TrackScorer: Skyrmion Logic-In-Memory Accelerator for Document Ranking

Martijn Noorlander
University of Twente
Email: m.s.noorlander@student.utwente.nl

***Abstract*—Skyrmion racetrack memories have shown to have lower leakage power and higher density compared to traditional memories like DRAM/SRAM. The cost of these memories comes with the penalty of time to shift data to an access port. Logic-In-Memory can be used to perform operations on data in memory without first transferring the data to a CPU. In this paper, the popular document ranking algorithm Quickscorer is mapped onto Skyrmion racetrack memory to use the advantages of this new memory technology. This mapping uses a Logic-In-Memory accelerator to improve performance on Skyrmion racetrack memories. The results show a decrease in the number of read and write operations that are performed, and in some cases show a decrease in the time spent shifting during the document ranking process.**

1. Introduction

Racetrack Memory (RM) is an upcoming non-volatile type of memory that packs bits of data in the domains of a ferromagnetic wire [1] or in Skyrmions [2]. This type of memory has several advantages, like a significantly lower leakage power compared to traditional DRAM or SRAM memories. Additionally, DWM and Skyrmion memories offer a higher density than DRAM/SRAM. These advantages do come at a cost, since the ferromagnetic wire, or Skyrmions, need to be shifted to an access port before data can be read or written. To minimize the number of shifts, data placement is integral to the performance of racetrack memories. In the background section several examples of data placement on RM will be discussed.

A limitation of all types of conventional memory is the time and bandwidth required to move large amounts of data between memory and logic [3]. To solve this problem, the idea of Logic-In-Memory (LIM) was introduced. By doing computations inside the memory, time and bandwidth can be saved. Skyrmion LIM has shown to be capable of binary NOT, AND, and OR through the physical properties of Skyrmions [4], allowing computations to happen inside of the memory without reading or writing data to the memory.

Given the nature of racetrack memory, applications require careful design in order to work efficiently. Various applications, such as sorting algorithms [5] and AES encryption [6], have been implemented in earlier research. Quickscorer [7] is a document ranking algorithm for tree ensembles that represents nodes as bitvectors, and is because of that able to use simple binary AND operations to find exit leaves. Quickscorer and its derivatives like V-Quickscorer [8] and Rapidscorer [9] have shown significant improvements compared to previous ranking algorithms [9]. The mapping of such a document ranking algorithm on Skyrmion racetrack memory (SK-RM) for the purpose of taking advantage of the reduced power consumption, non-volatility, and increased density has not yet been explored. Furthermore, given the nature of Quickscorer's binary operations which have been shown to be possible in Skyrmion, this research explores the concept of utilizing Skyrmion LIM to enhance the performance of Quickscorer on Skyrmion memory.

Our Contributions: In this work we propose methods for applying Quickscorer on SK-RM by combining the bitwise nature of Quickscorer with Skyrmion Logic-In-Memory. In short, the contributions are as follows:

- Several mappings for implementing Quickscorer on Skyrmion memories. These mappings include the usage of LIM and a genetic optimization algorithm for the ordering of tree result bitvectors in the memory.
- A proof of concept implementation of a Skyrmion Logic-In-Memory architecture that is capable of performing a binary AND operation between two tracks.
- Enhancements for RTSim to allow Skyrmion [10] and Skyrmion Logic-In-Memory [11] specific design exploration.

2. Background

This section provides an explanation about emerging racetrack memories with details on Skyrmion racetrack memories and Logic-In-Memory with Skyrmion. Additionally, the Quickscorer algorithm is explained.

2.1. Emerging memories

In the search for better types of memory, racetrack memories were developed. These offer both a higher density and a lower power consumption than the traditional DRAM/SRAM that they aim to replace. Two main types

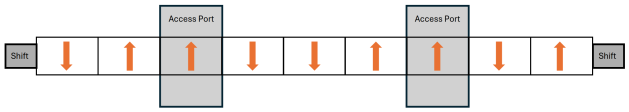


Figure 1. Domain Wall Memory with two access ports

of racetrack memory exist. The first type, Domain Wall Memory (DWM) consists of a ferromagnetic wire that is divided into domains. The direction in which these domains are magnetized represent '0' or '1'. To read or write a domain, a port uses a Magnetic Tunnel Junction (MTJ) to either detect the direction of the magnetization or change that direction. Domain Wall Memories typically contain multiple ports on each track. To access a bit, a shifting current is applied to the wire that moves the domains in the direction of that current [12]. To be able to read entire words at once, racetracks are often placed in parallel and shifted together. With these new techniques, a number of new problems arise, like under/over-shift errors and alignment issues between tracks.

Allwood et al. [13] proposed different domain-wall logic elements, including a binary NOT and a binary AND gate. The gates are achieved by creating Y shaped nanowires with specific widths and angles. Wang et al. [14] demonstrated the creation of domain-wall XOR logic by using two DW nanowires. This approach provided the same low leakage power as domain-wall memory. The XOR logic is then used to create DW based Adders, Multiplication, and Look-up table (LUT) logic. A block level architecture is proposed where data arrays, which can consist of multiple cells or racetracks, are mapped to a block of logic. For this work, the previously created components are mapped onto the logic blocks to create a Logic-In-Memory accelerator for image processing machine learning. The same authors then use this block-level architecture in a different work [6] to map the AES algorithm to LIM. This is done by changing the logic blocks that are mapped to the data array blocks in order to perform AES computations.

2.2. Skyrmion Memories

The second type of racetrack memory is Skyrmion racetrack memory. Skyrmions are nanoscale particle-like spin swirling configurations [5]. The existence of a Skyrmion in a racetrack encodes a '1', while absence encodes a '0'. Figure 2 shows a simple track. Skyrmions are moved by applying a small current (depinning current) to the material in which the Skyrmions reside [3]. Skyrmions can be destroyed by applying a current larger than the annihilation current. When applying the depinning current, the Skyrmion moves in multiple directions, caused by the Skyrmion Hall Effect. This effect can be likened to the behaviour of moving a rotating rigid ball [15]. Multiple solutions have been presented to counter this effect. Among these is the creation of a track that is curbed so the Skyrmions can only move in a single direction. In comparison to Domain Wall Memories, Skyrmions can thus move in different directions. Kang et al. [2] compared DW based memories with Skyrmion memories and concluded that Skyrmion has the potential for denser, robuster, and more efficient memories. Similar to DWM, access ports are required to create or read Skyrmions, Skyrmions are created with an injector, and the presence of a Skyrmion is detected using a detector [16].

