2021 International Symposium on Secure and Private Execution Environment Design (SEED) | 978-1-6654-2025-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/SEED51797.2021.00018

Seeds of SEED: H-CRAM: In-memory Homomorphic Search Accelerator using Spintronic Computational RAM

Hüsrev Cılasun, Salonik Resch, Zamshed I. Chowdhury, Masoud Zabihi,
Zhengyang Zhao, Thomas Peterson, Jian-Ping Wang, Sachin S. Sapatnekar, Ulya R. Karpuzcu
University of Minnesota, Twin Cities
{cilas001, resc0059, chowh005, zabih003, zhaox526, pete9290, jpwang, sachin, ukarpuzc}@umn.edu

***Abstract*—Growing use of cloud raises privacy concerns for search in sensitive databases such as bioinformatics, where homomorphic encryption can help through direct computation –hence search/pattern matching– on the encrypted data. The recently proposed homomorphic secure content-addressable memory (SCAM) exploits this principle. Data dependencies in SCAM, however, lead to a memory bottleneck which has been addressed by various near-memory computing solutions. In this paper we demonstrate a more efficient alternative based on the true in-memory computing substrate spintronic Computational RAM (CRAM). The resulting SCAM accelerator, H-CRAM, achieves significant speedup and energy reduction.**

I. Introduction

Homomorphic encryption allows direct computation on the encrypted data (i.e., the cypertext) alleviating the need for decryption before the computation takes place. The obvious benefit is privacy preservation. The cyphertext can be processed without knowing the actual content, which translates into more secure computation. At the same time, there is practically no need +for decryption and (re)encryption, before and after computation takes place, respectively. Decrypted data outside of its source always represents a vulnerability because an attack may expose not only the sensitive data but also the decryption key. Under homomorphic computing, even if data is compromised during computation (where data stays encrypted), no damage would be the case. This is of crucial importance in a cloud environment. Data privacy is important for emerging modern applications such as face recognition, and crucial for others such as genomic pattern search in bioinformatics databases.

The encryption cryptosystem can be either *Somewhat Homomorphic (SHE)*, supporting only a limited number of operations that can be performed homomorphically (where data corruption is inevitable if this limit is exceeded); or *Fully Homomorphic (FHE)*, which allows unlimited homomorphic operations (where any type of computation can be performed homomorphically). Homomorphic operations can be performed on cyphertexts of Boolean variables, enabling seamless transformation for any arbitrary application.

Homomorphic encryption dates back to 1978 [1]. Most of the subsequent work relies on existing encryption schemes, which, unfortunately, have not resulted in practically satisfying implementations until recently. A key performance bottleneck is the *bootstrapping* operation, which, interleaved with actual homomorphic computing steps, performs noise reduction on encrypted data to avoid potential data corruption via noise accumulation, as each homomorphic operation (in other words, each step of homomorphic computation) introduces a bounded amount of noise on the encrypted data. This type of noise tends to accumulate over multiple steps of computation, hence, if left untreated, can prevent correct decryption after sufficient number of homomorphic operations are executed. This also directly sets a limit on the number of homomorphic operations that can be performed on the encrypted data. Luckily, the noise bound is strictly deterministic and can be tuned to allow a desired number of homomorphic operations. Bootstrapping essentially "refreshes" the encrypted data to its post-encryption minimum noise state, preferably after each homomorphic operation, so that arbitrary-length computation can be made homomorphic [2].

FHEW homomorphic encryption scheme [3] introduced in 2015 proposed very fast bootstrapping of less than a second, followed by TFHE [4] in 2016 featuring less than a hundred milliseconds bootstrapping for the first time. TFHE is accelerated by several algorithmic improvements, enabling bootstrapping to be completed in ten milliseconds [5]–[7]. An overview of the current status of homomorphic encyrption is provided in [8]. Any computer program, in theory, can be made homomorphic. Libraries to automatize such transformations exist [8]. TFHE based homomorphic encryption is used as a privacy-preserving secure method for genome storage and query [9], for homomorphic deep neural networks [10], and for convolutional neural networks [11], respectively.

Although universal homomorphic computing is favorable for many applications, bootstrapping remains to be too costly at scale. Luckily, bitwise comparison (which dominates search applications) can be implemented homomorphically without requiring bootstrapping. This is enabled by non-bootstrapped homomorphism in [12], where a secure content-addressable memory, SCAM, is introduced as an extension to the FHEW cryptosystem. SCAM has sufficient computational margin such that any two consecutive gate operations can be performed without bootstrapping. The original SCAM is implemented as an ASIC, which is not feasible for large scale search considering prohibitive data transfer overheads. Homomorphic encryption essentially comes with a large memory footprint due to the blowup in encrypted data size[1]. More scalable SCAM implementations include a hybrid memory cube (HMC) based near-memory solution [14] and a FeFET based processing-in-memory design [15]. Due to the large memory overhead, less distance – both logically and physically – between the memory and the compute logic is desirable. While HMC combines the memory and the logic in the same system-on-chip, [15] further reduces the data transfer overhead by implementing the SCAM logic within customized sense amplifiers at the memory periphery. However, neither of these approaches are optimal as tighter coupling between logic and memory is still possible by using a compute substrate fusing logic and memory operations within the same array in a seamless fashion. This is the case for spintronic Computational RAM (CRAM) [16], which,

