DART-PIM: DNA read mApping acceleRaTor Using Processing-In-Memory

Appendix A - MAGIC-based Operations

Abstract

DART-PIM [1], a DNA sequence Alignment acceleRaTor using Processing-In-Memory, is a comprehensive framework for accelerating the entire read-mapping process. DART-PIM uses emerging memory technologies to execute **all read-mapping steps inside the memory units**, thus circumventing the data transfer bottleneck. This document for DART-PIM includes complementary explanations about the MAGIC NOR logic method in Section I, and details the logic and arithmetic operations that were developed and optimized for DART-PIM in Section II.

I. MAGIC - Executing Logic Operations with Memristors

Another unique property of the memristive memories is the capability of performing computations within the memory cells themselves [2]–[6]. In DART-PIM, the computations are based on the MAGIC NOR operations [3] due to their advantages, but the proposed PIM-based accelerator is not limited to this technique, and any other PIM method can be adopted. MAGIC NOR offers several benefits compared to other stateful logic methods, including the requirement of a single execution voltage (V_g), the separation between input and output memristors, and the absence of additional peripheral elements [7].

The execution of a MAGIC NOR gate involves applying voltages to the inputs and output memristors. The state of the output memory cell changes in accordance with the logical states of the input memristors. As a single-input NOR gate is essentially a NOT gate; hence, both NOR and NOT operations can be executed using MAGIC. The MAGIC NOR and NOT gates operation requires a two-clock cycle process:

1) A logical '1' is written to the output memristor by applying a voltage (called V_{w1}) to it.
2) V_g is applied to the column(s) of the input memristor(s), the output memristor's column is connected to ground.

The unique structure of the MAGIC NOR gate enables the parallel execution of multiple gate instances within the memristive memory. Each gate instance is placed in a different row, known as the MAGIC row operation, or a different column, known as the MAGIC column operation. To support both MAGIC row and MAGIC column operations, a transpose memory architecture is required [7]. It's important to note that while the inputs and output of a single gate can be located in non-adjacent cells, alignment of inputs and outputs is necessary for all gate instances. An illustration of N aligned MAGIC NOR gates executed in parallel is presented in Figure 1.

The NOR gate is a functionally complete gate, its operation spans the entire set of Boolean operations, making the MAGIC NOR gate capable of executing any desired logic function. This property allows MAGIC NOR to serve as the fundamental computing element for various processing tasks within the memory by breaking down the desired function into a sequence of MAGIC NOR operations.

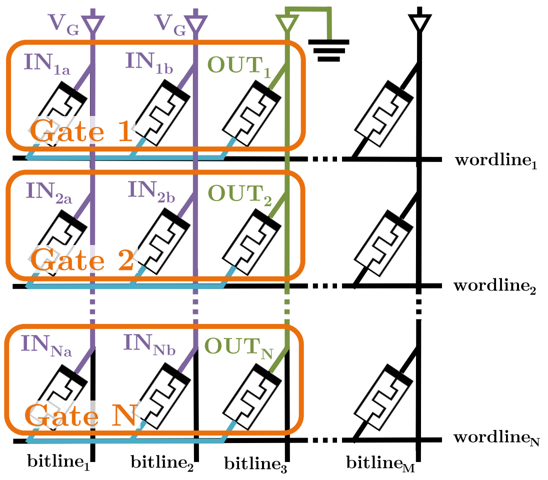

Fig. 1. An NxM memristive crossbar with in-memory parallel execution of N aligned MAGIC NOR gates. Each gate is located in a different row and uses three cells: two for the inputs and one for the output.

TABLE I
SUMMARY OF EXECUTION CYCLES FOR MAGIC-NOR-BASED OPERATIONS WITH N-BIT OPERANDS (UNLESS SPECIFIED OTHERWISE).

Logical Operation	**# Cycles**
AND	$3N$
XNOR	$4N$
XOR	$5N$
Copy	$1+N$
Addition of two N-bit in-memory numbers	$9N$
Addition of an N-bit and a single-bit in-memory numbers	$5N$
Addition of one in-memory number and a constant	$5N$
Subtraction of two in-memory numbers	$9N$
Mux between two in-memory numbers	$3N+1$
Minimum of two in-memory numbers	$12N+1$

All gates in the sequence of MAGIC NOR operations for executing logical or arithmetic operations are executed serially. As a result, reducing the number of gates in the sequence leads to decreased computation latency. However, determining the most efficient sequence of MAGIC NOR operations for in-memory computations poses a challenge. To address this, a tool called SIMPLER [8] has been utilized to identify the optimal NOR-based sequences for the required computations. A detailed description of these sequences can be found in Appendix A, and they are summarized in Table I.

