

HashPIM: High-Throughput SHA-3 via Memristive Digital Processing-in-Memory

Batel Oved, Orian Leitersdorf, Ronny Ronen, and Shahar Kvatinsky

Andrew and Erna Viterbi Faculty of Electrical and Computer Engineering, Technion – Israel Institute of Technology, Israel

batelov@campus.technion.ac.il, orianl@campus.technion.ac.il, ronny.ronen@technion.ac.il, shahar@ee.technion.ac.il

Abstract—Recent research has sought to accelerate cryptographic hash functions as they are at the core of modern cryptography. Traditional designs, however, suffer from the von Neumann bottleneck that originates from the separation of processing and memory units. An emerging solution to overcome this bottleneck is processing-in-memory (PIM): performing logic within the same devices responsible for memory to eliminate data-transfer and simultaneously provide massive computational parallelism. In this paper, we seek to vastly accelerate the state-of-the-art SHA-3 cryptographic function using the memristive memory processing unit (mMPU), a general-purpose memristive PIM architecture. To that end, we propose a novel in-memory algorithm for variable rotation, and utilize an efficient mapping of the SHA-3 state vector for memristive crossbar arrays to efficiently exploit PIM parallelism. We demonstrate a massive energy efficiency of 1,422 Gbps/W, improving a state-of-the-art memristive SHA-3 accelerator (SHINE-2) by $4.6\times$.

Index Terms—Cryptography, SHA-3, processing-in-memory (PIM), stateful logic, memristor.

I. INTRODUCTION

As we enter the era of data-intensive computing across many Internet of Things (IoT) devices, cryptography is emerging as a crucial field for secure communication. At the core of this field are cryptographic hash functions [1] which are fundamental for tasks such as digital signature generation and verification, key derivation, and pseudo-random bit generation [1], [2]. These functions generalize traditional hashing with additional properties aimed at improving security, such as being very infeasible to invert. An emerging state-of-the-art hash function is Secure Hash Algorithm-3 (SHA-3), which exploits techniques such as sponge construction to enhance security [2].

While hashing is traditionally implemented via software, hardware accelerators are emerging to provide unparalleled performance using ASICs [3]–[6], FPGAs [3], [7], and memristive memories (e.g., ReRAM) [8]–[10]. Hardware accelerators benefit from the inherent flexibility for bit-wise accesses that does not exist in CPUs and GPUs. Yet, processing units, including hardware accelerators, are subject to the *memory wall*; therefore, when hashing large objects stored in memory (or disk), data transfer becomes the bottleneck [8].

An emerging concept to overcome the memory wall is that of Processing-in-Memory (PIM). The fundamental idea of PIM is to shift the computation into the memory, thereby avoiding the data-transfer between the CPU and memory. Recent techniques for PIM involve using the same physical devices for both *binary* storage and basic *digital* logic gates, via technologies such as digital memristive memories [11],

[12], DRAM [13], SRAM [14], and FeFET [15]. Previous works have sought to utilize this emerging field to accelerate a wide range of applications, including the SHA-3 cryptographic hash function [8]–[10]. Unfortunately, previous SHA-3 designs require specific complex *near-array* periphery that leads to costly data conversion which diminishes the benefit of PIM. Conversely, we seek to design an *in-array algorithm* that utilizes the emerging general-purpose memristive Memory Processing Unit (mMPU) [16] *without any custom circuitry*.

We focus on a digital memristive PIM architecture [11], [12], [16] as this technology has vast potential for large-scale efficient PIM; regardless, the proposed algorithms can be generalized to additional PIM techniques and technologies. Memristors [17] are similar to resistors as they are two-terminal devices, yet they possess a highly unique property: an applied voltage can alter their internal resistance. Therefore, memristors can inherently support storage by representing binary information via their resistance. Interestingly, recent works [18]–[20] have shown that memristors also support basic logic functionality through *stateful logic* [16], [21]. Thus, memristors inherently enable both memory and digital logic.

In this paper, we propose an efficient *in-memory* algorithm for SHA-3, using the mMPU [16], that efficiently exploits the vast potential of PIM. We compare our design to other accelerators and demonstrate superior energy efficiency.

