Received 14 September, 2019; revised 17 October, 2019; accepted 27 October, 2019.
Date of publication 4 November 2019; date of current version 6 February 2020.

Digital Object Identifier 10.1109/JXCDC.2019.2951157

Spintronic In-Memory Pattern Matching

ZAMSHED I. CHOWDHURY[1] (Student Member, IEEE), S. KAREN KHATAMIFARD[1] (Member, IEEE), ZHENGYANG ZHAO[1] (Student Member, IEEE), MASOUD ZABIHI[1] (Student Member, IEEE), SALONIK RESCH[1] (Student Member, IEEE), MEISAM RAZAVIYAYN[2] (Member, IEEE), JIAN-PING WANG[1] (Fellow, IEEE), SACHIN SAPATNEKAR[1] (Fellow, IEEE), and ULYA R. KARPUZCU[1] (Member, IEEE)

[1]Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA
[2]Department of Industrial & Systems Engineering, University of Southern California, Los Angeles, CA 90089 USA

CORRESPONDING AUTHOR: Z. I. CHOWDHURY (chowh005@umn.edu)

This work was supported in part by NSF under Grant SPX-1725420.
This article has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors.

ABSTRACT Traditional Von Neumann computing is falling apart in the era of exploding data volumes as the overhead of data transfer becomes forbidding. Instead, it is more energy-efficient to fuse compute capability with memory where the data reside. This is particularly critical to pattern matching, a key computational step in large-scale data analytics, which involves repetitive search over very large databases residing in memory. Emerging spintronic technologies show remarkable versatility for the tight integration of logic and memory. In this article, we introduce SpinPM, a novel high-density, reconfigurable spintronic in-memory pattern matching spin–orbit torque (SOT)—specifically spin Hall effect (SHE)—substrate, and demonstrate the performance benefit SpinPM can achieve over conventional and near-memory processing systems.

INDEX TERMS Computational random access memory, pattern matching, processing in memory, spin Hall effect (SHE) magnetic tunnel junction (MTJ).

I. INTRODUCTION

CLASSICAL computing platforms are not optimized for efficient data transfer, which complicates large-scale data analytics in the presence of exponentially growing data volumes. Imbalanced technology scaling further exacerbates this situation by rendering data communication, and not computation, a critical bottleneck [1]. Specialization in hardware cannot help in this case unless conducted in a data-centric manner.

Tight integration of compute capability into the memory, processing in memory (PIM), is especially promising as the overhead of data transfer becomes forbidding at scale. The rich design space for PIM spans full-fledged processors and coprocessors residing in memory [2]. Until the emergence of 3-D stacking, however, the incompatibility of the state-of-the-art logic and memory technologies prevented practical prototype designs. Still, 3-D stacking can only achieve near-memory processing (NMP) [3]–[5]. The main challenge remains to be fusing computation and memory without violating array regularity.

Emerging spintronic technologies show remarkable versatility for the tight integration of logic and memory. This article introduces a high-density, reconfigurable spintronic in-memory compute substrate for pattern matching, SpinPM, which fuses computation and memory by using spin–orbit torque (SOT)— specifically, spin Hall effect (SHE)—as the switching principle to perform *in situ* computation in spintronic memory arrays. The basic idea is to add compute capability to the magnetic tunnel junction (MTJ)-based memory cell [6], [7], without breaking the array regularity; thereby, each memory cell can participate in gate-level computation as an input or as an output. Computation is not disruptive, i.e., memory cells acting as gate inputs do not lose their stored values.

SpinPM can implement different types of basic Boolean gates to form a functionally complete set; therefore, there is no fundamental limit to the types of computation. Each column in a SpinPM array can have only one active gate at a time, and however, computation in all columns can proceed in parallel. SpinPM provides true in-memory computing by reconfiguring cells within the memory array to implement logic functions. As all cells in the array are identical, inputs and outputs to logic gates do not need to be confined to a specific physical location in the array. In other words, SpinPM can initiate computation at any location in the memory array.

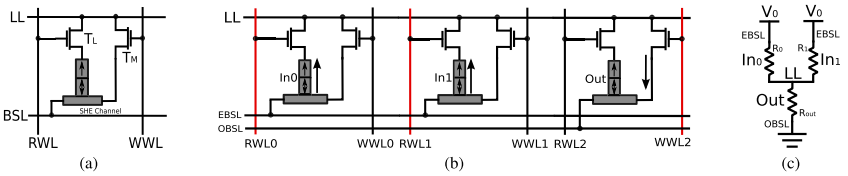

FIGURE 1. (a) SpinPM cell. (b) Two-input gate formation in the array. (c) Two-input NOR gate circuit equivalent.

Pattern matching is at the core of many important large-scale data analytics applications, ranging from bioinformatics to cryptography. The most prevalent form is string matching via repetitive search over very large reference databases residing in memory. Therefore, compute substrates, such as SpinPM, that collocate logic and memory to prevent slow and energy-hungry data transfers at scale, have great potential. In this case, each step of computation attempts to map a short character string to (the most similar substring of) an orders of magnitude longer character string and repeats this process for a very large number of short strings, where the longer string is fixed and acts as a reference.

In this article, we detail the end-to-end design of the SpinPM accelerator for large-scale string matching. The design covers device-, circuit-, and architecture-level details, including the programming interface. We evaluate SpinPM using representative benchmarks from emerging application domains to pinpoint its potential and opportunities for optimization. Specifically, Section II covers the basics of how SpinPM fuses compute with memory. Section III introduces a SpinPM implementation for pattern (string) matching; Sections IV and V provide the evaluation; Section VI compares and contrasts SpinPM to related work. Finally, Section VII concludes this article.

II. BASICS

A. FUSING COMPUTATION AND MEMORY

Without loss of generality, SpinPM adapts computational RAM (CRAM) [8] as the spintronic PIM substrate to accelerate large-scale string matching. In its most basic form, a CRAM array is essentially a 2-D magnetoresistive RAM (MRAM). When compared to the standard 2T(ransistor)1M(TJ) MRAM cell, however, the array features an additional logic line (LL) per cell —as shown in Fig. 1(a) and (b)—to connect cells to perform logic operations. A CRAM cell can operate as a regular MRAM memory cell or serve as an input/output to a logic gate.