[1]To put this into perspective, a traditional (non-homomorphic) encryption scheme such as AES [13] can have 1:2 (AES IV) unencrypted-to-encrypted memory ratio, where one bit unencrypted data requires 6.9KB [12] memory for the same 128-bit security level under homomorphic encryption.

on top of regular memory operations, can perform universal Boolean logic operations between its memory cells within the memory array in parallel without any need to offload them to customized peripheral circuitry. In this paper, we introduce H-CRAM, a CRAM-based, highly scalable SCAM accelerator for homomorphic search.

On the other end of the spectrum, universal processing-in-memory solutions such as CryptoPIM [17] can significantly accelerate the Number Theoretical Transform (NTT) at the core of bootstrapping, however, bootstrapping would still be disproportionately costly for search applications at scale which feature relatively less complex computational operations. This renders resistive-memory based CryptoPIM infeasible for homomorphic search, although (similar to H-CRAM) CryptoPIM also features a seamless fusion of logic and memory. We should also note that, compared to the resistive and ferroelectric variants, spintronic (magnetic) RAM is superior in terms of endurance, which is important for practical implementations [18][2]. Large magnetic memory arrays, such as Everspin's 256MB STT-MRAM design [21] have already been commercialized. Moreover, CRAM using Spin-Hall Effect (SHE) (or Spin-Orbit Torque, SOT) cells is demonstrated to be 3× faster while consuming 4× less energy than Spin-Transfer-Torque (STT) based variants in benchmark applications such as 2D convolution and digit recognition [22]. Therefore H-CRAM deploys SOT cells [23]. Putting it all together:

- We introduce H-CRAM, an efficient CRAM-based homomorphic search accelerator, which operates ≈ 2× faster than fastest known alternative.
- H-CRAM accelerates computation using an optimized long adder tree tailored to homomorphic search.
- H-CRAM deploys De Bruijn topology to minimize data transfers and to maximize memory reuse, in order to optimize the overall memory and energy consumption.
- H-CRAM features specialized control arrays for tighter coupling and further homogeneity.

The paper is organized as follows: Section II covers basics of CRAM and SCAM. Section III introduces our homomorphic search accelerator H-CRAM. Section IV provides quantitative characterization and Section V concludes the paper.

II. Background

A. Computational RAM (CRAM)

CRAM dates back to [16], where standard Spin-Torque-Transfer (STT) Magnetic RAM (MRAM) is slightly modified to enable logic operations within the memory array. By separating read and write paths, and thereby enabling independent optimization thereof (which would otherwise induce conflicting optimization targets), Spin-Hall Effect (SHE) or Spin-Orbit Torque (SOT) based CRAM significantly improves the energy and the operation speed of the basic STT based design [23].

Each SHE-CRAM cell comprises a magnetic tunneling junction (MTJ) and two access transistors. The MTJ consists of a fixed (polarity) magnetic layer, an insulating layer, a free (variable polarity) magnetic layer, and a SHE channel. In P (AP) state where the polarity of the free layer matches (does not match) the polarity of fixed layer, the MTJ has low (high) resistance, which encodes logic 0 (1).

Fig.1 shows the CRAM cell structure. Each cell is controlled using several driver lines –Wordlines for Read/Write (WLR/W)

[2]Endurance is typically captured by the switching activity. Although SOT-MRAM is considered to have a practically infinite endurance [19], earlier MTJ implementations report more than 10e+20 cycles [18], [20].

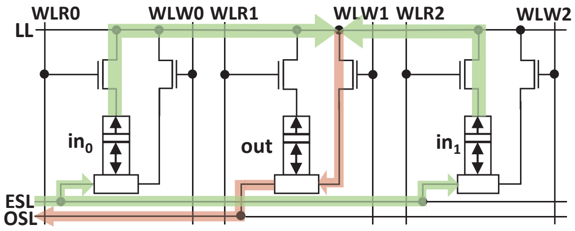


Fig. 1. Logic gate formation in SHE-CRAM. Currents passing through input cells in_0 and in_1 (highlighted in green) get combined (highlighted in red) and pass through the output cell (out). out is preset to a known value, which is fixed for each gate type. Input currents can be adjusted using a gate-specific bias voltage, such that out only switches for specific input combinations according to the respective truth table. For a given bias voltage, input currents evolve as a function of the resistance (state) of the input cells.