II. BACKGROUND

A. Secure Hash Algorithm-3 (SHA-3)

Hash algorithms reduce a large variable-sized input to a small fixed-sized output (hash value) while maintaining a near-uniform output distribution. Cryptography, the field in computer science establishing secure communication, extends the notion of hash functions to *cryptographic hash functions* [1] that also possess certain properties which enhance security. For example, these functions must be infeasible to invert and slight modifications to the input drastically change the hash value. The Keccak family of cryptographic hash functions was proposed by Bertoni *et al.* [22], and was later adopted as part of the state-of-the-art SHA-3 standard [2]. This standard proposes four variations for different output lengths; without loss of generality, we discuss SHA3-256 in this paper.

We describe the overall operation of SHA-3, as shown in Fig. 1(a). The input is a message of size m bits which is padded to be of a length that is a multiple of r and then split into t blocks of size r each (e.g., $r = 1088$). The algorithm

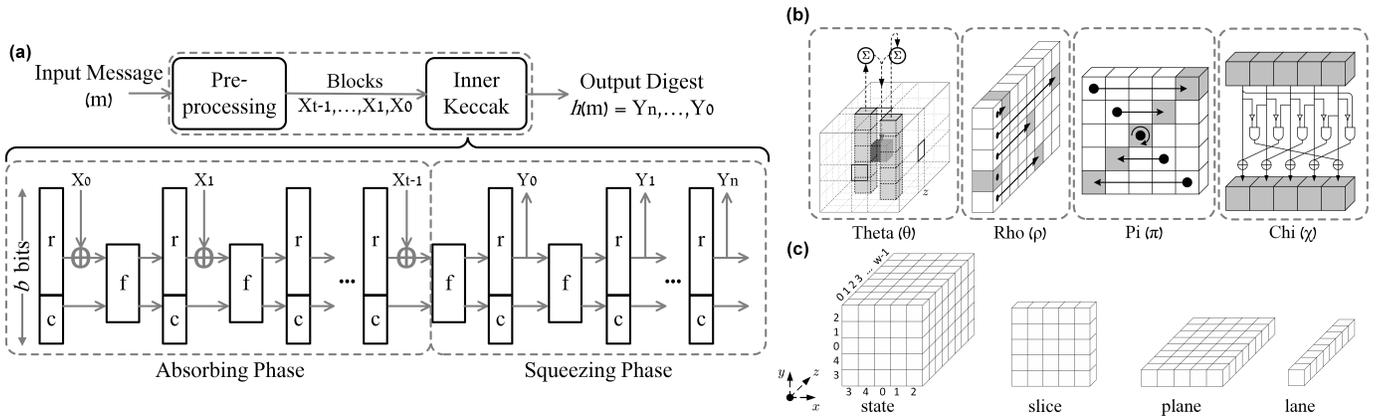


Fig. 1. (a) Overview of the SHA-3 algorithm. (b) The Keccak-f round function [22]. (c) The notation for the state vector and its subvectors [22].

Algorithm 1 Keccak-f

Input: State array $A[x][y]$ (for all $x, y \in [0, 4]$)

Output: State array $A[x][y]$ (for all $x, y \in [0, 4]$)

1: **for** $i_r = 0, \dots, 23$ **do**

Theta (θ) step:

2: $C[x] \leftarrow A[x][0] \oplus \dots \oplus A[x][4] \quad \forall x \in [0, 4]$

3: $D[x] \leftarrow C[x-1] \oplus (C[x+1] \lll^a 1) \quad \forall x \in [0, 4]$

4: $A[x][y] \leftarrow A[x][y] \oplus D[x] \quad \forall x, y \in [0, 4]$

Rho (ρ) step:

5: $A[x][y] \leftarrow A[x][y] \lll^b r[x][y]^b \quad \forall x, y \in [0, 4]$

Pi (π) step:

6: $A[y][2x+3y] \leftarrow A[x][y] \quad \forall x, y \in [0, 4]$

Chi (χ) step:

7: $A[x][y] \leftarrow A[x][y] \oplus (\overline{A[x+1][y]} \wedge A[x+2][y]) \quad \forall x, y$

Iota (ι) step:

8: $A[0][0] \leftarrow A[0][0] \oplus RC[i_r]^c$

9: **end for**

^aThe operator \lll denotes a rotation (cyclic shift),

^b $r[x][y]$ denotes the constants that define the rotation amount [2],

^c $RC[i_r]$ denotes the constants for each round [2].