Each MTJ consists of two layers of ferromagnets, termed pinned and free layers, separated by a thin insulator. The magnetic spin orientation of the pinned layer is fixed; the magnetic spin orientation of the free layer is controllable. SpinPM uses SHE MTJs, where—in order to separate read and write paths for more effective optimization—a heavy metal layer (i.e., the SHE channel) is juxtaposed with the free layer. The SHE channel has a high resistance that brings down the write current, leading to lower energy consumption. Changing the spin orientation of the free layer entails passing a (polarized) current through the SHE channel, where the current direction sets the free layer orientation. The relative orientation of the free layer with respect to the pinned layer, i.e., antiparallel (AP) or parallel (P), gives rise to two distinct MTJ resistance levels, i.e., R_{high} and R_{low}, which encode logic 1 and 0, respectively.

1) MEMORY CONFIGURATION

When the array is configured as a memory, the LL is active, i.e., connected to a voltage source. In the following, we detail the configuration for read word line (RWL) and write word line (WWL), considering various memory operations.

1) *Data Retention:* The RWL and WWL are set to 0 to isolate the cells and to prevent current flow through the MTJ and the SHE channel (which we refer to together as the SHE-MTJ).
2) *Read:* RWL is set to 1, to connect each SHE-MTJ to its LL. WWL is set to 0. A small voltage pulse applied between LL and bit select line (BSL) induces a current through the SHE-MTJ, which is a function of the resistance level (i.e., logic state) and in turn a sense amplifier attached to BSL captures.
3) *Write:* WWL is set to 1 and transistor T_M is switched ON to connect the SHE channel to its LL. RWL is set to 0. A large enough voltage pulse (in the order of the supply voltage) is applied between LL and BSL to induce a large enough current through LL and the SHE channel to change the spin orientation of the free layer.

2) LOGIC CONFIGURATION

In the logic mode, LL establishes the connection between inputs and outputs of logic gates. We also distinguish between two sets of BSL: even BSL (EBSL) and odd BSL (OBSL). These are utilized to connect all cells participating in computation, on a per column basis, as input and output cells. Such cells may act as logic gate inputs or outputs, where the only restriction is having all inputs connected on the same type of BSL and the output on the different types. For each SpinPM input cell participating in computation, RWL is set to 1 to connect its MTJ with LL. On the other hand, for each SpinPM output cell participating in computation, WWL is set to 1 to turn the switch T_M on, which in turn connects the SHE channel to the LL. A voltage pulse applied between EBSL and OBSL induces a current, dependent on the resistance levels of input cells, through the SHE channel of the output cell. As an example, Fig. 1(b) shows the formation of a two-input logic gate in the array, where cells labeled by "0," "1," and "2" correspond to the inputs In_0, In_1, and the output Out, respectively. The output cell is preset to a known logic value ("0" or "1"), depending on the type of logic operation to perform, by a standard write. Fig. 1(c) shows the equivalent circuit; OBSL is grounded, while EBSL is set to voltage V_0. The value of V_0 determines the current through the input MTJs, I_0 and I_1, as a function of their resistance values R_0 and R_1. Each input resistance captures the resistance of the corresponding SHE-MTJ, whereas the output resistance R_{Out} is only the SHE channel resistance of the output cell. Input and output cells are connected to LL by setting the respective RWL and WWL to 1 [colored red in Fig. 1(b)]. $I_{\text{Out}} = I_0 + I_1$

TABLE 1. Two-input NOR truth table (Out preset = 0).

In_0	In_1	Out	$I_{Out} = I_0 + I_1$	
0 (R_{low})	0 (R_{low})	1	I_{00}	$> I_{crit}$
0 (R_{low})	1 (R_{high})	0	I_{01}	$< I_{crit}$
1 (R_{high})	0 (R_{low})	0	$I_{10} = I_{01}$	$< I_{crit}$
1 (R_{high})	1 (R_{high})	0	I_{11}	$< I_{crit}$

flows through R_{Out}. If I_{Out} is higher than the critical switching current I_{crit}, it will change the free layer orientation of Out's MTJ and, thereby, the logic state of Out. Otherwise, Out will keep its previous (preset) state.

We can easily expand this example to more than two inputs. The key observation is that we can change the logic state of the output as a function of the logic states of the inputs, within the array, and voltages applied between BSLs of the participating cells dictate how such functions look like.

Continuing with the example from Fig. 1(b) and (c), let us try to implement a universal, two-input NOR gate. Table 1 provides the truth table. Out would be 0 in this case for all input combinations, but $In_0 = 0$ and $In_1 = 0$, which incurs the lowest R_0 and R_1 and, hence, the highest $I_{\text{Out}} = I_0 + I_1$. Let us refer to this value of I_{Out} as I_{00}, following Table 1. Accordingly, if we preset Out to 0 (before computation starts) and determine V_0 such that I_{00} does exceed I_{crit}, while both I_{11} and $I_{01} = I_{10}$ do not, Out would not switch from (its preset value) 0 to 1 for all input combinations, but $In_0 = 0$ and $In_1 = 0$.

As Boolean gates of practical importance (such as NOR) are commutative, a single voltage level at the BSLs of the inputs suffices to define a specific logic function. Each voltage level can serve as a signature for a specific logic gate. In the following, we will refer to such as V_{gate}. In the earlier example, $V_{\text{gate}} = V_{\text{NOR}}$. While NOR gate is universal, we can implement different types of logic gates following a similar methodology for mapping the corresponding truth tables to the SpinPM array.

B. BASIC COMPUTATIONAL BLOCKS

We will next introduce basic SpinPM computational blocks for pattern matching, including inverters (INV), buffers (COPY), three-input and five-input majority (MAJ) gates, and 1-bit full adders.

1) INV

INV is a single-input gate. Still, we can follow a similar methodology to the NOR implementation (see Table 1); preset output to 0 and define V_{INV} in a way such that I_0 (I_1), i.e., the current if the input is 0 (1) is higher (lower) than I_{crit} such that the output does (not) switch from the preset 0 to 1. By definition, $I_1 < I_0$ applies, as $R_1 > R_0$. However, in this case, the input cell will inevitably switch, as well, due to the same current flowing through the input and output cells. INV, therefore, is a destructive gate, which still can be very useful if the input data are not needed in the subsequent steps of computation—as it is the case in our case studies.