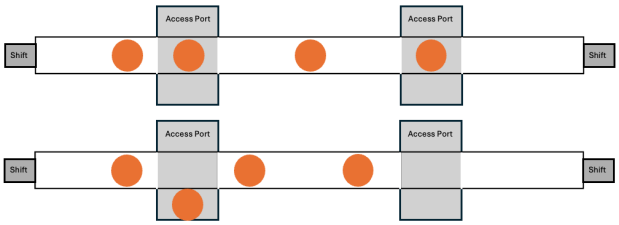


Figure 2. Skyrmion delete operation: a Skyrmion is shifted out of the track at the access port and the other Skyrmions are then shifted back.

One of the challenges of Skyrmion memories is the synchronization of Skyrmion positions in both memory and logic applications. For the logic gates to function correctly, Skyrmions or non-Skyrmions need to arrive within a certain timeframe from one another. For the data, when shifting multiple tracks at the same time, the bits within a word should stay aligned. A solution for this is the use of Voltage-Controlled Magnetic Anisotropy gates [17]. These VCMA gates are energy barriers capable of preventing Skyrmions of passing through when a certain voltage is present.

Another advantage of Skyrmion memories is the ability to insert or delete Skyrmions at certain locations without changing the rest of the track. As shown in Figure 2, a Skyrmion can be shifted out of the track, while this is reversed during Insert operations. Achieving the same data modification on DWM would require a large part of the nanowire to be updated. Hsieh et al. [5] proposed a shift-limited sorting algorithm for Skyrmion Racetrack memory. This algorithm minimizes the required number of shifts in order to sort a set of numbers. The ability to insert and delete Skyrmion is used throughout the shift-limited sorting algorithm to significantly reduce the number of shifts compared to running traditional sorting algorithms on racetrack memory. To minimize the number of Skyrmions that need to be created or destroyed, Yang et al. [16] used the insert/delete capabilities of Skyrmion racetrack memories to reuse Skyrmions during certain write operations. By temporarily storing Skyrmions in an assemble area, they can be shifted back into the track to represent different data values.

Hsieh et al. [18] described a method for detecting and correcting errors in Skyrmion Racetrack memories. The methods were tested on an adopted version of RTSim [19].

2.3. Skyrmion logic

In Chauwin [4] the Skyrmion Hall Effect is used to create reversible logic gates. By creating racetracks in a

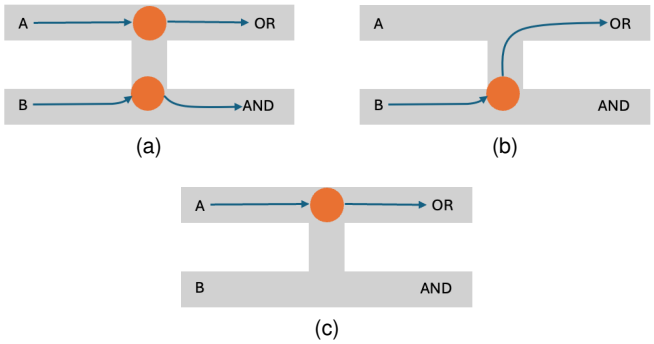


Figure 3. Skyrmion AND/OR gate. The scenario in 3a shows a Skyrmion from port A that moves directly to OR. The Skyrmion from B starts moving upwards because of the Skyrmion Hall Effect, but is then pushed into the AND output because of repulsion from the first Skyrmion. In 3b, a Skyrmion moves from B into the OR output because of the Skyrmion Hall Effect. Finally, in 3c a Skyrmion from A directly moves to OR.

specific shape with junctions, while using the repulsion effect between Skyrmions, several logic functions can be created within Skyrmion racetrack memories. This provided significant advantages in minimizing the required number of shifts and energy consumption. Two important types of gates are discussed: an AND/OR gate and NOT/COPY gate. The first gate takes two Skyrmion inputs and provides an OR and AND Skyrmion output. A schematic of an AND/OR gate is shown in Figure 3. The NOT/COPY gate takes two inputs: an input for the data and a control input. Three outputs are then provided: two ports that should be identical (copy 1 and copy 2) to the data input, and a NOT that should be the inverse of the data. The control input is used to provide a Skyrmion that can either be used as copy or as inverse of the input, since the number of Skyrmions going in and out of the gate should stay the same.

Gnoli et al. [3] proposed a Logic-In-Memory architecture for performing maximum/minimum search within Skyrmion racetrack memory. An architecture was designed in which the racetracks can both act as memory but also contain logic AND and OR gates. A control unit is responsible for executing the logic for the algorithm.

It is possible to convert between Skyrmions and Domain Wall as shown in Kan et al. [20]. This method is used in Liu et al. [17] to create a Processing-In-Memory (PIM) architecture for CNN's in a combination of both Skyrmion and DWM. The blocks in the architecture consist of a Domain-Wall storage unit and a Skyrmion computing unit. These two are then connected with two converters to convert between Skyrmions and Domain-Wall. The Skyrmion blocks contain a set of Skyrmion logic gates from previous work that are combined into an XOR gate that is used for adder logic. Furthermore, a Skyrmion nanowire based multiplier is designed.

In Pan et al. [21], a Logic-In-Memory architecture for binary neural convolution networks is shown within Skyrmion memory. The architecture consists of horizontal and vertical racetrack groups that have connected AND and XOR operators. Inputs are written to the vertical groups, whilst weights are written to the horizontal groups.

2.4. Quickscorer

Tree ensembles have shown great performance in ranking search engine query results [7]. When performing a query, large numbers of trees have to be traversed simultaneously. Luchesse [7] proposed Quickscorer, a document ranking algorithm for tree ensembles. Quickscorer relies on bitvectors to simplify computations by allowing the use of bitwise operations. The algorithm for Quickscorer is shown in Algorithm 1. The algorithm takes a tree ensemble T, and a feature vector x that will be scored. The tree ensemble is mapped to memory in a set of arrays. Most importantly, the nodes of all trees are grouped by matching features and then sorted by threshold value. For each of the nodes in the tree, a bitvector is created. This bitvector encodes the leaves that can be reached if a node is a true node.

The algorithm starts by setting the result bitvectors for each of the trees in the ensemble to 11...11. Next, the algorithm iterates through each of the features. For any node where the feature value is above that node's threshold, a binary AND operation is performed between the bitvector and the tree's bitvector to remove unreachable leaves. Once a node is found where the feature value is below the threshold, the algorithm can skip the remaining nodes and continue to the next feature. Finally, the algorithm takes the value of the exit leaf (the first leaf in the tree bitvector that is set to 1) together with the exit leaf values of all other trees to compute a score for the feature vector x.