$C[x]$, $D[x]$ are intermediates, and indexing is modulo 5.

proceeds with the absorbing phase that initializes an internal state of $b \triangleq r + c$ bits to zero and then iteratively performs an exclusive-or (XOR) with an input block followed by the Keccak-f routine. The routine considers the state vector as a $5 \times 5 \times 64$ binary tensor and applies the operations detailed in Algorithm 1 and Fig. 1(b). The state vector after the final iteration is truncated to receive the overall hash¹.

B. Memristive Digital Processing-in-Memory (PIM)

Memristors [17] are rapidly emerging as novel physical devices that inherently support both storage and logic functionalities. Memristors are similar to resistors in that they are two-terminal resistive devices, yet they also possess a unique property: a sufficiently-high current can modify their internal resistance. Therefore, memristors can be utilized for binary information storage (e.g., designating high resistance as logical

¹Extendable-output-functions continue with additional iterations of Keccak-f as part of the squeezing phase, to generate hash values longer than the state array. The proposed algorithm can also be generalized to such functions.

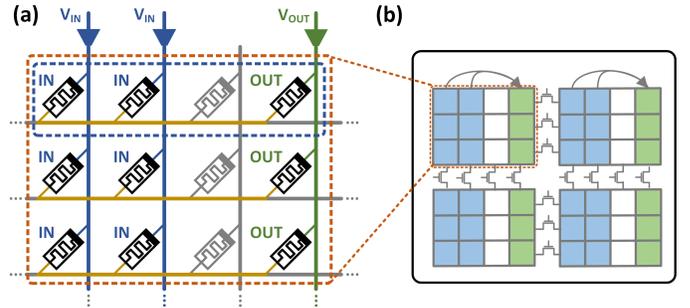


Fig. 2. (a) Memristive stateful logic – a single cycle operation across multiple rows simultaneously. (b) Switches creating four independent partitions.

0 and low resistance as logical 1) as data is written with a relatively high current and read using a relatively low voltage (measuring the current and deriving the resistance). Memristors are typically connected in crossbar array structures, as shown in Fig. 2(a). Furthermore, memristors inherently support *stateful logic* [16] in the resistive domain which sets the resistance of an output memristor conditional on the states of the input memristors (e.g., performing NOR) [18]–[20].

The dual functionality of memristors can be exploited towards the memristive Memory Processing Unit (mMPU): a general-purpose memory with massive computational parallelism for bitwise operations, originating from three forms:

- *Row/column parallelism:* By applying voltages on bitlines/wordlines of crossbars, we find that several gates can be performed in parallel within the array itself (when their columns/rows are aligned across different rows/columns) [11], [12], as shown in Fig. 2(a).
- *Partition parallelism:* An overall crossbar may be split into multiple smaller partitions, using transistor switches that divide the bitlines/wordlines, to enable further parallelism [20], [23], [24], as shown in Fig. 2(b). This enables parallel gates within the same row/column, in addition to the parallelism across multiple rows/columns.
- *Crossbar parallelism:* The overall mMPU consists of many crossbar arrays that may operate in parallel [25].

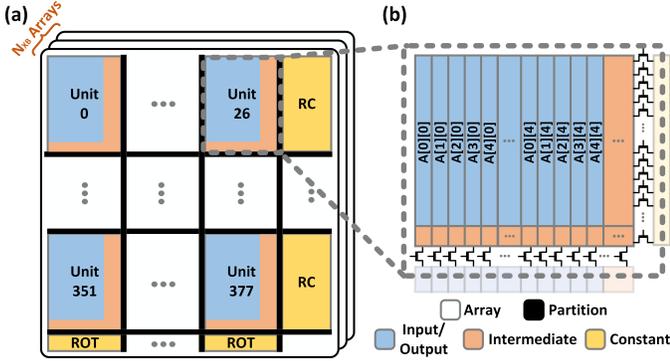


Fig. 3. Overview of the proposed architecture for a (a) single SHA-3 crossbar array that holds multiple SHA-3 units. (b) A single SHA-3 unit.