2) COPY

For 1-bit copy, two back-to-back invocations of INV can suffice. A more time and energy-efficient implementation, however, can perform the same function in one step as follows; preset output to 1 and define V_{COPY} in a way such that I_0 (I_1), i.e., the current if the input is 0 (1) is higher (lower) than I_{crit} such that the output does (not) switch from the preset 1 to 0. By definition, $I_1 < I_0$ applies, as $R_1 > R_0$.

TABLE 2. XOR implementation.

In_0	In_0	S_1= NOR(In_0,In_1)	S_2= COPY(S_1)	Out= TH(In_0,In_1,S_1,S_2)
0	0	1	1	0
0	1	0	0	1
1	0	0	0	1
1	1	0	0	0

3) MAJ

MAJ gates accept an odd number of inputs and assign the majority (logic) state across all inputs to the output. The structure for a three-input MAJ3 or five-input MAJ5 gate is not any different from the circuit structure in Fig. 1(c), except the higher number of inputs. As an example, I_{Out} of the MAJ3 gate assumes its highest value for the 000 assignment of the three inputs—as the resistances of the three inputs, R_0, R_1, and R_2, assume their lowest value for 000. Any input assignment having at least one 1 gives rise to a lower I_{Out} than I_{000} and having at least two 1s, to an even lower I_{Out}. Finally, I_{Out} reaches its minimum for the input assignment 111, for which the inputs assume their highest resistance. Accordingly, we can preset the output to 1 and define V_{MAJ3} in a way such that I_{Out} remains higher than I_{crit} if the three inputs have less than two 1s such that Out switches from the preset 1 to 0, to match the input majority. We can symmetrically define V_{MAJ5}, assuming a preset of 1.

4) XOR

XOR is an especially useful gate for comparison, and however, a single-gate SpinPM implementation is not possible: In this case, we need Out (not) to switch for 00 and 11, but not for 01 and 10, if the preset is 1 (0). However, due to $I_{00} > I_{01} = I_{10} > I_{11}$ and assuming a preset of 1, we cannot let both I_{00} and I_{11} remain higher than I_{crit} (such that Out switches), while $I_{01} = I_{10}$ remains lower than I_{crit} (such that Out does not switch). The same observation holds for a preset of 0 as well.

We can implement XOR using a combination of universal SpinPM gates, such as NOR; thereby, each XOR takes at least four steps (i.e., logic evaluations). For pattern matching, we will rely on a more efficient three-step implementation (see Table 2): In Step 1, we compute $S_1 =$ NOR (In_0 and In_1). In Step 2, we perform $S_2 = \text{COPY}(S_1)$. In the final Step 3, we invoke a four-input thresholding TH function, which renders a 1 only if its inputs contain more than two zeros: Out = $\text{TH}(In_0, In_1, S_1, S_2)$. TH has a preset of 0, and the operating principle is very similar to the majority gates, except that TH accepts an even number of inputs. We can further optimize this implementation and fuse Steps 1 and 2 by implementing NOR as a two-output gate.

5) FULL ADDER

A full adder has three inputs: In_0, In_1, and carry-in C_i. The two outputs are Sum and the carry-out C_o. Like other logic functions, we can implement this adder using NOR gates. However, an implementation based on a pair of MAJ gates reduces the required number of steps significantly [9]. Fig. 2 shows a step-by-step overview.

Step 1: $C_o = MAJ(In_0, In_1, C_i)$.
Step 2: $S_1 = INV(C_o)$.
Step 3: $S_2 = COPY(S_1)$.
Step 4: $Sum = MAJ(In_0, In_1, C_i, S_1, S_2)$.

C. COLUMN-LEVEL PARALLELISM

SpinPM can perform only one type of logic function in a column at a time. This is because there is only one LL that

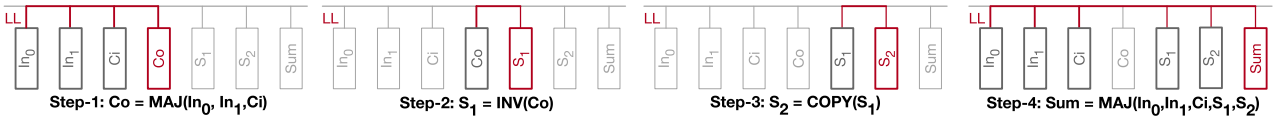

FIGURE 2. Full adder implementation [8]. Output of each gate is depicted in red.

spans the entire column, and any cell within the column to participate in computation gets directly connected to this LL (see Section II-A).

On the other hand, the voltage levels on BSLs determine the type of the logic function, where each BSL spans an entire column. Furthermore, in each row, each of WWL and RWL—which connect cells participating in computation to LL—spans an entire row. Therefore, all columns can perform the very same logic function in parallel on the same set of rows. In other words, SpinPM supports a special form of single-instruction multiple-data (SIMD) parallelism, where instruction translates into logic gate/operation and data into input cells in each column, across all columns, which span the very same rows.

Multistep operations are carried out in each column independently, one step at a time, while all columns operate in parallel. The output from each logic step performed within a column stays in that column and can serve as an input to the subsequent logic steps (performed in the same column). All columns follow the same sequence of operations at the same time. In case of a multibit full adder, as an example, the carry and sum bits are generated in the same column as the input bits, which are used in subsequent 1-bit additions in the very same column.

To summarize, SpinPM can have part or all columns computing in parallel or the entire array serving as memory. Regular memory reads and writes cannot proceed simultaneously with computation. Large-scale pattern matching problems can greatly benefit from this execution model, as we are going to demonstrate next.

III. SPINTRONIC PATTERN MATCHING

Pattern matching is a key computational step in large-scale data analytics. The most common form by far is character string matching, which involves repetitive search over very large databases residing in memory. Therefore, compute substrates, such as SpinPM, that collocate logic and memory to avoid the latency and energy overhead of expensive data transfers, have great potential. Moreover, comparison operations dominate the computation, which represents excellent acceleration targets for SpinPM. As a representative and important large-scale string matching problem, in the following, we will use deoxyribonucleic acid (DNA) sequence prealignment [10] as a running example without loss of generality and expand SpinPM's evaluation to other string matching benchmarks in Section V.