and Even/Odd Bit Select Lines (E/OSL)– and two access transistors. CRAM can be used in memory mode or logic mode. Fig.1 illustrates logic gate formation. Logic lines (LL) are used to connect cells (along a column) to act as inputs and outputs to logic gates. If inputs reside in even columns, the output should be in an odd column and vice versa. By driving WLR high (low) and WLW low (high) in the inputs (output cell), and enforcing a logic gate specific voltage difference on bit select lines, the combined current from the input MTJs passes through the output MTJ. The output MTJ is preset to a logic gate specific, known value –which would be AP/High-resistance/logic 1 for a NAND gate, e.g. In this case, the logic gate specific voltage difference along with this preset value guarantee that the output MTJ switches according to the truth table of the respective logic gate. In the example of the NAND gate, the output MTJ switches state (to P/Low-Resistance/ logic 0) only if the combined current through the output MTJ is high enough (i.e., both input MTJs are in low resistance P state). For all other input MTJ state combinations (AP-P,P-AP,AP-AP) the output MTJ does not switch (remains in the AP state). Since NAND gate is universal, CRAM can perform any computation. NAND, however, is not the only type of gate that CRAM supports: a large variety of elementary gates such as AND, OR, NOR, and 3/5-input majority (MAJ3/5) can be implemented using the same principle. It is also important to note that CRAM features two types of parallelism: finer-grain column level and coarser-grain array level. Column level parallelism enables the same Boolean gate operation to be performed in all columns of a single CRAM array in parallel, while array level parallelism allows multiple arrays to perform the very same computation simultaneously.

B. SCAM

Although general purpose homomorphic encryption to accommodate arbitrary types of computation can be more desirable for flexibility, it requires bootstrapping, which consists of many polynomial multiplications and therefore is too costly for even the simplest Boolean logic gate operations [14]. Bootstrapping is inevitable when the computational depth is high or the computation is not homogenous, i.e., when different operations are performed in different parts of the data. This is because both conditions increase the chances of distruptive noise accumulation, where some noise is introduced at each computational step while processing encrypted data homomorphically. Recently, it has been shown that by modifying the unencrypted data (i.e., the plaintext) space by a constant, it is possible to perform homomorphic comparison without bootstrapping [12]. The basic comparison of two w-bit unencrypted words (strings) x and y (in the plaintext domain) using

$$f(x,y) = \prod_{i=1}^{w} \overline{x_i \oplus y_i} \quad (1)$$

in this case translates into the bit-level homomorphic $Hom\widehat{XOR}\text{-}OR$ operation implemented in the encrypted data (cyphertext) domain by

$$Hom\widehat{XOR}\text{-}OR(x,y) = \sum_{i=0}^{w} c_{x_i} - c_{y_i} \quad (2)$$

where c_x and c_y represent cyphertexts for input words x and y. This forms the basis of the SCAM scheme. It is important to note that the cyphertext size is a function of encryption parameters n (lattice dimension) and q (ring modulo). Each unencrypted bit expands to $(n+1)\log_2 q$-bits. Hence, a w-bit word expands to $w(n+1)\log_2 q$-bits, which sets the operand length in Equation 2: for medium level security parameters $n = 1052$ and $\log_2 q = 42$, each bit is encrypted as 44226 bits and a single 32-bit word requires 173KB memory [12], [14], [15].

Although homomorphic search is effectively reduced to the simple addition captured by Equation 2 with SCAM, such an unconventionally long addition cannot be handled in traditional hardware efficiently. In a nutshell, SCAM operation relies on a single word comparison in hardware. Since the words to compare are in encrypted domain, the comparison operation reduces to a long addition for each bit in the unencrypted word. We will next cover how H-CRAM is architected to handle the complex memory requirement of this operation.

III. H-CRAM Architecture

A. Adder Architecture

Per Equation 2, SCAM reduces to a basic addition, yet on unconventionally long operands. Accordingly, along with memory accesses at scale, addition represents a critical acceleration target where adders optimized for much shorter and more typical operand widths fall short of meeting the performance requirement. This $(n+1)\log_2 q$-bit-long addition results in significant latency, that would grow with $O(N)$ for $N = (n+1)\log_2 q$ bit operands using a standard Ripple-Carry Adder (RCA)[3]. H-CRAM by construction, addresses the memory access problem at scale, and adapts a Carry-Lookahead Adder (CLA) [24] of $O(N \log N)$ complexity as a more scalable solution. Note that an order of $O(N)$ parallelism in CLA reduces gate depth to $O(\log N)$, which allows significant speedup over RCA.

Previous work on hardware acceleration of SCAM [15] to handle this challenge uses Carry-Select Adder instead, which has $O(\sqrt{N})$ worst-case latency for N bit operands. Although this brings a significant reduction in latency, $O(\log N)$ scalability is possible using logarithmic tree adders. Several adders such as Sklansky [25], Kogge-Stone [26], Han-Carlson [27] or Knowles [28] have $\log N$ or $\log(N+1)$ gate depth, but they are not suitable for homomorphic search acceleration on CRAM fabric either because they require non-homogeneous logic in each stage and/or excessive fanout, which would easily wipe out any scalability benefit . Therefore, we use the standard Carry-Lookahead Adder (CLA) [24]. Fig.2 illustrates the dataflow, considering $(n+1)\log_2 q$–bit addition of two operands, representing the query and the reference for search. CLA involves generate/propagate pairs (G/P) for each bit, which are calculated by A-blocks. Next comes a forward pass (B-blocks) followed by a backward pass (C-blocks) of the tree logic to calculate and propagate the carry values. Then, a stage of two `XOR` gates per bit (D-blocks) computes partial sums. Finally, partial sums are combined (reduced) in a tree structure to $(n+1)\log_2 q$-bits to check if they are all zero (E-blocks), and are reduced one more time (F-blocks) to obtain the end result.