III. HASHPIM ARCHITECTURE

We propose to exploit the vast potential of the mMPU towards an efficient, scalable, and high-throughput SHA-3 algorithm. The benefit of the mMPU over alternative techniques is both due to the ability to efficiently compute precisely where the state vector is stored, and the high efficiency of processing within an array rather than near-array via peripheral circuits.

We detail the proposed architecture within a single crossbar array as the same operations may be performed in parallel across multiple crossbar arrays. Consider a crossbar array as shown in Fig. 3(a), with size of 1024×1024 bits, divided into 378 partitions (27 horizontally, 14 vertically). Each partition (group of 72×37 memristors) is designated a *SHA-3 unit* and is assigned to compute the SHA-3 hash for a specific message. Thus, the 378 units enable the parallel computation of SHA-3 on 378 different messages within the same crossbar array.

Each SHA-3 unit stores the state-vector corresponding to that message (throughout the computation) and is responsible for applying the *Keccak-f* function on the state-vector. We choose to map the $5 \times 5 \times 64$ state-vector onto 25×64 memristors (similar to [9]) as an analysis of the routines in the *Keccak-f* function revealed that this mapping supports optimal parallelism for the θ, π, χ and ι steps. Regarding the ρ step, previous works have been unable to implement this step within the array due to the different rotation amount provided for each *lane* (see Fig. 1(c)), and have thus instead required to read/write the state-arrays and perform the computation in the periphery. The drawback of such a solution is that the periphery becomes the bottleneck when scaling (e.g., 378 SHA-3 units). Conversely, by extending a recent concept proposed for floating-point operations [24], we demonstrate an efficient in-array design which circumvents such periphery.

In this section, we detail the proposed design within each SHA-3 unit for the various steps that comprise Algorithm 1.

A. Theta (θ) Step

We start by reducing the state-vector (see Fig. 1(c)) across the y dimension using exclusive-or (XOR) operations; that is, we compute the XOR of every five columns from the 25×64 state-vector to result in a 5×64 vector representing

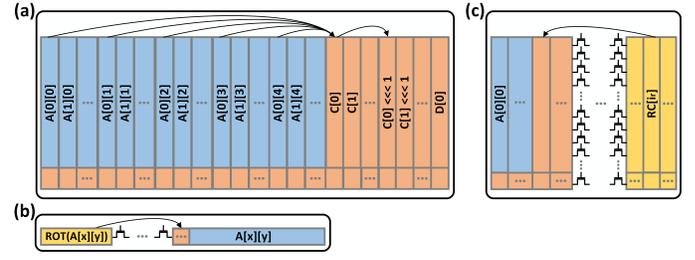


Fig. 4. Overview of the operation of the (a) Theta, (b) Rho, and (c) Iota steps. The Pi and Chi steps require only row operations and are thus not drawn.

Algorithm 2 Variable Rotation

Input: N_x -bit x_i , N_t -bit t_i , for every column i .

Output: N_x -bit output $z_i = x_i \lll t_i$ for every column i .

- 1: $\forall i : z_i \leftarrow x_i$
- 2: **for** $j = 0, \dots, \log_2(N_x)$ **do**
- 3: $\forall i : z_i \leftarrow \text{mux}_{t_i}(z_i \lll 2^j, z_i)$
- 4: **end for**
- 5: **return** z_i for every column i .

$C[0], \dots, C[4]$ (see Algorithm 1) that is stored in the intermediate area (see Fig. 4(a)). We compute $D[0], \dots, D[4]$ by copying $C[0], \dots, C[4]$ to 5 additional intermediate columns (in-row gates), shifting all of the columns once (in-column gates), and then computing the XOR with $C[0], \dots, C[4]$. Lastly, the original state-vector is updated by computing the XOR of all columns with the relevant lanes from $D[0], \dots, D[4]$ (e.g., $D[0]$ is XORed with the first five columns in the state-vector).