At each step, DNA sequence prealignment tries to map a short character string to (the most similar substring of) an orders of magnitude longer character string and repeats this process for a very large number of short strings, where the longer string is fixed and acts as a reference. For each string, the characters come from the alphabet A(denine), C(ytosine), G(uanine), and T(hymine). The long string represents a complete genome; short strings represent short DNA sequences (from the same species). The goal is to extract the region of the reference genome to which the short DNA sequences correspond to. We will refer to each short DNA sequence as a pattern and the longer reference genome as reference.

Aligning each pattern to the most similar substring of the reference usually involves character-by-character comparisons to derive a similarity score, which captures the number of character matches between the pattern and the (aligned substring of the) reference. Improving the throughput performance in terms of number of patterns processed per second in an energy-efficient manner is especially challenging, considering that a representative reference (i.e., the human genome) can be around 10^9 characters long, at least 2 bits are necessary to encode each character, and a typical pattern data set can have hundreds of millions patterns to match [11], where SpinPM can help due to reduced data transfer overhead and (column) parallel comparison/similarity score computations.

By effectively pruning the search space, DNA prealignment can significantly accelerate DNA sequence alignment—which, besides complex pattern matching in the presence of errors, include preprocessing and postprocessing steps typically spanning (input) data transformation for more efficient processing, search space compaction, or (output) data reformatting. The execution time share of pattern matching (accounting for possible complex errors in patterns and the reference) can easily reach 88% in highly optimized GPU implementations of popular alignment algorithms [12].[1] In the following, we will only cover basic pattern matching (which can still account for basic error manifestations in the patterns and the reference) within the scope of prealignment.

Mapping any computational task to the SpinPM array translates into cooptimizing the data layout, data representation, and the spatiotemporal schedule of logic operations, to make the best use of SpinPM's column-level parallelism. This entails distribution of the data to be processed, i.e., the reference and the patterns, in a way such that each column can perform independent computations. The data representation itself, i.e., how we encode each character of the pattern and the reference strings, has a big impact on both the storage and the computational complexity. Specifically, data representation dictates not only the type but also the spatiotemporal schedule of (bitwise) logic operations. Spatiotemporal scheduling, on the other hand, should take intermediate results during computation into account, which may or may not be discarded (i.e., overwritten), and which may or may not overwrite existing data, as a function of the algorithm or array size limitations.

A. DATA LAYOUT AND DATA REPRESENTATION

We use the data layout captured in Fig. 3 by folding the long reference over multiple SpinPM columns. Each column

[1]For this implementation of the common Burrows–Wheeler–Aligner (BWA) algorithm, the time share of the pattern matching kernel, inexact_match_caller, increases from 46% to 88%, as the number of base mismatches allowed (an input parameter to the algorithm) is varied from one to four (both representing typical values).

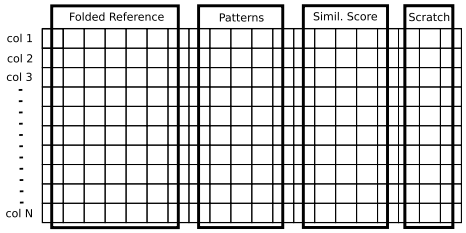

FIGURE 3. **Data layout per SpinPM array.**

has four dedicated compartments to accommodate a fragment of the folded reference: one pattern, the similarity score (for the pattern when aligned to the corresponding fragment of the reference), and intermediate data (which we will refer to as scratch). The same format applies to each column, for efficient column-parallel processing. Each column contains a different fragment of the reference.

We determine the number of rows allocated for each of the four compartments as follows. In the DNA prealignment problem, the reference corresponds to a genome, and therefore, it can be very long. The species determines the length. As a case study for large-scale pattern matching, in this article, we will use approximately 3×10^9 character-long human genome. Each pattern, on the other hand, represents the output from a DNA sequencing platform, which biochemically extracts the location of the four characters (i.e., bases) in a given (short) DNA strand. Hence, the sequencing technology determines the maximum length per pattern, and around 100 characters is typical for modern platforms processing short DNA strands [13]. The size of the similarity score compartment, to keep the character-by-character comparison results, is a function of the pattern length. Finally, the size of the scratch compartment depends on both the reference fragment and the pattern length.

While the reference length and the pattern length are problem-specific constants, the (reference) fragment length (as determined by the folding factor) is a SpinPM design parameter. By construction, each fragment should be at least as long as each pattern. The maximum fragment length, on the other hand, is limited by the maximum possible SpinPM column height, considering the maximum affordable capacitive load (hence, *RC* delay) on column-wide control lines, such as BSL and LL. However, column-level parallelism favors shorter fragments (for the same reference length). The shorter the fragments, the more columns would the reference occupy, and the more columns, hence regions of the reference, would be ''pattern-matched'' simultaneously.

For data representation, we simply use 2 bits to encode the four (base) characters, and hence, each character-level comparison entails two bit-level comparisons.

B. PROOF OF CONCEPT SpinPM DESIGN

SpinPM comprises two computational phases, which Algorithm 1 captures at the column level: match, i.e., aligned bitwise comparison and similarity score computation. As each column performs the very same computation in parallel, in the following, we will detail column-level operations.

In Algorithm 1, len (fragment) and len (pattern) represent the (character) length of the reference fragment and the pattern, respectively; and loc is the index of the fragment string where we align the pattern for comparison. The computation in each column starts with aligning the fragment and the

Algorithm 1 Two-Phase Pattern Matching at Column Level

```
loc = 0
while loc < len(fragment)−len(pattern) do
    Phase-1: Match (Aligned Comparison)
    align pattern to location loc of reference fragment;
    (bit-wise) compare aligned pattern to fragment
    Phase-2: Similarity Score Computation
    count the number of character-wise matches;
    derive similarity score from count
    loc++
end while
```

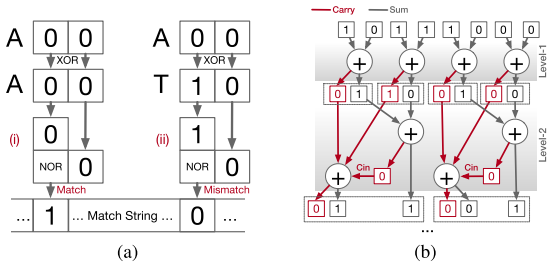

FIGURE 4. **(a) Aligned bitwise comparison. (b) Adder reduction tree used for similarity score computation.**

pattern string, from the first character location of the fragment onward. For each alignment, a bitwise comparison of the fragment and pattern characters comes next. The outcome is a len (pattern) bits long string, where a 1 (0) indicates a characterwise (mis)match. We will refer to this string as the match string. Hence, the number of 1s in the match string acts as a measure for how similar the fragment and the pattern are when aligned at that particular character location (loc per Algorithm 1).