B. Topology

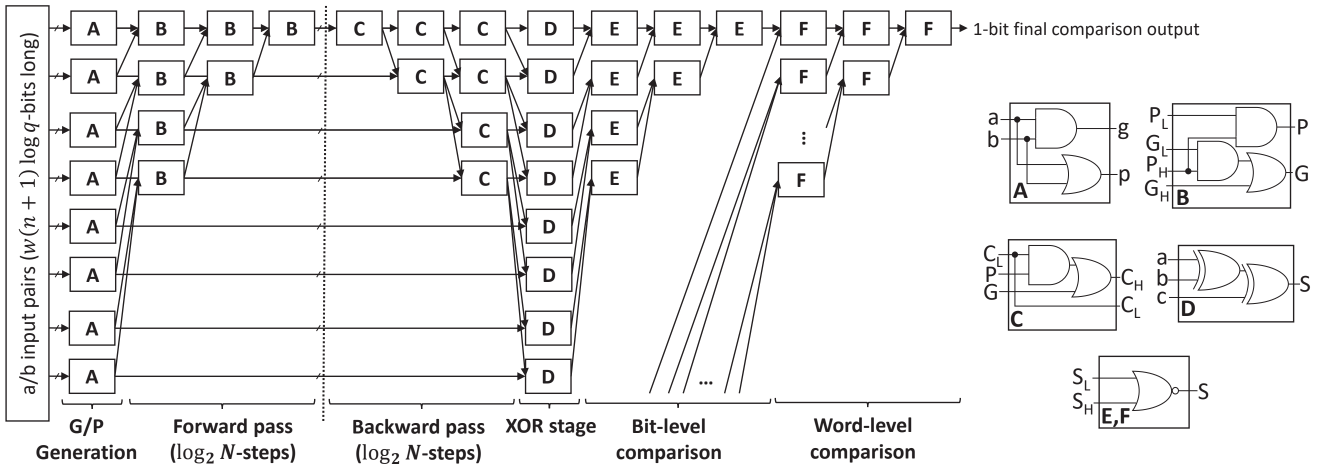
Although the adder architecture from Fig.2 can be implemented as is, in an unrolled fashion, in CRAM, hardware (hence, interconnect) reuse at the block-level can reduce the overall area significantly, where each block being a computational block described in Sec.III-A. By using a De Bruijn graph [29] to implement the dataflow between, we can survive by mapping only a single stage (each corresponding to a column in Fig.2) of blocks to CRAM at a time. De Bruijn graph has a diameter of $\log_2 N$ for N vertices (nodes), hence, represents a natural choice to implement tree structures from Fig.2. Such a tree structure is each part of the entire graph with a binary tree connectivity, where the graph nodes are computational blocks and the edges are the electrical links. In other words, any two arbitrary nodes in an N-node De Bruijn graph are connected to each other with at most $\log_2 N$ edges. This means any N-node De-Bruijn graph can span N-node trees *temporally*, over $\log_2 N$ *time* steps. Recall from Fig. 2 that in each stage, the number of logic blocks that needs to be executed may change. E.g., there are 4, 2, and 1 B blocks in consecutive columns. Disabling half of the nodes in the De Bruijn graph suffices to implement the tree corresponding to each stage in this case. An example 4-node De-Bruijn graph is shown in Fig.3(a). Fig.3(b) illustrates a tree starting from node 0, without loss of generality. Note that in Step 2 all nodes are active while only half of them is active in Step 1. Starting from any other node would result in a similar tree that can reach all other nodes in $\log_2 N$ steps for N nodes. Here active node entails a performing a logic operation at that node, while inactive nodes do not partake in the computation. Note that nodes establish data communication by simple links. De Bruijn topology enables a binary tree that can be constructed starting from any node, which causes only a subset of the links to be used in each tree. As such, the link 3-to-1 is not used, while it could be used if the binary tree started from node 3.

When it comes to physical design, it is possible to implement the De Bruijn graph by using only two routing layers [30]. Routing algorithms for De Bruijn graphs are discussed in detail in [31]. De Bruijn graph is also demonstrated to be an efficient Network-on-Chip (NoC) topology to reduce latency, area, dynamic and leakage power, respectively [30], [32]. One-dimensional (1-D) De Bruijn graph can be extended to 2-D by applying the same connectivity scheme in one dimension along the second dimension in a straightforward manner. Overall footprint is similar to the standard 2-D mesh routing. Therefore we use the De Bruijn topology in H-CRAM to implement the tree structures between the computational blocks from Fig. 2, which we will cover next in detail.