Algorithm 1 Quickscorer(features x, ensemble T)

```
// Set all result bitvectors to 1
for h = 0 to h = |T| do
    result_bitvectors[h] ← 11...11
end for

// Iterate through features
for k ← 0 to k ← |F| do
    for i ← offsets[k] to i ← offsets[k + 1] do
        if x[k] > thresholds[i] then
            h ← tree_ids[i]
            result_bitvectors[h] ← result_bitvectors[h] ∧
            bitvectors[i]
        else
            break
        end if
    end for
end for

score = 0
for h = 0 to h = |T| do
    i ← first bit index set to 1 in bitvectors[h]
    score ← score + leaf_values[h ∗ |L| + i]
end for
```

In V-Quickscorer [8], SIMD instructions are used to parallelize the scoring of documents. Depending on the

hardware platform, up to 8 documents (assuming 32bit bitvectors) can be ranked at once. The changes compared to Quickscorer are shown in Algorithm 2. To further improve performance, Rapidscorer is presented in Ye et al. [9]. In this algorithm, a combination of SIMD instructions and a different memory layout are used. One of the advantages of this memory layout is a more compact representation of the bitvector which can represent a larger number of exit leaves.

Algorithm 2 V-Quickscorer(features $\{x\}$, ensemble T)

// Set all result bitvectors for all documents to 1
for $i = 0$ to $i = |T| * NDocuments$ **do**
  $result\_bitvectors[i] \leftarrow 11...11$
end for

for $k \leftarrow 0$ to $k \leftarrow |F|$ **do**
  $vectorized\_x \leftarrow$ vectorize$(x[k+NFeatures], x[k+NFeatures*4], x[k+NFeatures*8], ...)$
  for $i \leftarrow offsets[k]$ to $i \leftarrow offsets[k+1]$ **do**
    $mask \leftarrow$ compare($vectorized\_x$,thresholds$[i]$)
    if $mask == 00...00$ **then**
      break;
    end if
    $vectorized\_result\_bitvector \leftarrow$ vectorize$(\&result_bitvectors[h*NDocuments])$
    $m \leftarrow$ vectorized_andnot$(bitvector, mask)$
    $vectorized\_result\_bitvector \leftarrow$ vectorized_andnot$(vectorized_result_bitvector, m)$
    store_to_memory$(\&result_bitvectors[h*NDocuments], resultvectorized_result_bitvector)$
  end for
end for

3. Skyrmion Logic-In-Memory Simulation

A simulator for SK-RM is required in order to test different mappings of the algorithm. This section details the LIM element that the Quickscorer mappings use and subsequently presents the enhancements that have been made to RTSim in order to support Skyrmion and Logic-In-Memory.

3.1. Hardware architecture

For Gnoli [3], an HDL framework was defined to simulate Skyrmions. This framework tracks the positions of each Skyrmion in the defined layout and supports the aforementioned Skyrmion logic elements AND/OR and COPY/NOT. Using this library, a proof of concept of a LIM architecture that can be used for mapping Quickscorer to Skyrmion racetrack memory was created. The requirements for this architecture were as follows:

- There need to be two racetracks, one for the bitvector input and one for the result of the AND operation.
- The original bitvectors should be immutable, since the data is reused between the scoring of different documents.

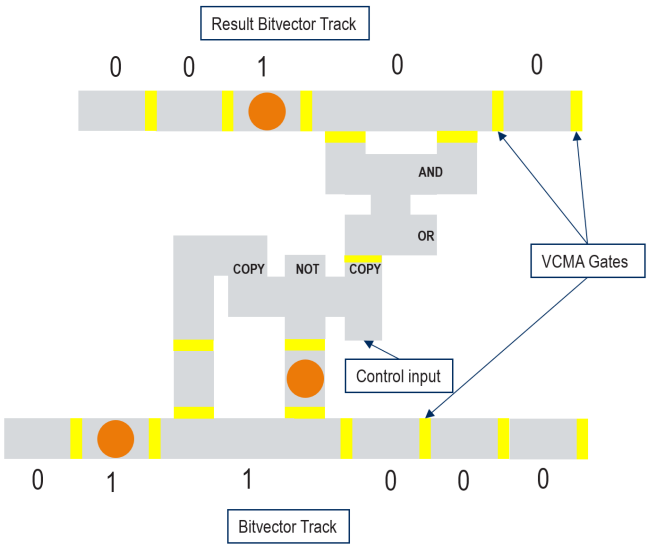


Figure 4. LIM Architecture for Quickscorer. Yellow lines are VCMA gates. An AND/OR and a COPY/NOT element are present. A Skyrmion of the bitvector track is present at the input for the COPY/NOT so that it is copied back into the track.

A Skyrmion layout was defined that could perform the AND operation between two memory tracks and subsequently store the result in one of the original locations. This operation represents the binary AND between a bitvector and result bitvector in the Quickscorer algorithms. The layout consists of three major components. First, there are the two racetracks for storage, built from blocks of track and VCMA controls for synchronizing Skyrmion locations. One of the racetracks stores one of the bits for all the bitvectors, the other track stores said bit for the result bitvectors of each tree. The third component is the logic. It is important that the contents of the original bitvector track do not change, since this bitvector is reused between different runs of the algorithm. A COPY/NOT gate is therefore connected to the two T-shaped junctions in the bitvector track using VCMA gates. When shifting within the bitvector track, control logic for the different VCMA gates and different current directions are used to copy a Skyrmion or the absence of a Skyrmion. The result is then fed to the AND/OR gate, whereas the other Skyrmion/non-Skyrmion is fed back to the original racetrack. This guarantees that the result of the binary AND operation cannot influence the original value of the bitvector track that is used in future executions of the algorithm. For the COPY/NOT gate to work, a Skyrmion has to be generated/inserted into one of the input ports of the COPY/NOT gate. An overview of the Skyrmions' locations during operation is given in Table 1. Given that a Skyrmion will always be present at either the NOT and/or the OR output, this Skyrmion can potentially be used as input for the COPY. However, to simplify the implementation of the Skyrmion logic, this is currently not implemented. A testbench was created to verify that the separate tracks could be shifted in two directions and that the logic implementation functioned as expected.

Bit Vec	Res Vec	Gen	NOT	OR	AND
0	0	1	1	0	0
0	1	1	1	1	0
1	0	1	0	1	0
1	1	1	0	1	1

TABLE 1. TABLE REPRESENTING SKYRMION LOGIC THROUGHOUT THE OPERATION.