B. Rho (ρ) Step

At the core of this step is a variable rotation: for each lane in the state vector, we need to rotate that lane by a given number of bits (determined according to $r[x][y]$ see [2]). The difficulty in this *variable* rotation is that, unlike the *constant* shift in θ , every lane may contain a different rotation amount [13]. This is seemingly an inherent contradiction to the operation of the mMPU as the operation involves data-dependent control flow.

A similar task of in-array *variable shifting* was recently considered for in-memory floating-point operations [24]. Essentially, the variable shift is represented as a sequence of multiplexer operations that are chosen according to a logarithmic-shifter [26] design, thereby converting the control-flow to data-flow. We extend this algorithm to the task of variable rotation (cyclic shifting) in Algorithm 2. To remain in-place (i.e., not require a copy of the state-vector), we analyze the cyclic dependencies in the rotation and execute them serially with a single slice of redundancy for the storage of the state array.

We store the constant shift amounts ($r[x][y]$) at the bottom of the crossbar array (ROT in Fig. 3(a)), shared across all of the units that are aligned vertically (to improve memory utilization for a negligible latency increase). Therefore, as shown in Fig. 4(b), we find that each iteration of Algorithm 2 begins by retrieving the relevant bit for the shift amount from the shared ROT for that vertically-aligned set of units.

C. Pi (π) Step

The step involves reordering the lanes, whereas, at each time, one lane, $A[x][y]$ is copied to the intermediate area along with the target lane $A[y][2x + 3y]$ and then the original lane, $A[x][y]$, is copied to the target address. This process is then repeated for $A[y][2x + 3y]$ until all of the lanes in the state-array have moved (except for $A[0][0]$ that remains in-place).

D. Chi (χ) Step

We convert the expression in Algorithm 1 to a sequence of NOT, NOR, XOR, and COPY operations (using De Morgan's law). Each *plane* (see Fig. 1(c)) holds five dependent lanes; hence, all lanes of that plane are inverted and stored at the intermediate area (later to perform the NOR operation with the adequate neighbor lane, see Algorithm 1). Lastly, each lane of that plane is XORed with the NOR result of the neighbor lanes, and is stored back by a COPY operation.

E. Iota (ι) Step

In this step, an XOR operation is performed on lane $A[0][0]$, after creating a local copy of the round constant [2], $RC[i_r]$: $i_r \in [0, n_r - 1]$ from the RC block (see Fig. 4(c)), to each SHA-3 unit of that row. The new value of $A[0][0]$ is mapped to the original location as the input $A[0][0]$.

IV. EVALUATION

The HashPIM architecture and algorithm are evaluated on a 1024×1024 crossbar array which contains $U_{XB} \triangleq 378$ independent SHA-3 units that operate in parallel (each of size 72×37). The results are verified by a cycle-accurate simulator [27] that logically models the crossbar array (with partitions) and provides an interface for in-memory operations, while recording latency, energy and area usage. The results are extended to N_{XB} crossbars. Overall, one SHA-3 round requires 3,494 cycles ($Latency_{Round}$) and consumes $0.765nJ$ ($Energy_{Unit}$), assuming $r = 1088$ and MAGIC [19] gate parameters of $3ns$ delay ($333MHz$) and $6.4fJ$ energy [12]. The expressions for throughput and power are:

$$T_{put_{Unit}} = \frac{r}{Latency_{Round}} * f, \quad (1)$$

$$T_{put_{System}} = T_{put_{Unit}} * U_{XB} * N_{XB}, \quad (2)$$

$$Power_{System} = \frac{T_{put_{System}} * Energy_{Unit}}{r}. \quad (3)$$

Table I summarizes our results and compares to other SHA-3 accelerators (both CMOS-based and memristor-based). The code repository [27] includes additional results as well as an explanation for the methodology that led to these results.

V. CONCLUSION

This paper demonstrates the vast potential of the mMPU for energy-efficient cryptographic hash algorithms through a case study with the SHA-3 function. We propose a novel variable rotation algorithm and an efficient mapping that exploits the inherent parallelism of the mMPU towards high-throughput SHA-3 execution. This provides a massive throughput per watt of 1,422 Gbps/W, improving the state-of-the-art by $4.6\times$.