C. Homomorphic Search in CRAM

H-CRAM consists of a standard non-volatile memory controller, which orchestrates 32 processing units. Each processing unit computes a single-bit match. Single-bit matches then get combined to obtain the overall match result at the word level. Processing units are connected to each other over single wire unidirectional connections to this end. Each processing unit

[3]As a hardware optimization, SCAM guarantees that q will be a power of 2 and therefore eliminating modulo operations [12].



a/b input pairs ($w(n+1)\log q$-bits long)
A
B
C
D
E
F
1-bit final comparison output
G/P Generation
Forward pass ($\log_2 N$-steps)
Backward pass ($\log_2 N$-steps)
XOR stage
Bit-level comparison
Word-level comparison
a b g p A
P_L G_L P_H G_H P G B
C_L P G C_H C_L C
a b c S D
S_L S_H S E,F

Fig. 2. Data flow for homomorphic search in H-CRAM, using carry-lookahead adder to implement Equation 2. **G/P Generation:** A-block inputs/outputs are pairs and they are available throughout the same row, i.e., later blocks still use the initially generated a,b,p,g values. Logic gates for computing generate/propagate (G/P) pairs are encapsulated by A-blocks, where a and b represent the input bits. **Forward Pass:** The G/P tree is implemented using B-blocks . Note that P_L,G_L and P_H,G_H are the two G/P pairs, where subscripts L and H denote low and high. B-blocks propagate G/P pairs following a tree structure. At the end of the forward pass P/G pairs are ready for C-blocks. **Backward Pass:** C-blocks implement the reverse tree that calculates and propagates the carry values (carries from the low and high pairs, C_L and C_H, respectively), using the G/P pairs that are obtained after A-block computation. Essentially C-blocks generate carry values using the G/P pairs. **XOR Stage:** D-blocks use the initial inputs a and b as well as the final carry values to compute the addition output. A-, B-, C-, and D-blocks implement the $(n+1)\log_2 q$-bit long addition. **Bit- and Word-level Comparison:** E- blocks perform bitwise comparison for the $(n+1)\log_2 q$-bits while F-blocks do the same at word-level for w-bits. In E and F blocks, low and high S inputs are shown with S_L and S_H.

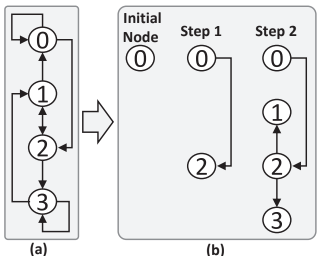


Fig. 3. (a) A 4-node De Bruijn Graph (b) Spanning a binary tree from node 0 in $\log_2 4 = 2$ steps.

comprises 4 512×512 control arrays (CtrlA) and 692 64×64 processing arrays (PA). The processing array granularity is similar to [15]. Each CtrlA provides control signals –to drive word/bit select lines, to distribute gate bias voltages and to manage presets. This is established by a buffered design of wordlines, therefore, no extra compute logic is needed. An initialization pulse which goes through a series of buffers activates wordlines one by one. This essentially corresponds to consecutive reads from the control array in each switching interval, producing a sequence of voltages to be broadcasted to the processing arrays (PAs). PAs are connected to each other in a De Bruijn graph topology with 64-bit buses. PAs perform logic operations as well as reads/writes as instructed by the control signals broadcasted from CtrlAs. The overall architecture is shown in Fig.4; memory layouts for CtrlA and PA, in Fig.5. The buffer space (Buff.) is used for capturing intermediate gate outputs during multi-gate operations.

As an offline step, CtrlAs are pre-programmed with LL, E/OSL, WLR/W, as well as an array enable bit with for each processing array PA. This allows selecting the active columns as well as PAs which will be involved in the computation on a per stage basis. PAs generate and process the G/P values. The hardwired De Bruijn connectivity enables each stage of the computation from Fig.2 to be directly mapped to PAs. If required input bits are not present in the PA (i.e., are coming from another PA), they are first written to the respective PA. Otherwise, computation can fire immediately within the PA.

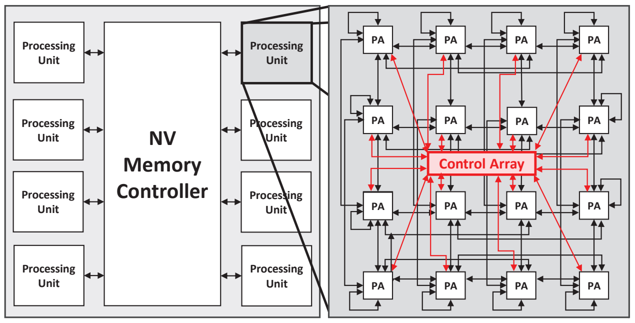


Fig. 4. H-CRAM Architecture (PA: Processing Array).