A reduction tree of 1-bit adders counts the number of 1s in the match string to derive the similarity score. Once the similarity score is ready, the next iteration starts. This process continues until the last character of the pattern reaches the last character of the fragment when aligned.

Phase 1 (Match, i.e., Aligned Comparison): Each aligned characterwise comparison gives rise to two bitwise comparisons, each performed by a two-input XOR gate. Fig. 4(a) shows an example, where we compare the base character ''A'' (encoded by ''00'') of the fragment with the base character ''A'' (i) and ''T'' (encoded by ''10'') (ii), of the pattern. A two-input NOR gate converts the 2-bit comparison outcome to a single bit, which renders a 1 (0) for a characterwise (mis)match. Recall that a NOR gate outputs a 1 only if both of its inputs are 0 and an XOR gate generates a 0 only if both of its inputs are equal. The implementation of these gates follows Section II-B.

SpinPM can only have one gate active per column at a time (see Section II-C). Therefore, for each alignment (i.e., for each loc or iteration of Algorithm 1), such a 2-b comparison takes place len (pattern) times in each column, one after another. Thereby, we compare all characters of the aligned pattern to all characters of the fragment, before moving to the next alignment (at the next location loc per Algorithm 1). Having said that, each such 2-bit comparison takes place in parallel over all columns, where the very same rows participate in computation.

Phase 2 (Similarity Score Computation): For each alignment (i.e., iteration of Algorithm 1), once all bits of the match string are ready—i.e., the characterwise comparison of the fragment and the aligned pattern string is complete for all characters, we count the number of 1s in the match string to calculate the similarity score. A reduction tree of 1-bit adders performs the counting, as captured in Fig. 4(b), with the carry and sum paths shown explicitly for the first two levels. The top row corresponds to the contents of the match string and each $\oplus$ to a 1-bit adder from Section II-B.

len (pattern), the pattern length in characters, is equal to the match string length in bits. Hence, the number of bits required to hold the final bit count (i.e., the similarity score) is $N = \lfloor log_2[len(pattern)] \rfloor + 1$. A naive implementation for the addition of len (pattern) number of bits requires len (pattern) steps, with each step using an N-bit adder, to generate an N-bit partial sum toward the N-bit end result. For a typical pattern length of around 100 [13], this translates into approximately 100 steps, with each step performing an $N = 7$ bit addition. Instead, to reduce both the number of steps and the operand width per step, we adopt the reduction tree of 1-bit adders from Fig. 4(b). Each level adds bits in groups of two, using 1-bit adders. For a typical pattern length of around 100 [13], we thereby reduce the complexity to 188 1-bit additions in total.

Alignment under basic error manifestations in the pattern and the reference is also straightforward in this case. For DNA sequence alignment, the most common errors take the form of substitutions (due to sequencing technology imperfections and genetic mutations), where a character value assumes a different value than actual value [14]–[17]. We can set a tolerance value t (in terms of number of mismatched characters) based on expected error rates and pass an alignment as a ''match'' if less than t characters mismatch.

Assignment of Patterns to Columns: In each SpinPM array, we can process a given pattern data set in different ways. We can assign a different pattern to each column, where a different fragment of the reference resides, or distribute the very same pattern across all columns. Either option works as long as we do not miss the comparison of a given pattern to all fragments of the reference. In the following, we will stick to the second option, without loss of generality. This option eases capturing alignments scattered across columns (i.e., where two consecutive columns partially carry the most similar region of the reference to the given pattern). A large reference can also occupy multiple arrays and give rise to scattered alignments at array boundaries, in which column replication at array boundaries can address.

Many pattern matching algorithms rely on different forms of search space pruning to prevent unnecessary brute-force search across all possibilities. At the core of such pruning techniques lies indexing the reference, which is known ahead of time, in order to direct detailed search for any given pattern to the most relevant portion of the reference (i.e., the portion that most likely incorporates the best match). The result is pattern matching at a much higher throughput. SpinPM, as well, features search space pruning for efficient pattern matching. The idea is chunking each pattern and the reference into substrings of known length and creating a hash (bit) for each substring; thereby, both the pattern and the reference become bit vectors, of much shorter length than their actual representations. Search space pruning in SpinPM simply translates into the bitwise comparison of each pattern bit vector to the (longer) reference bit vector, within the memory array, in a similar fashion to the actual full-fledged pattern mapping algorithm, considering all possible alignments exploiting SpinPM's massive parallelism at the column level. Thereby, we eliminate unnecessary attempts for full-fledged matching (using actual data representation and not hashes).

TABLE 3. Technology parameters.

	SHE	STT Near-term	STT Long-term
MTJ Type		Interfacial PMTJ	
MTJ Diameter (nm)	10	45	10
TMR (%)	100	133 [21]	500
RA Product ($\Omega\mu m^2$)	20	5	1 [22]
Critical Current I_{crit} (μA)	3.0 (SHE) 3.9 (MTJ)	100	3.95
Switching Latency (ns)	1	3 [23]	1 [21]
R_P ($K\Omega$)	253.97	3.15	12.7
R_{AP} ($K\Omega$)	507.94	7.34	76.39
R_{SHE} ($K\Omega$)	64	–	–
Write Latency (ns)	1.72	3.65	1.72
Read Latency (ns	1.24	1.21	1.24
Write Energy (fJ)	0.4	12.41	2.62
Read Energy (fJ)	0.29	0.29	0.29
V_{INV} (V)	1.05-1.81	0.84–1.3	0.23–0.48
V_{COPY} (V)	1.05-1.81	0.84–1.3	0.23–0.48
V_{NOR} (V)	0.62-0.75	0.68–0.74	0.20–0.22
V_{MAJ3} (V)	0.53-0.61	0.65–0.69	0.20–0.21
V_{MAJ5} (V)	0.40-0.43	0.61–0.62	0.19–0.20
V_{TH} (V)	0.43-0.86	0.62–0.63	0.19–0.20

IV. EVALUATION SETUP

A. TECHNOLOGY PARAMETERS

Table 3 provides the technology parameters for a representative SHE-MTJ and a more conventional spin–torque transfer (STT)-MTJ (near-term and projected long-term). The critical current I_{crit} refers to an MTJ switching probability of 50%, which would incur a high write error rate (WER). To compensate, when deriving gate latency and energy values, we conservatively assume a $2\times$ ($5\times$) larger I_{crit} for the near (long) term STT-MTJ. The corresponding value for SHE-MTJ is, by definition, lower due to spin Hall effect-based switching. The long-term STT specification is based on projections from the literature [18]. SHE-MTJ specification comes from [19]. We model access transistors after 22-nm (HP) PTM [20].