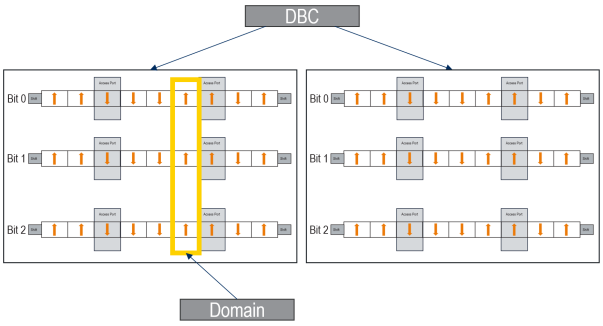


Figure 5. Demonstration of the difference between DBC's, Domains, and bits in RTSim. Two DBC's are shown with a wordsize of 3, so each DBC therefore consists of 3 tracks. The highlighted domains have a value of '111'.

In the application, this architecture is multiplied depending on the required configuration. In this research, a wordsize of 32-bit is used, hence 32 of these tracks are created in parallel.

3.2. Memory simulation

Khan [19] proposed a simulator for the simulation of racetrack memories. Both DWM and Skyrmion based memories can be simulated. This simulator, RTSim, is an extension to NVMain, a simulator for memory on an architectural level. The simulator has the ability to simulate memory traces containing read and write actions. It is possible to use RTSim/NVMain as memory backend for Gem5 [22]. This way, complete progress can be simulated with a Racetrack memory backend. To support racetrack memories, several changes were made in RTSim compared to NVMain.

First, RTSim introduces the concept of Domain Block Cluster's (DBC's) and Domains. A DBC represents a set of tracks that is equal to the configured word size. For a wordsize of 32-bit, a DBC with 32 subtracks is defined. A domain references a specific bit within said DBC. Figure 5 demonstrate this difference. In this figure, two DBC's are shown with 9 domains each. The width of each DBC is 3 bits. The number of access ports in RTSim can be configured. For each of the configured number of DBC's a separate set of access ports is maintained. All tracks within a DBC shift together in order for words to stay aligned.

In the trace files, two different types of memory requests are supported: Read and Write. When either of those is dispatched, RTSim automatically dispatches a Shift request that will processed beforehand. This Shift request finds the closest access port to the requested memory location and then updates all port locations within said DBC to reflect that number of shifts. Two different shift behaviours can be configured: lazy and eager. When lazy shifting, a DBC is shifted to a location and kept there. When eager shifting, the access ports are returned to their initial position when the request is complete. Trace files have a number of fields, some of these are the cycle, instruction type, address, and data fields. Since RTSim/NVMain is an architectural level simulator, actual data is not stored in the sim. Therefore, the data fields represent the data before and after the instruction, so that metrics that require knowledge of bits can still be created.

Cycle	Instr	Addr	New Data	Old Data	Thread ID
10	W	0x781	173513500	223313500	0
20	R	0x781	000000000	173513500	0

TABLE 2. TRACE FILE FORMAT

As the naming in RTSim suggests, these changes were created to support Domain Wall Memory, not necessarily Skyrmion racetrack memories. In order to use RTSim for this research, several enhancements were made to RTSim. First, the ability to perform Insert and Delete requests was added in order to be able to use these abilities of Skyrmion memories later on. As with Write and Read, automatic Shift requests are dispatched together with these operations. The second enhancement added a metric to keep track of the number of Skyrmions that are created or destroyed during certain operations. For example, when performing a Write request, the number of Skyrmions that are created or destroyed can be determined by comparing each bit in both the new and old data field. For the sake of simplicity, a group of tracks is still referred to as a DBC, while a bit in a track is still referred to as a Domain even though this is technically not the correct name. These two modifications have been accepted and merged back into mainstream.

The third modification was the addition of the Logic-In-Memory request to simulate LIM operations. For this operation, there are no automatic shift requests, since this operation shifts two DBC's at once. The address in the request refers to the address of the bitvector that should be used as input for the LIM operation. Because a trace line only has a single address column, the new data field was used to encode the address of the result bitvector where the result of the LIM operation would end up. This prevented large modifications to RTSim to change the structure of requests. In this instruction, the assumption is made that LIM blocks, like the previously presented architecture, are present at each access port. Execution for the LIM requests starts by finding the closest port for the request address, then both the DBC of the request address (which is the DBC with bitvectors) and the address in the new data field (which is the DBC with result bitvectors) are shifted to that access port to represent a Logic-In-Memory simulation. The new and old data fields are then used to compute the number of Skyrmions created and destroyed during this instruction. These numbers are based on the results from Table 1. For

the parallel LIM mapping that is discussed later, a mask was added to only apply the shifting and Skyrmion counts to DBC's that have to be activated.

The final modification was the addition of another metric: total shift duration. For the original metric that keeps track of the total number of shifts, each Skyrmion/non-Skyrmion that is shifted a position is counted once. For example, in a DBC with a wordsize of 32-bit that is shifted 4 positions, the total number of shifts will increase by 32*4. This does not, however, represent the total time spent waiting for shifts to complete, since the 32 tracks can be shifted in parallel. Therefore, the total time spent waiting will only be that of 4 shifts. Ignoring parallel shifts, the final added metric is the total shift duration. For the LIM instruction, this number is increased by the longest shift of the two tracks, since that shift will bottleneck the operation.

4. Quickscorer on Skyrmion

This section discusses the method for generating RTSim trace files for Quickscorer and the four mappings that were used to map Quickscorer to SK-RM. The first method is a basic mapping without Logic-In-Memory, whereas the last three mappings use the LIM operation that was added to RTSim.

4.1. Quickscorer Trace Generation

The code used to run Quickscorer in Lucchese et al. [23] was used to run the different Quickscorer experiments. The Microsoft Learning to Rank dataset [24] was used as dataset to create and train tree ensembles to run the code on. The first fold of the dataset was used, which consists of five parts: three for training, one for validation, and one for tests. This fold was then trained using XGBoost [25].

In order to create memory traces from the Quickscorer runs, a C++ class was written that could create and fill an RTSim compatible trace file. This class contains methods for all the different Skyrmion memory instruction (Read, Write, Insert, Delete, LIM) and calculates the correct memory address based on the requested DBC/Domain. The Quickscorer code was then modified to include calls to this class. This results in the code still running on a normal architecture to validate the correctness of the changes to the algorithm, but provides the ability to measure the effect of all memory instructions on Skyrmion memory. Algorithm 4 shows an example of this. This case assumes that the $result\_bitvectors$ array is mapped to an entire (currently empty) DBC in racetrack memory called res_dbc. The $skyrmion\_trace\_write$ function will then create a line in the trace file with a cycle count, "W" instruction, calculated memory address from res_dbc, and h, the hexadecimal representation of 11...11 as new data and 0...0 as old data.