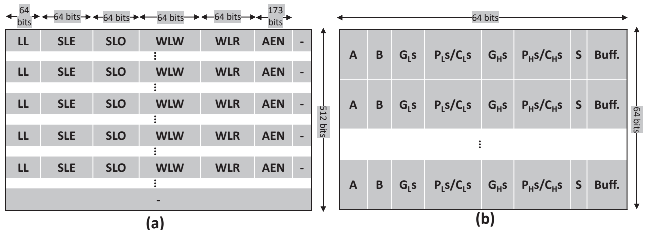


Fig. 5. Memory layouts for: **(a)** Control Array CtrlA. Control line fields are 64 bits each, reflecting the PA data layout. There are 4 CtrlAs and 692 PAs in each Processing Unit. The length of each Array Enable (AEN) field is equal to the number of PAs per CtrlA. For **(b)** Processing Array PA. The fields A, B, G_L, G_H, P_H, P_L, C_H, and C_L correspond to the respective blocks and data from Figure 2.

H-CRAM can support multi-word search by simply activating more processing units to operate simultaneously, as there is no inter-word data dependency.

The search operation in H-CRAM spans the following stages:

1) **Initialization:** Input strings are written to all arrays. A single startup pulse is provided to fire computations. Initial gate operations which are captured by A-blocks in Fig.2 are performed on the input strings, and G/P pairs are produced. G/P pairs are then forwarded to the respective PAs to perform the next stage.
2) **Forward pass:** Respective PAs perform forward pass

computations using intermediate G/P pairs (B-blocks from Fig.2). The new G/P pairs produced this way are then forwarded to the respective output arrays for the next step. This operation repeats for $\log_2((n+1)\log_2 q)$ steps.

3) **Backward pass:** Respective PAs perform backward pass computations (C-blocks from Fig.2). Recall that C-blocks use as inputs G/P pairs generated by A-blocks. The new carry bits produced in this manner are then forwarded to respective PAs for the next step. This operation also repeats for $\log_2((n+1)\log_2 q)$ steps.
4) **XOR:** Respective PAs perform two XOR gates per bit to obtain the partial sum (D-blocks in Fig.2). D-blocks inputs are a and b. The produced sum gets forwarded to respective PAs for the next stage.
5) **Comparisons:** The final stage involves a $\log_2((n+1)\log_2 q)$ step bit-level comparison (implementing E-blocks in Fig.2) followed by a $\log_2 w$ step word-level comparison (implementing F-blocks in Fig.2).

IV. Evaluation

We evaluate H-CRAM using a hierarchical CRAM simulator similar to [33], [34] which covers circuit models based on state resistances from Table I. In this context, we first decompose high-level constructs to arithmetic operations which we then express in terms of CRAM library logic gates. This makes direct extraction of the overall energy and latency from lower level device and circuit parameters possible. In order to estimate peripheral time and energy overhead resulting from sense amplifiers and row decoders, we deploy NVSim [35]. We use NVSim output only to derive pessimistic estimates for the peripheral circuitry overhead for a desired array size. All dynamic computation cost comes from the CRAM simulator. The rest of our configuration parameters, including the cryptographic ones, are given in Table I. The total memory footprint of the proof-of-concept accelerator is 9.5MB for single word comparison, which spans both processing and control arrays.

TABLE I
Experimental Parameters

Parameter	Value
P state resistance	7.34 $k\Omega$
AP state resistance	76.39 $k\Omega$
Switching Time	1 ns
Switching Current	3 μA
Ring Modulo $\log_2 q$	42
Lattice Dimension n	1052 ($\lambda = 80$), 1318 ($\lambda = 128$)
Word size w	32
Memory size	5.5 MB (Processing), 4 MB (Control)

Our performance (latency), energy, and energy efficiency (in terms of energy delay product, EDP) results are provided in Table II. We use three representative baselines for comparison: *SCAM* is the original ASIC design from [12]; *HEGA*, the HMC-based system-on-chip design; and *2T+1 FeFET* [15], the near-memory implementation where computation is performed at the memory periphery. Further, we consider two design points for our accelerator: *H-CRAM RCA* is the slow variant which uses the standard Ripple-Carry Adder; and *H-CRAM CLA* is the optimized variant using Carry-Lookahead Adder. Note that H-CRAM RCA is an unoptimized and highly inefficient design and presented here just as a reference to better understand the complex trade-offs in the design space.

Our results show that, when compared to HEGA, H-CRAM RCA is 1.92× faster, while consuming significantly lower energy. Compared to the 2T+1 FeFET, on the other hand, H-CRAM RCA is 11.23× faster but consumes 11.6× more energy, and as a result, has a similar energy efficiency. As noted previously, we consider H-CRAM RCA only as a comparison point, and it does not deliver a feasible performance vs. energy trade-off. 2T+1 FeFET design features a relatively more efficient adder architecture, but H-CRAM CLA still manages to outperform 2T+1 FeFET by a large margin in terms execution time, which points to H-CRAM's efficiency as an effective compute substrate.

TABLE II
Performance and Energy Comparisons

Implementation	Latency (μs)	Energy (nJ)	EDP (fJs)
Intel Xeon [12]	57.7	75010†	432810†
SCAM [12]	9.47	11.41	108.05
HEGA [14]	0.61	1401†	854.67†
2T+1 FeFET [15]	3.56	0.71	2.52
H-CRAM RCA	221.13	1.37	302.95
H-CRAM CLA	0.317	8.24	2.61

† Since the actual numbers are not reported, we conservatively estimate energy using the lowest idle power state (C6) for the baseline 8-core Intel Xeon Processor [36].