B. SIMULATION INFRASTRUCTURE

We developed a step-accurate simulator to capture the throughput performance and energy consumption of SpinPM-based pattern matching as a function of the technology parameters. We model the peripheral circuitry using NVSIM [24] to extract the row decoder, mux, precharge, and sense amplifier induced energy and latency overheads at 22 nm. NVSIM captures parasitic effects, such as the temperature impact on wire resistance.

C. ARRAY SIZE AND ORGANIZATION

It is evident that, depending on the pattern matching problem at hand, we might need SpinPM arrays ranging from modest to very large in size. The thought-provoking issue here is how to deal with sufficiently large arrays as it might restrict the design space, considering fabrication and circuit-level-design-related limitations. As an example, the

TABLE 4. Benchmark applications.

Benchmark	Problem Size	Pattern Length	Sub-Array Size
DNA	3G char.	100 char.	512×512
Bit count	1000000 32-bit vectors	1-bit	512×512
String Matching	10396542 words	10 char. string	512×512
Rivest Cipher 4	10396542 words	248 bit	512×512
Word count	1471016 words	32 bits	512×512

proof-of-concept implementation requires 300 arrays of 10k columns and around 2k rows each for the string matching case study from genomics. This renders a total size of roughly 24 Mb per array, which is not excessively large. Still, the fabrication technology might not be mature enough to synthesize such an array. Commercial MRAM manufacturers address this challenge by banking. For example, EverSpin [25] uses eight banks in its 256 Mb (32 Mb $\times$ 8) MRAM product. Distributing array capacity to banks helps satisfy the latency and energy requirement per access as well. For SpinPM-based pattern matching, we too are inclined to use a hierarchy of banks to enhance scalability. While a clever data layout, operation scheduling, and parallel activation of banks can mask the time overhead, the energy and area overhead would be largely due to replication of control hardware across banks. The most straightforward option for banked SpinPM would be to treat each bank simply as an individual array, which would map even shorter fragments of the reference to patterns from the input pattern data set.

D. SEARCH SPACE PRUNING

Without loss of generality, we use GRIM filter [26] to convert the patterns and the reference to bit vectors. Except for bit-vector generation, all operations (including bit-vector mapping) are implemented entirely in SpinPM arrays. We synthesize the dedicated logic for bit-vector generation in 22 nm to extract the energy and time overheads. We account for the overhead of search space pruning throughout the evaluation, which spans bit-vector generation and matching in the SpinPM arrays.

E. BENCHMARKS

We evaluate SpinPM using four pattern matching applications [which also include common computational kernels for pattern matching such as bit count (BC)], besides the running example of DNA sequence prealignment throughout this article. Table 4 tabulates these applications along with the corresponding problem sizes.

DNA sequence prealignment (DNA) is our running case study throughout this article. We use a real human genome, NCBI36.54, from the 1000 genomes project [27] as the reference and 3M 100-base character long real patterns from SRR1153470 [28].

BC [29] counts the number of ones in a set of vectors of fixed length. This includes the addition of bits in the vectors and the subsequent addition of all individual counts. The input vectors are mapped to the columns of SpinPM to exploit parallelism.

String match (SM) [30] matches a search string with a prestored reference string to identify the part of the reference of highest or lowest similarity. Space-separated string segments and the search substring (which forms the pattern) are mapped to SpinPM columns such that all searches are performed in parallel.

Rivest Cipher 4 (RC4) is a popular stream cipher. Upon generating a cipher key, i.e., a string, it performs bitwise XOR on the cipher key and the text to cipher. The same key is used to decipher the text as well. Segments of input text and the cipher key are mapped to SpinPM columns.

Word count (WC) [30] counts the number of occurrences of specific words in an input text file through word matching. The words are mapped to SpinPM columns along with search words, and the word matching in each column is executed concurrently.

F. BASELINES FOR COMPARISON

1) NMP BASELINE

For NMP-based pattern matching, we use an hybrid memory cube (HMC) model based on the published data [3], which covers memory and logic layers, and communication links. To favor the NMP baseline, we ignore the power required to navigate the global wires between the memory controller and the logic layer, and intermediate routing elements. For the logic layer, we consider single-issue in-order cores, modeled after ARM Cortex A5 [31] with 1-GHz clock and 32-kB instruction and data caches. We first consider a total of 64 cores to provide parallel processing. For communication, we assume an HMC-like configuration with four communication links operating at their peak frequency of 160 GB/s. To derive the throughput performance, we use the same reference and input patterns to profile each benchmark. We then use the instruction and memory traces to calculate the throughput. We validated this model through CasHMC [32] simulations. For reference, we also include a hypothetical NMP variant, which includes 128 cores in the logic layer and incurs zero memory overhead.

2) SpinPM-STT BASELINE

To demonstrate the effect of different cell technologies, we also use an STT-MRAM-based SpinPM implementation as a baseline for comparison. STT-MTJ-based SpinPM (SpinPM-STT) performs logic operations following the same principle as the SHE-based (SpinPM-SHE) implementation, however, transposed [8].

V. EVALUATION

We next characterize benchmark applications in terms of match rate and compute efficiency when mapped to SHE-based SpinPM (SpinPM-SHE). We use match rate (in terms of number of patterns processed per second) for throughput and match rate per milliwatt for compute efficiency. Fig. 5(a) shows the match rates for SpinPM-SHE normalized to four baselines: NMP, a hypothetical variant of NMP with no memory overhead (NMP-Hyp), and two variants of STT-based SpinPM (near-term SpinPM-STT and projected SpinPM-STT), respectively. Overall, we observe that SpinPM-SHE outperforms NMP baselines in all benchmark applications. Moreover, in comparison to near-term SpinPM-STT, SpinPM-SHE shows a significant improvement in throughput performance and compute efficiency. All applications have smaller improvement with respect to NMP-Hyp compared with NMP since NMP-Hyp has no memory overhead and hence has a much higher match rate than NMP to start with.