II. MAGIC-BASED ARITHMETIC OPERATIONS

All gates in the sequence of MAGIC NOR operations for execution of a logical or arithmetic operations are executed serially. Therefore, as the number of gates is smaller, the latency of the computation decreases. However, choosing the smallest sequence of MAGIC NOR operations for efficient in-memory computations is challenging. Using a previously developed tool called SIMPLER [8], the best NOR-based sequences for the necessary computations were chosen. The sequences are described in details in Appendix A, and are summarised in Table I. For the description of the executions, two N-bit numbers $A = a_1, a_2, ..., a_N, B = b_1, b_2, ..., b_N$ which are stored in the same row in the memristive memory are used.

A. *Bitwise AND, XOR and XNOR of Two In-memory Numbers*

All bitwise operations between A and B are serially executed using MAGIC NOR. Hence, each logic function is done N times, once for each pair of bits $a_i, b_i, \forall i \in [1, N]$.
The bitwise logic functions that SPIM executes are AND, XOR, XNOR. Their minimized execution with NOR and NOT gates for a pair of bits is as follows:

$$AND(a_i, b_i) = NOR\big(NOT(a_i), NOT(b_i)\big) \tag{1}$$

$$XOR(a_i, b_i) = NOR\big(AND(a_i, b_i), NOR(a_i, b_i)\big) \tag{2}$$

$$\begin{aligned} &XNOR(a_i, b_i) = \\ &NOR\Big(NOR\big(a_i, NOR(a_i, b_i)\big), \\ &\quad NOR\big(b_i, NOR(a_i, b_i)\big)\Big) \end{aligned} \tag{3}$$

Therefore, the number of NOR/NOT gates are three, four and five for respectively AND, XNOR and XOR logic functions for a single bit (XNOR has four and not five gates since the result of the same gate $NOR(a_i, b_i)$ is used twice).

B. *Addition of Two In-memory Numbers*

In-memory addition of A and B is done by executing N single-bit Full Adder ($1bitFA$) serially. Hence, each 1bit FA is done N times, once for each pair of bits a_i, b_i along with a carry-in bit Cin_i, $\forall i \in [1, N]$.
The minimal implementation of a 1bit FA with NOR and NOT gates has nine gates:

$$\begin{gathered}\textbf{Assume:}\\ X = NOR(a_i, b_i)\\ Y = NOR(NOR(a_i, X), NOR(b_i, X))\\ Z = NOR(Cin_i, Y)\\ \textbf{Then:}\\ Cout_i(a_i, b_i, Cin_i) = NOR(X, Z)\\ S_i(a_i, b_i, Cin_i) = NOR(NOR(Y, Z), NOR(Z, Cin_i))\end{gathered} \tag{4}$$

Therefore, the total number of clock cycles is $9N$.

When one of the numbers is a single-bit number and the other is an N-bits number, the addition can be degenerated to a single-bit half adder (1bit HA), where the carry-out bit of each HA inputs one of the inputs of the next HA. In this case, the minimal implementation of a 1bit HA has five NOR gates:

$$\begin{gathered}Cout_i(a_i, b_i) = NOR(NOT(a_i), NOT(b_i))\\ S_i(a_i, b_i) = NOR(NOR(a_i, b_i), Cout_i))\end{gathered} \tag{5}$$

Therefore, the total number of clock cycles is $5N$.

C. Addition of One In-memory Number with a Constant Known to the Controller

When adding A which is stored in the same row in the memory with a known N-bits constant $K = k_1, k_2, ..., k_N$, the trivial way is to use a single-bit full adder, same as when both numbers are in the memory. To reduce the number of steps, a technique offered in [9] is adopted.

Since the constant is known to the controller of the memory, the logic function that is performed over each bit a_i of the number, depends on the value of the bit of the constant k_i, $\forall i \in [1, N]$:

- When $k_i ='0'$:
  $S_i(a_i, Cin_i) = XOR(a_i, Cin_i)$,
  $Cout_i = AND(a_i, Cin_i)$
  As described in Subsection II-A, $AND(a_i, Cin_i)$ is contained in $XOR(a_i, Cin_i)$; therefore five NOR gates are needed.
- When $k_i ='1'$:
  $S_i(a_i, Cin_i) = XNOR(a_i, Cin_i)$,
  $Cout_i = OR(a_i, Cin_i)$
  As described in Subsection II-A, $NOR(a_i, Cin_i)$ is contained in $XNOR(a_i, Cin_i)$. By adding a single NOT gate, an OR operation is received: $OR(a_i, Cin_i) = NOT(NOR(a_i, Cin_i))$. Therefore, again five NOR gates are necessary.