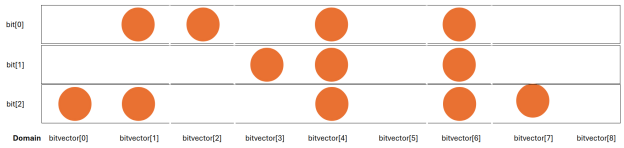


Figure 6. Simple mapping of Quickscorer

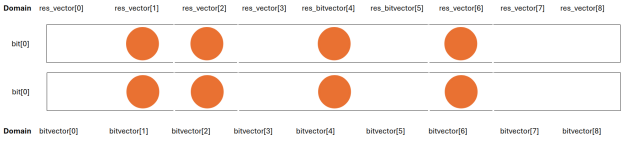


Figure 7. LIM mapping of Quickscorer

Algorithm 3 TraceDemonstrator(ensemble T)

1: // Set all result bitvectors to 1
2: **for** $h = 0$ to $h = |T|$ **do**
3: $\quad result\_bitvectors[h] \leftarrow 11...11$
4: $\quad skyrmion\_trace\_write(res\_dbc, h, 00...00, 11...11)$
5: **end for**

4.2. Algorithm Mapings

4.2.1. Base mapping. Quickscorer is mapped to Skyrmion memories in several different ways. The first mapping is rather simple, as it maps each of the input arrays to a DBC. This produced a memory with 8 DBC's, each consisting of 32 subtracks as this was the chosen word size. Quickscorer code was modified to produce memory trace information during runtime. Each time one of the data arrays was accessed/modified, this action was appended to the trace file with a correct instruction. The trace also contained the data before and after completion of the instruction, in order for RTSim to be able to calculate the number of Skyrmions that had to be created or destroyed. Figure 6 shows an overview of a basic mapping of the bitvector array to the memory. In all cases, traces were only generated for the ranking algorithms, not the data preparation phase, since this is only done once, independent from the number of documents that are ranked.

4.2.2. LIM Mapping. The second mapping is similar to the first, except for the utilization of Logic-In-Memory for the AND operation to calculate the result bitvector. Instead of separate instructions that first read the result_bitvectors, then the bitvector for that leaf, and then write that back to the correct result_bitvector, a single instruction is used. This LIM instruction has the bitvector DBC and domain as address while the result_bitvector DBC and domain are encoded in one of the data fields. For both tracks the original and new data are also present in the trace. The implementation of the LIM instruction in RTSim will shift both the tracks for bitvectors and result_bitvectors to the same access port in order for the correct domains to be used. A single bit mapping is shown in Figure 7. Instead of

all tracks of a data type being mapped together, the bits of bitvectors and result_bitvectors are interleaved to allow for logic elements to be placed physically between tracks.

With this mapping, the requirement for bits in the bitvector and result bitvector to be in physical proximity of one another, in order for the Skyrmions to be fed into the logic, causes an overhead in shifting in the result bitvector. Depending on the number of access ports in the racetrack, this overhead can be significant. For example, with a track length of 32768 and inter-port distance of 32, a Skyrmion would normally take at most 32 shifts in order to be at an access port. With the introduction of LIM, this can be increased up to the number of trees in the ensemble, since a bitvector n might belong to the first tree in the ensemble and bitvector $n+1$ might belong to the last tree in the ensemble.

To decrease the overhead in shifts caused by the LIM implementation, a more efficient mapping of the trees in the result bitvectors is needed. Quickscorer uses a look-up-table (LUT) called $tree\_ids$ to determine which result bitvector belongs to the node. By default, tree 0 is mapped to result bitvector 0, tree 1 to result bitvector 1, and so on. In a worst case scenario where each node in the ensemble has to be visited, this problem can be defined as

$$shifts = \sum_{i=o}^{i=|bitvectors|-2} |tree\_ids[i] - tree\_ids[i+1]|$$

Since every tree can be accessed multiple times, this problem cannot be efficiently solved. Iterating over all permutations would result in a complexity of $|T|!$ and is thus not feasible for larger ensembles. In order to compute a better ordering than the default, a genetic algorithm was used.

Algorithm 4 GeneticOptimization($tree\_ids[]$)

for $h = 0$ to $h = population\_size$ **do**
  $population[] \leftarrow [random()]$
end for
for $i = 0$ to $i = iterations$ **do**
  $evaluate(population)$
  $sort\_by\_score(population)$
  $new\_pop \leftarrow population[: 10]$
  $new\_pop \leftarrow gen\_offspring(population[: 50])$
  $population \leftarrow new\_population$
end for

$evaluate(population)$
$sort\_by\_score(population)$
return $population[0]$

In this genetic algorithm, a population of randomly sorted mappings is created. For each iteration of the algorithm, these mappings are evaluated by calculating the required number of shifts for the worst-case ranking in which all the nodes in an ensemble need to be visited. The top 10% are then directly added to the new population. The top 50% are used to create new offspring. This consists of taking two random members of the top 50% and defining a random crossover point. The first member is copied up to the crossover point, and then the missing tree ids are added in order of the second member. Finally, to reduce the chance of a local minimum [26], a random mutation can occur in 10% of the cases. In this situation, two random ids in the mapping of the new member are swapped. After a configured number of iterations have completed, the final population is ranked by score, and the population member with the lowest score is selected.

4.2.3. Sequential LIM Mapping. For the third option, a mapping similar to that of V-Quickscorer was done. Instead of processing a single document for each iteration, multiple documents were processed at once. This was done by placing them sequentially in the racetrack memory. The size of the result_bitvectors was thus increased by a factor of $NDocuments$. Instead of vectorized operations, up to $NDocuments$ LIM instructions were dispatched for each bitvector comparison.

4.2.4. Parallel LIM Mapping. The fourth option builds on the third mapping. Instead of sequentially processing multiple documents in one iteration of the algorithm, the documents are processed in parallel. The number of DBC's is increased in order for there to be $NDocuments$ DBC's for both the bitvectors and result bitvectors. As with the previous option, it is assumed that only the normal wordsize (32) can be read and written in an instruction. The LIM instruction then processes multiple bitvectors at once, based on a mask that is added to the LIM instruction.

For all options, the mapping only takes into account the actual execution of the algorithm. Data preparation is not taken into account, since this research focuses on the actual ranking performance.