Fig. 5(b) shows the outcome for compute efficiency. Generally, we observe a similar trend to match rate, with all

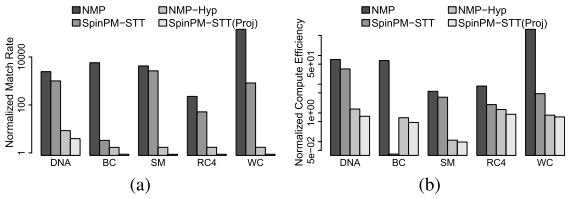

FIGURE 5. Throughput and energy efficiency of SpinPM-SHE. (a) Match rate (pattern/second). (b) Compute Efficiency (pattern/second/mW).

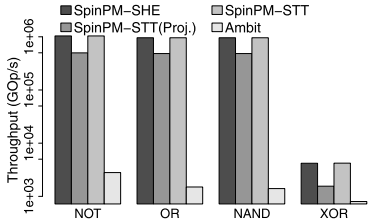

FIGURE 6. Throughput with respect to Ambit [33].

benchmarks (but BC) featuring $\geq 2\times$ improvement even with respect to the ideal baseline NMP-Hyp. Overall, BC shows the least benefit since BC has a lower compute to memory access ratio.

SpinPM-SHE performs significantly better, in terms of match rate and compute efficiency, than near-term SpinPM-STT due to smaller switching latency and energy consumption. Moreover, the match rate and compute efficiency of SpinPM-SHE are quite close to that of projected long-term STT-MTJ-based implementations.

A. IMPACT OF PROCESS VARIATION

We conclude the evaluation with a discussion on the impact of process variation, which, due to imperfections in manufacturing technology, may result in significant deviation in device parameters from their expected values. Both access transistors and the SHE-MTJ are subject to process variation. Being a relatively new technology, MTJ devices are more susceptible to process variation, which directly affects critical parameters such as switching current and switching latency. However, as MTJ technology matures, it is likely that it too will be able to reduce the impact of process variation.

One concern is variation in critical switching current, which can directly translate into variation in bias voltages on BSLs, i.e., V_{gate}, which determines the gate type. However, different SpinPM gates featuring close V_{gate} values (and hence subject to this type of variation) are usually distinguished either by a different value of the preset or a different number of inputs, which makes it unlikely that the gate functions would overlap with each other as a result of variation. We validated this observation assuming a variation in switching current by $\pm 5\%$, $\pm 10\%$, and $\pm 20\%$ for all evaluated gates implemented in the SpinPM array.

B. GATE-LEVEL CHARACTERIZATION

We next compare the throughput performance of SpinPM with Ambit [33] and Pinatubo [34]. Ambit reports a comparative bulk throughput with respect to CPU and GPU baselines, in executing basic logic operations on fixed-sized vectors of 1-bit operands. Pinatubo reports bitwise throughput of OR operation only, on a 2^{20} bit long vector. We considered the highest throughput (for a 128-row operation) reported by Pinatubo. To conduct a fair comparison, we assume the same vector size of 32 MB used in Ambit. Fig. 6 captures the outcome, with respect to Ambit, in terms of gigaoperations per second (GOPs), for NOT, OR, NAND, and XOR implementations. We observe a higher throughput for SpinPM-SHE across all of these bitwise operations. The high degree of parallelism and lack of actual data transfer within the array are the main reasons behind such improvement. For the more complex XOR, the throughput improvement for SpinPM-SHE and projected SpinPM-STT is $\approx 4\times$ over Ambit, whereas for near-term SpinPM-STT, the throughput for SpinPM-SHE and projected SpinPM-STT is only $1.34\times$. In comparison to OR throughput of Pinatubo, SpinPM-SHE has similar improvement as projected SpinPM-STT ($12\times$). For this comparison, we do not optimize data layout or operation scheduling for SpinPM.

VI. RELATED WORK

Without loss of generality, we base SpinPM on the spintronic PIM substrate CRAM that was briefly presented in [8] and evaluated for a single-neuron digit recognizer along with a small-scale 2-D convolution in [18]. CRAM is unique in combining multigrain (possibly dynamic) reconfigurability with true processing in memory semantics.

The conventional bitline computing substrates employ sense amplifier-based logic operations and cannot truly eliminate data movement overhead within the array boundary. The SRAM-based compute cache [35] can carry out different vector operations in the cache, but SpinPM supports a wider range of computations on much larger data than could fit in cache. Recent proposals for bitwise in memory computing include Ambit [33], DRISA [36], Pinatubo [34], and STT-CiM [37]. DRAM-based solutions, such as Ambit or DRISA, use modified sense amplifier-based designs. These designs support bitwise operations in DRAM but can only perform computation on a designated set of rows. Thus, to compute on an arbitrary row, the row must first be copied to these dedicated compute rows and then copied back once the computation is complete. While both designs feature high degrees of (column) parallelism, they suffer from data movement overheads within the array boundary. Pinatubo [34], on the other hand, can perform bitwise operations on data residing in multiple rows, using a specialized sense amplifier with variable reference voltage, which increases the susceptibility to variation.

VII. CONCLUSION

This article introduces SpinPM, a novel, reconfigurable spintronic pattern matching substrate for true in-memory pattern matching, which represents a key computational step in large-scale data analytics. When configured as memory, SpinPM is not any different than an MRAM array. Each MRAM cell, however, can act as an input or output to a logic gate on demand. Therefore, reconfigurability does not compromise memory density. Each column can have only one logic gate active at a time, but the very same logic operation can proceed in all columns in parallel. We implement a proof-of-concept SpinPM array with SHE-MTJ technology for large-scale string matching to pinpoint design bottlenecks and aspects subject to optimization. The encouraging results from Section V indicate a great potential for throughput performance and compute efficiency.

REFERENCES

[1] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[2] G. H. Loh *et al.*, "A processing in memory taxonomy and a case for studying fixed-function PIM," in *Proc. Workshop Near-Data Process. Conjunct MICRO*, 2013, pp. 1–4.