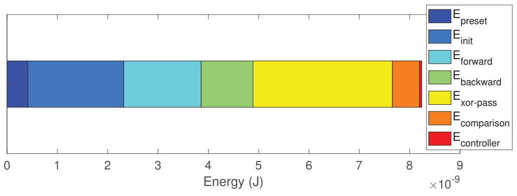


Fig. 6. Energy breakdown for H-CRAM CLA.

The energy breakdown of operations for H-CRAM CLA is provided in Fig.6. Only a fraction of the total energy is consumed indirectly, by preset and controller operations, respectively. On the other hand, a significant energy is spent during initialization, i.e., while writing the inputs to the Processing Arrays as well as while computing the G/P pairs. Final XOR pass dominates the overall energy consumption mainly because XOR gates are implemented using multiple basic gates. We consider communication overhead, i.e., array read/write energy as part of the energy overhead for each type of operation in Figure 6.

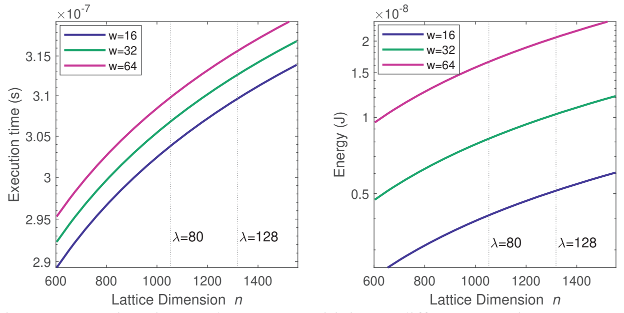


Fig. 7. Execution time and energy sensitivity to different security parameters and plaintext word lengths. $\lambda = 80$ corresponds to medium; $\lambda = 128$, to high security, respectively, as defined in [12].

In order to show the impact of security parameters and plaintext width, we sweep lattice dimension n and word length w and report the execution time and energy in Figure 7. Security parameter λ comes from the asymptotic complexity of $O(2^\lambda)$ operations to decrypt an encrypted bit, and is defined as a function of encryption parameters: $\lambda = 80$ is characterized by $n = 1052$ and $\lambda = 128$ is characterized by $n = 1318$, as described in [12].

H-CRAM supports arbitrary parallelism, and as processing of each word can proceed independently, can easily be extended to support multi-word search as is the case for HEGA [15].

HEGA relies on a full-fledged multicore processor, 8 GB memory, and extra logic on top in vault processing units. To put this into prespective, an H-CRAM CLA design with 8 GB would enable 862 parallel SCAM operations, without any need for extra logic or processing element outside of the arrays. However, it is important to note that HEGA uses this much memory to store the database; and H-CRAM, both for storage and computation while achieving similar scalability. Thanks to strictly adhering to standard memory semantics and restricting the processing/control array connectivity to a minimal topology, scalability can be abstracted from the device level performance parameters such as pulse period. At the same time, H-CRAM CLA is optimized for area using the De Bruijn topology. For higher throughput, we could directly hardwire the data dependency graph from Fig.2 to enable seamless pipelining. Although the original SCAM [12] is proposed as a biomarker search application, the underlying search architecture is completely independent of the application domain. Finally, system integration for H-CRAM is straightforward since it only entails a standard memory interface for data transfer and control.

V. Conclusion

In this paper, we introduce H-CRAM, a novel, spintronic in-memory accelerator for homomorphic search. We provide an area-efficient latency-optimized proof-of-concept design and pinpoint challenges and opportunities for further optimization. This design outperforms the fastest baseline for comparison by $1.92\times$, while consuming significantly lower energy. When compared to another representative alternative optimized for low power, on the other hand, our design is $11.23\times$ faster while maintaining similar energy efficiency. The improvements do not only stem from the low-energy computing medium and the tight coupling thereof with a data-hungry application domain, but a network topology that enables energy-efficient data communication. Unlike alternatives, H-CRAM's scalability is only limited by memory availability since it does not incur any extra logic for computation, neither in array periphery nor in control lines. In summary, our implementation demonstrates an area-efficient solution with execution time optimization, yet it can be tailored for high-parallelism or high-throughput needs easily –by exploiting the inherent multi-grain parallelism of CRAM– to reveal further feasible design options in the complex execution time vs. energy vs. area trade-off space. We will explore opportunities for more generic homomorphic computation in our future work.

Acknowledgment

This work was supported in part by NSF grant no. SPX-1725420.

References

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, 1978.

[2] L. Chen and Z. Zhang, "Bootstrapping fully homomorphic encryption with ring plaintexts within polynomial noise," in *International Conference on Provable Security*, Springer, 2017.

[3] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2015.

[4] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2016.

[5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer, 2017.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, 2018.

[7] T. Zhou, X. Yang, L. Liu, W. Zhang, and N. Li, "Faster bootstrapping with multiple addends," *IEEE Access*, vol. 6, 2018.