5. Evaluation

This section presents the results of the trace files that were generated with Quickscorer and then ran under RTSim with several different configurations. The setup and configured options are discussed together with the metrics that were chosen to evaluate the performance of the implementations.

5.1. Evaluation Setup

The four different mappings were tested in RTSim with a variety of different parameters. For all mappings, three different port counts were tested: 128, 512, and 1024. In RTSim, the shifting policy was set to "Lazy", so after shifting a domain to an access port, the track will remain in that position until the next instruction. All mappings used a size of 32768 domains per DBC and the number of DBC's was set to 8 for all experiments except for the parallel LIM mapping. For this mapping, the number of DBC's was increased to 24 to account for the increased number of parallel operations (both the bitvector and result bitvector will now require 8 DBC's each). The word size was kept the same at 32.

All experiments were performed on a tree ensemble of 1000 trees with each experiment ranking 512 documents. For the genetic algorithm, 3600 iterations were executed on the ensemble of 1000 trees. Both the default and genetic mapping were tested. The genetic mapping was stored after running, to allow subsequent experiments to use the same genetic mapping. Finally, a parameter was added into RTSim to test the effects of reusing Skyrmions that are otherwise destroyed in the LIM operation. As this is currently not modelled on the HDL part of the simulations, this indicates a possible future improvement.

To evaluate this research, several parameters from RTSim were selected. These were chosen as parameters related to Skyrmion racetrack memories, in order for the influence of different mappings to be evaluated.

Number of Shifts This value represents the total number of shifts that had to be completed during the runtime of the algorithm. Shifts in tracks are multiplied by the word size of the track. Shifting 10 positions in a track with word size 32 will thus result in 320 shifts. For energy consumption, this number is significant, since each shift consumes energy. The aforementioned situation will result in 320 times the shifting energy.

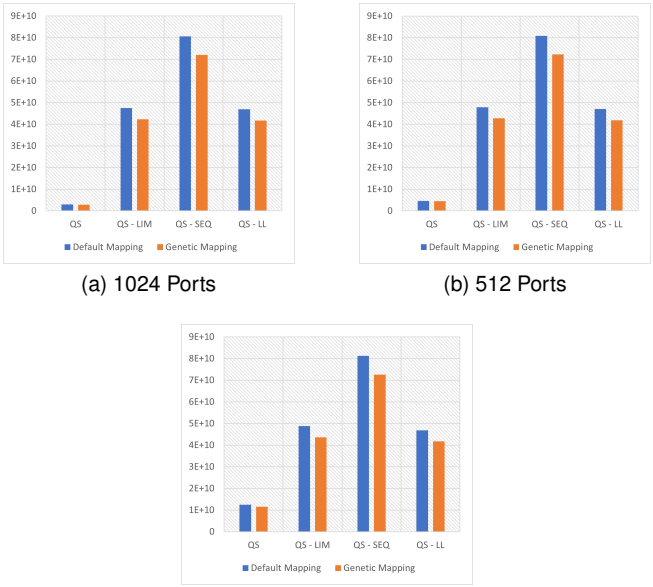
Total Shift duration This represents the number of shifts that caused delay. Since shifting can happen in parallel, the number of shifts parameter does not necessarily present a fair evaluation of the time wasted on shifting. Using the previous example, shifting 10 positions on a track with wordsize 32 only causes a delay of 10 shifts, since 32 of these happen in parallel. More importantly, for the LIM operation, when both tracks are shifted at the same time, only the track with the longest distance to shift has to be taken into account. When shifting two tracks, one with a distance of 10 and the other with a distance of 20, the time limiting factor is 20 shifts, not the combination of 30.

Reads and Writes These represent the total number of read and write operations that are performed during the ranking of the documents.

Detect This parameters represents the total number of detect operations that are performed. When reading, a detect has to be performed for every bit in order to determine whether a Skyrmion is present. When writing, a detect has to be performed for every bit, in order to determine if a Skyrmion has to be created or destroyed.

Skyrmions Created This is the total number of Skyrmions that have been created. A higher value indicates more Skyrmions that are created and thus a higher energy consumption. Since the creation of Skyrmions is energy expensive, this is an important metric for energy consumption.

Skyrmions Destroyed This value represents the total number of Skyrmions that have been destroyed. Since destroying a Skyrmion requires a higher current than the shifting current, this parameter is important for energy consumption.



(a) 1024 Ports (b) 512 Ports

(c) 128 Ports

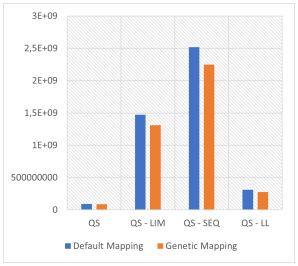
Figure 8. Total shifts during the ranking of a set of documents. For each track per DBC a shift is counted. For example, shifting a DBC with wordsize of 32-bits by 3 positions, counts as 3*32=96 shifts.

5.2. Shifts

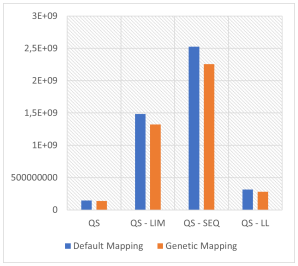
The total number of shifts in Figure 8 shows that the Logic-In-Memory implementations all significantly increase the total number of shifts that occur during the experiments. This can be attributed to the requirement of physical proximity of the data in order to be able to use the logic as explained earlier. Even for the lower port counts, this shows a significant increase in shifting energy for the LIM implementation. Parallel shifts also show an increase for some of the experiments. The parallel logic implementation is the most promising and shows that for a lower number of access ports, the parallel LIM implementation spends less time shifting to rank the same number of documents on the base Quickscorer algorithm. For the LIM implementations, the number of access ports does not significantly change the amount of shifting operations. The genetic mapping shows the most improvement for the Logic-In-Memory implementations, since the alignment of bitvector and result bitvector tracks causes larger shift distances. For the base Quickscorer mapping, an improvement of 3% is found for both the total number of shifts and the number of parallel shifts. The LIM mappings all show around a 10% improvement in the total number of shifts and the number of parallel shifts when the genetic mapping is used.