TABLE I
PERFORMANCE COMPARISON OF SHA-3 HARDWARE DESIGNS

Work	f (MHz)	Tput (Gbps)	Tput/W (Gbps/W)	Tput/Area (bps/ F^2)
65nm ASIC [5]	1K	48	-	7,619
SHINE-1 [8]	2K	33.4	263	21,916
SHINE-2 [8]	2K	54	311	22,227
HashPIM (1 XB)	333	39.2	1,422	9,354
HashPIM (2 XB)		78.4		

ACKNOWLEDGMENT

This work was supported by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259.

REFERENCES

- [1] A. Menezes, P. V. Oorschot, and S. Vanstone, "Handbook of applied cryptography (1st edition)," *CRC Press*, 1997.
- [2] M. J. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions," *NIST*, 2015.
- [3] A. Akin *et al.*, "Efficient hardware implementations of high throughput SHA-3 candidates Keccak Luffa and Blue Midnight Wish for single-and multi-message hashing," in *SIGCONF*, 2010.
- [4] P. Pessl and M. Hutter, "Pushing the limits of SHA-3 hardware implementations to fit on RFID," *CHES*, 2013.
- [5] M. M. Wong *et al.*, "A new high throughput and area efficient SHA-3 implementation," *ISCAS*, 2018.
- [6] R. Ramanarayanan, "18Gbps 50mW reconfigurable multi-mode SHA hashing accelerator in 45nm CMOS," in *ESSCIRC*, 2010.
- [7] J. Strömbergsson, "Implementation of the Keccak hash function in FPGA devices," *Technical report, InformAsic AB*, 2008.
- [8] K. Nagarajan *et al.*, "SHINE: A novel SHA-3 implementation using ReRAM-based in-memory computing," *ISLPEd*, 2019.
- [9] D. Bhattacharjee, V. Pudi, and A. Chattopadhyay, "SHA-3 implementation using ReRAM based in-memory computing architecture," *18th International Symposium on Quality Electronic Design (ISQED)*, 2017.
- [10] C. Yang and Z. Chen, "A processing-in-memory implementation of SHA-3 using a voltage-gated spin hall-effect driven MTJ-based crossbar," in *GLSVLSI*, 2019.
- [11] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (MAGIC)," *TNANO*, 2016.
- [12] M. S. Q. Truong *et al.*, "RACER: Bit-pipelined processing using resistive memory," in *MICRO*, 2021.
- [13] L. Shuangchen *et al.*, "DRISA: A DRAM-based reconfigurable in-situ accelerator," *MICRO*, 2017.
- [14] S. Aga *et al.*, "Compute caches," *HPCA*, 2017.
- [15] D. Reis *et al.*, "Computing in memory with FeFETs," in *ISLPEd*, 2018.
- [16] S. Kvatinsky, "Making real memristive processing-in-memory faster and reliable," in *CNNA*, 2021.
- [17] L. Chua, "Memristor-the missing circuit element," *TCT*, 1971.
- [18] J. Borghetti *et al.*, "'Memristive' switches enable 'stateful' logic operations via material implication," *Nature*, 2010.
- [19] S. Kvatinsky *et al.*, "MAGIC—memristor-aided logic," *TCAS-II*, 2014.
- [20] S. Gupta *et al.*, "FELIX: Fast and energy-efficient logic in memory," in *ICCAD*, 2018.
- [21] J. Reuben *et al.*, "Memristive logic: A framework for evaluation and comparison," in *PATMOS*, 2017, pp. 1–8.
- [22] G. Bertoni, "Keccak," *Advances in Cryptology – EUROCRYPT*, 2013.
- [23] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "MatPIM: Accelerating matrix operations with memristive stateful logic," in *ISCAS*, 2022.
- [24] O. Leitersdorf *et al.*, "AritPIM: High-throughput in-memory arithmetic," in *arXiv*, 2022.
- [25] R. Ronen *et al.*, "The Bitlet Model: A parameterized analytical model to compare PIM and CPU systems," *JETC*, 2021.
- [26] B. Parhami, "Computer Arithmetic: Algorithms and hardware designs," *Oxford University Press*, vol. 2, 2010.
- [27] Available at <https://github.com/BatelOved/HashPIM>.