[8] R. A. Hallman, K. Laine, W. Dai, N. Gama, A. J. Malozemoff, Y. Polyakov, and S. Carpov, "Building applications with homomorphic encryption," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, ACM, 2018.

[9] S. Carpov and T. Tortech, "Secure top most significant genome variants search: idash 2017 competition," *BMC medical genomics*, 2018.

[10] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *International Cryptology Conference*, Springer, 2018.

[11] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Simulating homomorphic evaluation of deep learning predictions," in *International Symposium on Cyber Security Cryptography and Machine Learning*, Springer, 2019.

[12] S. Bian, M. Hiromoto, and T. Sato, "Scam: Secured content addressable memory based on homomorphic encryption," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017.

[13] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[14] A. O. Glova, I. Akgun, S. Li, X. Hu, and Y. Xie, "Near-data acceleration of privacy-preserving biomarker search with 3d-stacked memory," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[15] D. Reis, M. T. Niemier, and X. S. Hu, "A computing-in-memory engine for searching on homomorphically encrypted data," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 2019.

[16] J.-P. Wang and J. D. Harms, "General structure for computational random access memory (cram)," Dec. 29 2015. US Patent 9,224,447.

[17] H. Nejatollahi and S. Gupta, "Cryptopim: In-memory acceleration for rlwe lattice-based cryptography," in *Design Automation Conference*, 2020.

[18] J.-P. Wang, S. S. Sapatnekar, C. H. Kim, P. Crowell, S. Koester, S. Datta, K. Roy, A. Raghunathan, X. S. Hu, and M. Niemier, "A pathway to enable exponential scaling for the beyond-cmos era," in *DAC*, 2017.

[19] G. Prenat, K. Jabeur, P. Vanhauwaert, G. Di Pendina, F. Oboril, R. Bishnoi, M. Ebrahimi, N. Lamard, O. Boulle, and K. Garello, "Ultra-fast and high-reliability sot-mram: From cache replacement to normally-off computing," *IEEE Trans. on Multi-Scale Computing Systems*, 2015.

[20] A. D. Kent and D. C. Worledge, "A new spin on magnetic memories," *Nature nanotechnology*, vol. 10, no. 3, 2015.

[21] "Everspin EMD3D256MxxBS1 datasheet." https://www.everspin.com/supportdocs/EMD3D256M08G1-150CBS1, 2018. Accessed: 2021-08-12.

[22] M. Zabihi, Z. Zhao, D. Mahendra, Z. I. Chowdhury, S. Resch, T. Peterson, U. R. Karpuzcu, J.-P. Wang, and S. S. Sapatnekar, "Using spin-hall mtjs to build an energy-efficient in-memory computation platform," in *20th ISQED*, IEEE, 2019.

[23] J.-P. Wang, S. S. Sapatnekar, U. R. Karpuzcu, Z. Zhao, M. Zabihi, M. S. Resch, Z. I. Chowdhury, and T. Peterson, "Computational random access memory (cram) based on spin-orbit torque devices," Sept. 3 2020. US Patent App. 16/803,454.

[24] G. B. Rosenberger, "Simultaneous carry adder," 1960. US Patent 2,966,305.

[25] J. Sklansky, "Conditional-sum addition logic," *IRE Trans. on Electronic computers*, no. 2, 1960.

[26] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. on computers*, vol. 100, no. 8, 1973.

[27] T. Han and D. A. Carlson, "Fast area-efficient vlsi adders," in *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*, 1987.

[28] S. Knowles, "A family of adders," in *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*, 1999.

[29] N. G. De Bruijn, "A combinatorial problem," in *Proc. Koninklijke Nederlandse Academie van Wetenschappen*, vol. 49, 1946.

[30] M. Hosseinabady, M. R. Kakoee, J. Mathew, and D. K. Pradhan, "De bruijn graph as a low latency scalable architecture for energy efficient massive nocs," in *2008 Design, Automation and Test in Europe*, 2008.

[31] G. Liu and K. Y. Lee, "Optimal routing algorithms for generalized de bruijn digraphs," in *1993 International Conference on Parallel Processing-ICPP'93*, vol. 3, IEEE, 1993.

[32] R. Sabbaghi-Nadooshan, M. Modarressi, and H. Sarbazi-Azad, "The 2d dbm: An attractive alternative to the simple 2d mesh topology for on-chip networks," in *2008 IEEE International Conference on Computer Design*.

[33] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Mouse: Inference in non-volatile memory for energy harvesting applications," in *International Symposium on Microarchitecture (MICRO)*, 2020.

[34] H. Cılasun, S. Resch, Z. I. Chowdhury, E. Olson, M. Zabihi, Z. Zhao, T. Peterson, J.-P. Wang, S. S. Sapatnekar, and U. Karpuzcu, "Crafft: High resolution fft accelerator in spintronic computational ram," in *Design Automation Conference (DAC)*, 2020.

[35] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, vol. 31, July 2012.

[36] "Intel® Xeon® Processor E5 v2 Family: Datasheet, Vol. 1." https://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-v2-datasheet-vol-1.html. Accessed: 2020-09-17.