5.3. Read and write operations

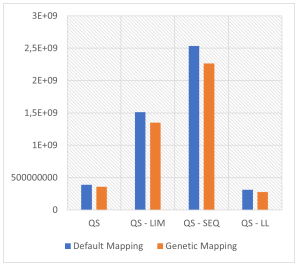
The results in Figure 10 show a significant decrease in read and write operations in all of the LIM implementations. The sequential and parallel LIM implementations show an improvement over the base Logic-In-Memory mapping,



(a) 1024 Ports

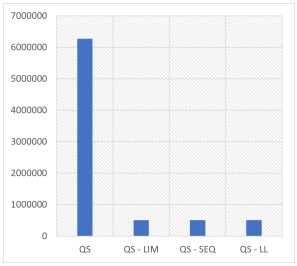


(b) 512 Ports

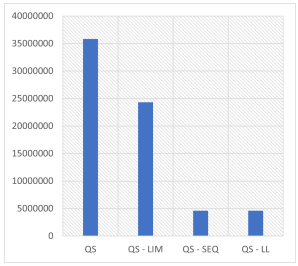


(c) 128 Ports

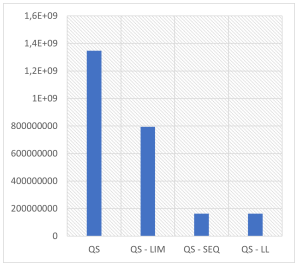
Figure 9. Parallel shifts. This metric only counts the number of sequential shifts. Multiple shifts in parallel only count for a single shift, this represents the total time spent shifting during the ranking process.



(a) Write operations



(b) Read operations



(c) Skyrmion detect operations

Figure 10. IO Operations with regards to reading and writing to the Skyrmion memory. Each read or write represents reading or writing a 32-bit word. The detect operations are the number of single Skyrmion detect operations that were executed.

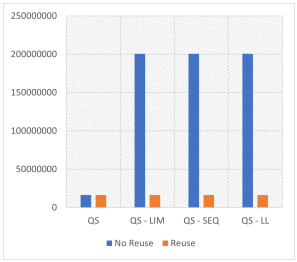
since data elements like node thresholds and tree_ids are only read once for every 8 documents. Since the LIM blocks purely rely on the physical properties of Skyrmion to perform the logical AND operation, the number of detect operations is also significantly reduced.

5.4. Skyrmion creation and destruction

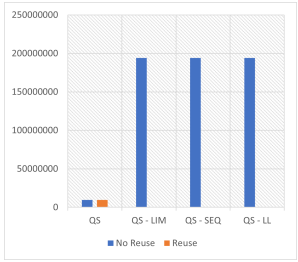
As shown in Figure 11, all LIM implementations show a very significant increase in Skyrmions that are created during the ranking of documents when the reuse option is not used. This can be attributed to the required input Skyrmion during the copy phase of the LIM block. Assuming the ability to reuse Skyrmions is present, the number of Skyrmions created is the same as that of the standard Quickscorer implementation. In these cases, Skyrmions are only created once during the initialization phase of the algorithm, when all result bitvectors are set to 1. Similar to the creation of Skyrmions, the absence of reuse shows in the number of Skyrmions that are destroyed. For every LIM operation, at least one Skyrmion is destroyed, in some cases two. When the reuse option is used here, the results show that no Skyrmions are destroyed.

5.5. Discussion

The experiments show mixed results for some areas depending on the parameters. The total number of shift operations that occur are increased compared to that of the base Quickscorer mapping, indicating an increased energy consumption in this area. For the higher port counts, the



(a) Skyrmions created



(b) Skyrmions destroyed

Figure 11. Results of the different implementations with regards to the number of Skyrmions that were created or destroyed. The LIM implementations show two different sets of results, one where the input Skyrmion can be reused and one where this is not the case.

difference between the base mapping and best LIM mapping (parallel LIM) is 14.2x. For the lower port count this difference lowers to 3.6x. For the LIM mapping itself, the shifting difference between different port counts does not influence the number of shifts, showing that a less complex architecture can achieve similar results. The total time spent shifting shows the best LIM mapping (parallel LIM) performing 3.1x worse than the base Quickscorer mapping for the highest port count, while performing 22% better for the lowest port count.

The number of IO operations show a clear decrease across all experiments, with the sequential and parallel LIM mappings performing the best. These two implementations only read data like thresholds, offsets, and tree_ids once for each set of 8 documents. The base LIM mapping does these

for each document separately. The base Quickscorer implementation has to read both the node and result bitvector and write that back, resulting in a significantly higher operation count.

For the Skyrmion creation and destruction results, the experiments show that the reusability of Skyrmions needs to be implemented for the LIM mappings to be viable. Without reuse, 12.3x as many Skyrmions need to be created and 20x as many Skyrmions needs to be destroyed to rank the same set of documents. Assuming the ability to reuse Skyrmions, the LIM mappings only require the creation of Skyrmions when filling the result bitvector array and do not destroy any Skyrmions during the ranking process (since no write requests are issued).

6. Conclusion

In this work, a method for mapping Quickscorer to Skyrmion racetrack memories is proposed that uses Logic-In-Memory to take advantage of the algorithms' characteristics. The experiments show a reduction in the number of IO instructions and show that when using SK-RM with a lower number of access ports, the time spent on shifting decreases. Enhancements to RTSim are introduced that provide methods to perform design exploration with Skyrmion racetrack memories and Skyrmion Logic-In-Memory.

Currently, two levels of simulation are used: one architectural on a memory level in RTSim, and one architectural on the Skyrmion level using the provided HDL library. The Skyrmion level simulation currently only serves as a proof of concept for a single LIM element. To further determine the feasibility of this LIM implementation, simulations of larger LIM blocks should be performed. Another possible improvement of the Skyrmion level simulation would be to simulate the reusability of Skyrmions, which is tested in some of the RTSim experiments. In order for this to be possible, a path and logic have to be created that allow Skyrmions from the NOT and OR gate to be transported back to the second input of the COPY/NOT element.

On the other hand, a genetic algorithm is used to improve the order of the result bitvectors. This genetic algorithm currently optimizes on the worst case tree traversals. Therefore, an improvement to this mapping is to use the probabilities for certain nodes in the ensemble to be visited, as the evaluation metric. A more optimum solution for the average traversal can then be found.

In this research, only a single level of parallelism was explored, ranking 8 documents instead of just one in an iteration of the algorithm. This number was taken from the code that was used to create the Quickscorer experiments. No other numbers were tested to keep the number of experiments manageable. Since the parallelism mapping shows the most promise, it could be beneficial to further increase the number of parallel ranked documents.

