut. A few that do, such as the first observation of carbon nanotubes (number 36) are indeed classic discoveries. But the vast majority describe experimental methods or *software that have become essential in their fields. [...] The list reveals just how powerfully research has been affected by computation and the analysis of large data sets.*
> — Van Noorden et al. [28], emphasis mine.

It is no coincidence that scientific journals such as BMC Neuroscience, or the Journal of Machine Learning Research have specific sections on open-source software. The journal Neurocomputing states, bluntly: "software is scientific method by machine".

Of course, for the skeptical reader who may consider software a less worthy pursuit, there is also new work here. The mathematics of the model has been shown to be correct numerically (with a single, small, anomaly); we have shown how to execute unsupervised learning with nothing besides operations original to the SDM; we have studied the generalized Murilo read; we have seen noise filtering; the death of neurons; how information-theory may be of use; and finally, we have reproduced numerous of the original propositions put forth by Kanerva. The emphasis might have been on *breadth of topics*, in detriment of depth here or there. But this is due to our research group's enthusiam for the topic; we do indeed believe that SDM is — if not correct — extremely close to a full

1 Legend has it that my friend & colleague, Dr. Daniel de Magalhães Chada, along with Linhares, did not consult and re-check with Kanerva's book and only discovered the discrepancy in code and ideas a couple of years afterwards.

scientific understanding of human long-term memory. If so, it is such a monumental achievement that we want readers to be able to see all of what we see and imagine the vastness of possibilities. The work on, say, reinforcement learning, is most definitely not the definitive work we will see on the subject, but a challenge left for readers to contemplate. Ralph Waldo Emerson once said *Do not go where the path may lead. Go, instead, where there is no path, and leave a trail.* Dr. Kanerva has left the trail. It is my job to pave it and illuminate it and to try to deliver an easier pathway for the next generation. Some essays completely shut the door close at the end; this one intends to leave it wide open. As the reader might have noticed, this final section does not read as an analysis of the work done; it reads, instead, as a *desiderata*, a prologue, a yearning for others to join me in imagining the shape of things to come.

15

APPENDIX

LIST OF JUPYTER NOTEBOOKS

BIBLIOGRAPHY

[1] Ashraf Anwar and Stan Franklin. Sparse distributed memory for 'conscious' software agents. *Cognitive Systems Research*, 4(4): 339–354, 2003.

[2] M. S. Brogliato. Understanding the critical distance in sparse distributed memory. Master's thesis, Escola Brasileira de Administração Pública e de Empresas - EBAPE, Fundação Getulio Vargas, 2011.

[3] Marcelo Brogliato. Sdm framework documentation. URL http://sdm-framework.readthedocs.io/.

[4] Marcelo S Brogliato, Daniel M Chada, and Alexandre Linhares. Sparse distributed memory: understanding the speed and robustness of expert memory. *Frontiers in Human Neuroscience*, 8: 222, 2014.

[5] Daniel de Magalhães Chada. *Are you experienced? Contributions towards experience recognition, cognition, and decision making*. PhD thesis, 2016.

[6] Timothy M Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the twenty-seventh annual symposium on Computational geometry*, pages 1–10. ACM, 2011.

[7] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[8] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[9] Antônio de Pádua Braga and Igor Aleksander. Geometrical treatment and statistical modelling of the distribution of patterns in the n-dimensional boolean space. *Pattern Recognition Letters*, 16 (5):507–515, 1995.

[10] Kuo-Chin Fan and Yuan-Kai Wang. A genetic sparse distributed memory approach to the application of handwritten character recognition. *Pattern Recognition*, 30(12):2015–2022, 1997.

[11] R. M. French. When coffee cups are like old elephants, or why representation modules dont make sense. In A. Riegler and M. Peschl, editors, *Proceedings of the 1997 International Conference*

on New Trends in Cognitive Science, pages 158–163. Austrian Society for Cognitive Science, 1997.

[12] Frank Harary, John P Hayes, and Horng-Jyh Wu. A survey of the theory of hypercube graphs. *Computers & Mathematics with Applications*, 15(4):277–289, 1988.

[13] Tim A Hely, David J Willshaw, and Gillian M Hayes. A new approach to kanerva's sparse distributed memory. *IEEE transactions on Neural Networks*, 8(3):791–794, 1997.

[14] Douglas R Hofstadter. On seeing a's and seeing as. *Stanford Humanities Review*, 4(2):109–121, 1995. URL https://web.stanford.edu/group/SHR/4-2/text/hofstadter.html.

[15] P. Kanerva. *Sparse Distributed Memory*. MIT Press, 1988.

[16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks-a publishing format for reproducible computational workflows. In *ELPUB*, pages 87–90, 2016.

[17] Alexandre Linhares, Daniel M. Chada, and Christian N. Aranha. The emergence of miller's magic number on a sparse distributed memory. *PLOS One*, 6(1):e15592, Jan 2011. doi: 10.1371/journal.pone.0015592.

[18] Mateus Mendes, Manuel Crisóstomo, and A Paulo Coimbra. Robot navigation using a sparse distributed memory. In *Robotics and automation, 2008. ICRA 2008. IEEE international conference on*, pages 53–58. IEEE, 2008.

[19] Meng et al. A modified sparse distributed memory model for extracting clean patterns from noisy inputs. *Proceedings of International Joint Conference on Neural Networks*, June 2009.

[20] Aaftab Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[21] Kenneth A Norman and Randall C O'reilly. Modeling hippocampal and neocortical contributions to recognition memory: a complementary-learning-systems approach. *Psychological review*, 110(4):611, 2003.

[22] Mohammad Norouzi, Ali Punjani, and David J Fleet. Fast exact search in hamming space with multi-index hashing. *IEEE transactions on pattern analysis and machine intelligence*, 36(6):1107–1119, 2014.

[23] Ram Pai, Badari Pulavarty, and Mingming Cao. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium*, volume 2, pages 105–116, 2004.

[24] Rajesh Rao and Olac Fuentes. Hierarchical learning of navigational behaviors in an autonomous robot using a predictive sparse distributed memory. *Machine Learning*, pages 87–113, 1998.

[25] Rajesh PN Rao and Dana H Ballard. Natural basis functions and topographic memory for face recognition. In *IJCAI*, pages 10–19, 1995.

[26] Helen Shen. Interactive notebooks: Sharing the code. *Nature News*, 515(7525):151, 2014.

[27] Javier Snaider and Stan Franklin. Extended sparse distributed memory. Paper presented at the Biological Inspired Cognitive Architectures 2011, Washington D.C. USA.

[28] Richard Van Noorden, Brendan Maher, and Regina Nuzzo. The top 100 papers. *Nature*, 514(7524):550, 2014.

[29] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.