[3] *Hybrid Memory Cube (HMC)*. Accessed: Sep. 14, 2019. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.3-memory-FPGA/HC23.18.320-HybridCube-Pawlowski-Micron.pdf

[4] *Hybrid Bandwidth Memory (HBM)*. Accessed: Sep. 14, 2019. [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/hbm

[5] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM J. Res. Develop.*, vol. 59, nos. 2–3, pp. 17:1–17:14, Mar./May 52015.

[6] A. Lyle *et al.*, "Direct communication between magnetic tunnel junctions for nonvolatile logic fan-out architecture," *Appl. Phys. Lett.*, vol. 97, Sep. 2010, Art. no. 152504.

[7] J. Wang, H. Meng, and J.-P. Wang, "Programmable spintronics logic device based on a magnetic tunnel junction element," *Appl. Phys. Lett.*, vol. 97, no. 10, 2005, Art. no. 10D509.

[8] Z. Chowdhury *et al.*, "Efficient in-memory processing using spintronics," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 42–46, Jan./Jun. 2018.

[9] C. Augustine, G. Panagopoulos, B. Behin-Aein, S. Srinivasan, A. Sarkar, and K. Roy, "Low-power functionality enhanced computation architecture using spin-based devices," in *Proc. IEEE/ACM Int. Symp. Nanosc. Architectures*, Jun. 2011, pp. 129–136.

[10] R. Kaplan, L. Yavits, and R. Ginosar, "RASSA: Resistive pre-alignment accelerator for approximate DNA long read mapping," 2018, *arXiv: 1809.01127*. [Online]. Available: https://arxiv.org/abs/1809.01127

[11] S. S. Ajay, S. C. J. Parker, H. O. Abaan, K. V. F. Fajardo, and E. H. Margulies, "Accurate and comprehensive sequencing of personal genomes," *Genome Res.*, vol. 21, no. 9, pp. 1498–1505, 2011.

[12] P. Klus *et al.*, "BarraCUDA—A fast short read sequence aligner using graphics processing units," *BMC Res. Notes*, vol. 5, no. 1, p. 27, 2012.

[13] *Illumina Sequencing by Synthesis (SBS) Technology*. Accessed: Sep. 14, 2019. [Online]. Available: https://www.illumina.com/technology/next-generation-sequencing/sequencing-technology.html

[14] M. Schirmer, R. D'Amore, U. Z. Ijaz, N. Hall, and C. Quince, "Illumina error profiles: Resolving fine-scale variation in metagenomic sequencing data," *BMC Bioinf.*, vol. 17, no. 1, p. 125, 2016.

[15] J. M. Mullaney, R. E. Mills, W. S. Pittard, and S. E. Devine, "Small insertions and deletions (INDELs) in human genomes," *Hum. Mol. Genet.*, vol. 19, no. R2, pp. R131–R136, 2010.

[16] S.-M. Ahn *et al.*, "The first Korean genome sequence and analysis: Full genome sequencing for a socio-ethnic group," *Genome Res.*, vol. 19, no. 9, pp. 1622–1629, 2009.

[17] J. Wala *et al.*, "SvABA: Genome-wide detection of structural variants and indels by local assembly," *Genome Res.*, vol. 28, no. 4, pp. 581–591, 2018.

[18] M. Zabihi, Z. I. Chowdhury, Z. Zhao, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "In-memory processing on the spintronic CRAM: From hardware design to application mapping," *IEEE Trans. Comput.*, vol. 68, no. 8, pp. 1159–1173, Aug. 2019.

[19] M. Zabihi *et al.*, "Using spin-Hall MTJs to build an energy-efficient in-memory computation platform," in *Proc. 20th Int. Symp. Qual. Electron. Design (ISQED)*, Mar. 2019, pp. 52–57.

[20] *Predictive Technology Model*. Accessed: Sep. 14, 2019. [Online]. Available: http://ptm.asu.edu/

[21] G. Jan *et al.*, "Demonstration of fully functional 8Mb perpendicular STT-MRAM chips with sub-5ns writing for non-volatile embedded memories," in *Proc. Symp. VLSI Technol. (VLSI-Technol.)*, Jun. 2014, pp. 1–2.

[22] H. Maehara *et al.*, "Tunnel magnetoresistance above 170% and resistance–area product of 1 Ω $(\mu m)^2$ attained by in situ annealing of ultra-thin MgO tunnel barrier," *Appl. Phys. Express*, vol. 4, no. 3, 2011, Art. no. 033002.

[23] H. Noguchi *et al.*, "A 3.3 ns-access-time 71.2 μW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2015, pp. 136–138.

[24] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, 2012.

[25] *Everspin Technologies*. Accessed: Sep. 14, 2019. [Online]. Available: https://www.everspin.com/

[26] J. S. Kim *et al.*, "GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies," *BMC Genomics*, vol. 19, no. 2, p. 89, 2018.

[27] *1000 Genomes Project*. Accessed: Sep. 14, 2019. [Online]. Available: http://ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/

[28] *SRR1153470*. Accessed: Sep. 14, 2019. [Online]. Available: https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR1153470

[29] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization (WWC)*, Dec. 2001, pp. 3–14.

[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2007, pp. 13–24.

[31] *Cortex-A5 Processor*. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a5.php/

[32] D.-I. Jeon and K.-S. Chung, "CasHMC: A cycle-accurate simulator for hybrid memory cube," *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 10–13, Jan. 2017.

[33] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*. New York, NY, USA: ACM, 2017, pp. 273–287, doi: 10.1145/3123939.3124544.

[34] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. 53nd Annu. Design Autom. Conf. (DAC)*, 2016, p. 173.

[35] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, and D. Blaauw, "Compute caches," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 481–492.

[36] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 288–301.

[37] S. Jain, A. Ranjan, K. Roy, and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic RAM," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 3, pp. 470–483, Mar. 2018.

[38] *Micron tn-41-01: Calculating Memory System Power for DDR3*. Accessed: Sep. 14, 2019. [Online]. Available: https://www.micron.com/resource-details/3465e69a-3616-4a69-b24d-ae459b295aae

[39] C.-M. Liu *et al.*, "SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[40] J. Kim *et al.* (2017). *Genome Read In-Memory (GRIM) Filter: Fast Location Filtering in DNA Read Mapping Using Emerging Memory Technologies*. [Online]. Available: https://people.inf.ethz.ch/omutlu/pub/GRIM-genome-read-in-memoryfilter_psb17-poster.pdf