It can be concluded that independent of the value of k_i, the single-bit full adder takes five cycles. Therefore, adding an in-memory number with a constant known to the controller takes $5N$.

D. Subtraction of Two In-memory Numbers

In-memory subtraction of A and B is done by executing N single-bit Full Subtractor ($1bit\ FS$) serially. Hence, each 1bit FS is done N times, once for each pair of bits a_i, b_i along with a borrow-in bit Bin_i, $\forall i \in [1, N]$.

The minimal implementation of a 1bit FS with NOR and NOT gates has nine gates:

$$\begin{gathered}\textbf{Assume:}\\ X = NOR(a_i, b_i)\\ Y_1 = NOR(a_i, X), Y_2 = NOR(b_i, X)\\ Z = NOR(Y_1, Y_2), W = NOR(Z, Bin_i)\\ V = NOR(W, Bin_i)\\ \textbf{Then:}\\ Bout_i(a_i, b_i, Bin_i) = NOR(V, Y_2)\\ D_i(a_i, b_i, Cin_i) = NOR(NOR(W, Z), V)\end{gathered} \tag{6}$$

Therefore, the total number of clock cycles is $9N$.

E. Maximum between Two In-memory Numbers

Finding the maximum between A and B is done by first subtracting the numbers, and second choosing the subtrahend (A) if the result was positive and the minuend (B) if the result was negative using a multiplexer (MUX). The subtraction is described in II-D, and the MUX is executed using a single-bit MUX for each bit, where the select (sel) input is the most significant bit (MSB) of the subtraction result (A and B are represented in two's complement notation; therefore, the MSB is the sign bit, so if $MSB =' 0' \Rightarrow A \geq B$ and if $MSB =' 1' \Rightarrow A < B$). An efficient implementation of such a single-bit MUX using 4 NOR gates is:

$$\begin{aligned} &MUX(a_i, b_i) = \\ &\quad NOR\Big(NOR\big(NOT(sel_i), a_i\big), NOR(sel_i, b_i)\Big) \end{aligned} \tag{7}$$

Therefore, the total number of clock cycles is $4N$.
As described in II-D, the subtraction takes $9N$ cycles, so the total number of cycles for finding the maximum is $(9+4)N = 13N$.

F. Maximum with a Zero

When finding the maximum between A and a zero, the result is A if its MSB (the sign bit) is zero, and a zero otherwise. The most efficient execution is done by performing a NOT operation to A's MSB (NOT_MSB_A) and then performing a bitwise AND operation between it and all of A's bits. Therefore, if A is positive, $NOT\_MSB\_A =' 1'$, so the result is A, and if A is negative, $NOT\_MSB\_A =' 0'$, so the result is a zero.

A NOT operation of the MSB takes a single clock cycle, and the bitwise AND operation takes $3N$ cycles, as described in II-A. Therefore, the total number of cycles for finding the maximum with a zero is $3N + 1$.

References

[1] Anonymous, "DART-PIM: DNA read mApping acceleRaTor Using Processing-In-Memory," in Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (submitted), 2024.

[2] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "'Memristive' Switches Enable 'Stateful' Logic Operations via Material Implication," Nature, vol. 464, no. 7290, pp. 873–876, Apr. 2010.

[3] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC - Memristor-Aided Logic," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 61, no. 11, pp. 895–899, nov 2014.

[4] S. Gupta, M. Imani, and T. Rosing, "FELIX: Fast and Energy-Efficient Logic in Memory," in 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018, pp. 1–7.

[5] B. Hoffer, V. Rana, S. Menzel, R. Waser, and S. Kvatinsky, "Experimental Demonstration of Memristor-Aided Logic (MAGIC) Using Valence Change Memory (VCM)," IEEE transactions on electron devices, vol. 67, no. 8, pp. 3115 – 3122, 2020. [Online]. Available: https://juser.fz-juelich.de/record/885492

[6] N. Peled, R. Ben-Hur, R. Ronen, and S. Kvatinsky, "X-magic: Enhancing pim using input overwriting capabilities," 10 2020, pp. 64–69.

[7] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," IEEE Transactions on Nanotechnology, vol. 15, no. 4, pp. 635–650, July 2016.

[8] R. Ben-Hur, R. Ronen, A. Haj-Ali, D. Bhattacharjee, A. Eliahu, N. Peled, and S. Kvatinsky, "SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 10, pp. 2434–2447, 2020.

[9] B. Perach, R. Ronen, B. Kimelfeld, and S. Kvatinsky, "PIMDB: Understanding Bulk-Bitwise Processing In-Memory Through Database Analytics," 2022.