References

[1] R. Venkatesan, V. J. Kozhikkottu, M. Sharad, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, "Cache Design with Domain Wall Memory," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1010–1024, Apr. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7349153/

[2] W. Kang, X. Chen, D. Zhu, X. Zhang, Y. Zhou, K. Qiu, Y. Zhang, and W. Zhao, "A Comparative Study on Racetrack Memories: Domain Wall vs. Skyrmion," in *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. Hakodate: IEEE, Aug. 2018, pp. 7–12. [Online]. Available: https://ieeexplore.ieee.org/document/8537687/

[3] L. Gnoli, F. Riente, M. Vacca, M. Ruo Roch, and M. Graziano, "Skyrmion Logic-In-Memory Architecture for Maximum/Minimum Search," *Electronics*, vol. 10, no. 2, p. 155, Jan. 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/2/155

[4] M. Chauwin, X. Hu, F. Garcia-Sanchez, N. Betrabet, A. Paler, C. Moutafis, and J. S. Friedman, "Skyrmion Logic System for Large-Scale Reversible Computation," *Physical Review Applied*, vol. 12, no. 6, p. 064053, Dec. 2019. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevApplied.12.064053

[5] Y.-S. Hsieh, P.-C. Huang, P.-X. Chen, Y.-H. Chang, W. Kang, M.-C. Yang, and W.-K. Shih, "Shift-Limited Sort: Optimizing Sorting Performance on Skyrmion Memory-Based Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4115–4128, Nov. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9211559/

[6] Y. Wang, L. Ni, C.-H. Chang, and H. Yu, "DW-AES: A Domain-Wall Nanowire-Based AES for High Throughput and Energy-Efficient Data Encryption in Non-Volatile Memory," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 11, pp. 2426–2440, Nov. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7484726/

[7] C. Lucchese, F. M. Nardini, S. Orlando, R. Perego, N. Tonellotto, and R. Venturini, "QuickScorer: A Fast Algorithm to Rank Documents with Additive Ensembles of Regression Trees," in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Santiago Chile: ACM, Aug. 2015, pp. 73–82. [Online]. Available: https://dl.acm.org/doi/10.1145/2766462.2767733

[8] ——, "Exploiting CPU SIMD Extensions to Speed-up Document Scoring with Tree Ensembles," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. Pisa Italy: ACM, Jul. 2016, pp. 833–836. [Online]. Available: https://dl.acm.org/doi/10.1145/2911451.2914758

[9] T. Ye, H. Zhou, W. Y. Zou, B. Gao, and R. Zhang, "RapidScorer: Fast Tree Ensemble Evaluation by Maximizing Compactness in Data Level Parallelization," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. London United Kingdom: ACM, Jul. 2018, pp. 941–950. [Online]. Available: https://dl.acm.org/doi/10.1145/3219819.3219857

[10] [Online]. Available: https://github.com/tud-ccc/RTSim

[11] [Online]. Available: https://github.com/mnoorl/RTSim

[12] Z. Chen, Q. Deng, N. Xiao, K. Pruhs, and Y. Zhang, "DWMAcc: Accelerating Shift-based CNNs with Domain Wall Memories," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, pp. 1–19, Oct. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3358199

[13] D. A. Allwood, G. Xiong, C. C. Faulkner, D. Atkinson, D. Petit, and R. P. Cowburn, "Magnetic Domain-Wall Logic," vol. 309, 2005.

[14] Y. Wang, H. Yu, L. Ni, G.-B. Huang, M. Yan, C. Weng, W. Yang, and J. Zhao, "An Energy-Efficient Nonvolatile In-Memory Computing Architecture for Extreme Learning Machine by Domain-Wall Nanowire Devices," *IEEE Transactions on Nanotechnology*, vol. 14, no. 6, pp. 998–1012, Nov. 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7128727/

[15] W. Kang, B. Wu, X. Chen, D. Zhu, Z. Wang, X. Zhang, Y. Zhou, Y. Zhang, and W. Zhao, "A Comparative Cross-layer Study on Racetrack Memories: Domain Wall vs Skyrmion," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 16, no. 1, pp. 1–17, Jan. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3333336

[16] T.-Y. Yang, M.-C. Yang, J. Li, and W. Kang, "Permutation-Write: Optimizing Write Performance and Energy for Skyrmion Racetrack Memory," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, Jul. 2020, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9218642/

[17] B. Liu, S. Gu, M. Chen, W. Kang, J. Hu, Q. Zhuge, and E. H.-M. Sha, "An Efficient Racetrack Memory-Based Processing-in-Memory Architecture for Convolutional Neural Networks," in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. Guangzhou, China: IEEE, Dec. 2017, pp. 383–390. [Online]. Available: https://ieeexplore.ieee.org/document/8367291/

[18] Y.-S. Hsieh, P.-C. Huang, Y.-H. Chang, B.-J. Chen, W. Kang, and W.-K. Shih, "Granularity-Driven Management for Reliable and Efficient Skyrmion Racetrack Memories," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 95–111, Jan. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9771090/

[19] A. A. Khan, F. Hameed, R. Blasing, S. Parkin, and J. Castrillon, "RTSim: A Cycle-Accurate Simulator for Racetrack Memories," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 43–46, Jan. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8642352/

[20] W. Kang, Y. Huang, X. Zhang, Y. Zhou, W. Lv, and W. Zhao, "Skyrmions as Compact, Robust and Energy-Efficient Interconnects for Domain Wall (DW)-based Systems."

[21] Y. Pan, P. Ouyang, Y. Zhao, S. Yin, Y. Zhang, S. Wei, and W. Zhao, "A Skyrmion Racetrack Memory based Computing In-memory Architecture for Binary Neural Convolutional Network," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. Tysons Corner VA USA: ACM, May 2019, pp. 271–274. [Online]. Available: https://dl.acm.org/doi/10.1145/3299874.3318015

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/2024716.2024718

[23] S. Koschel, S. Buschjäger, C. Lucchese, and K. Morik, "Fast Inference of Tree Ensembles on ARM Devices," May 2023, arXiv:2305.08579 [cs]. [Online]. Available: http://arxiv.org/abs/2305.08579

[24] T. Qin and T. Liu, "Introducing LETOR 4.0 datasets," *CoRR*, vol. abs/1306.2597, 2013. [Online]. Available: http://arxiv.org/abs/1306.2597

[25] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2016, pp. 785–794, arXiv:1603.02754 [cs]. [Online]. Available: http://arxiv.org/abs/1603.02754

[26] A. Lambora, K. Gupta, and K. Chopra, "Genetic Algorithm- A Literature Review," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*. Faridabad, India: IEEE, Feb. 2019, pp. 380–384. [Online]. Available: https://ieeexplore.ieee.org/document/8862255/