Exploring Large Integer Multiplication for Cryptography Targeting In-Memory Computing

Florian Krieger
Graz University of Technology
Graz, Austria
florian.krieger@tugraz.at

Florian Hirner
Graz University of Technology
Graz, Austria
florian.hirner@tugraz.at

Sujoy Sinha Roy
Graz University of Technology
Graz, Austria
sujoy.sinharoy@tugraz.at

***Abstract*—Emerging cryptographic systems such as Fully Homomorphic Encryption (FHE) and Zero-Knowledge Proofs (ZKP) are computation- and data-intensive. FHE and ZKP implementations in software and hardware largely rely on the von Neumann architecture, where a significant amount of energy is lost on data movements. A promising computing paradigm is computing in memory (CIM) which enables computations to occur directly within memory thereby reducing data movements and energy consumption. However, efficiently performing large integer multiplications – critical in FHE and ZKP – is an open question, as existing CIM methods are limited to small operand sizes. In this work, we address this question by exploring advanced algorithmic approaches for large integer multiplication, identifying the Karatsuba algorithm as the most effective for CIM applications. Thereafter, we design the first Karatsuba multiplier for resistive CIM crossbars. Our multiplier uses a three-stage pipeline to enhance throughput and, additionally, balances memory endurance with efficient array sizes. Compared to existing CIM multiplication methods, when scaled up to the bit widths required in ZKP and FHE, our design achieves up to 916x in throughput and 281x in area-time product improvements.**

***Index Terms*—Computing In Memory, Large Integer Multiplication, Karatuba Multiplication**

I. Introduction

Zero-knowledge Proofs (ZKP) and Fully Homomorphic Encryption (FHE) offer privacy-preserving properties that are highly relevant for modern applications. Nonetheless, they introduce magnitudes of computational overhead and commonly involve several Gigabytes of data [1], [2]. For example, ZKP proofs with a circuit size of 2^{26}, 256-bit polynomial coefficients, and 384-bit elliptic curve points, require 8.8GB of memory [3]. Recent research focuses on accelerating these data-intensive schemes using von Neumann-based architectures including CPU [4], FPGA [3], and ASIC [2] platforms. While these approaches enhance arithmetic operations, the large data transfers between the processor and memory result in significant energy and latency overheads.

A promising alternative to address the memory bottleneck is computing in memory (CIM). The CIM paradigm avoids expensive data movements and performs computations directly in the memory array [5]. This resolves the limitations of classical von Neumann architectures and offers novel opportunities for data-intensive cryptographic schemes. However, performing large integer multiplication – a crucial operation of ZKP and FHE with hundreds of bits wide operands – efficiently in CIM is still an unsolved challenge. Existing in-memory multipliers [6]–[9] do not consider cryptographic demands and, hence, only support small operand sizes. In addition, existing CIM multipliers show a drastic increase in area or latency when scaled up to cryptographically relevant operand sizes. This stems from the commonly used schoolbook multiplication method. Although the schoolbook method is simple to implement, it prevents an efficient in-memory integration of large integer multiplication.

This project was funded in part by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark

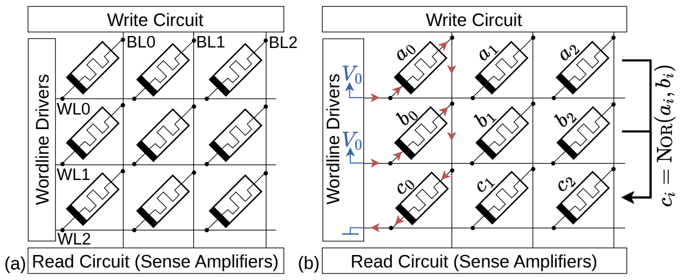


Fig. 1. 3×3 ReRAM array (a) and MAGIC NOR operation (b).

This paper addresses this limitation and explores more efficient large integer multiplication methods for CIM. Our contributions are fourfold and can be summarized as follows: (1) We investigate the Toom-Cook and Karatsuba methods and discuss their suitability for an efficient CIM deployment. This reveals the Karatsuba method as best suited. (2) We hence design the first Karatsuba-based CIM multiplier for ZKP and FHE-relevant large operand sizes. Our multiplier design benefits from algorithmic and architecture design optimizations. (3) On the algorithmic level, we unroll the Karatsuba tree to enhance the uniformity of operations, which is crucial for efficient CIM implementations. (4) On the implementation level, we present an optimized Kogge-Stone adder for the needed additions in Karatsuba. Moreover, we pipeline our Karatsuba multiplier to enhance the throughput of our design.

Compared to existing CIM multipliers, our Karatsuba approach scales better to cryptographically relevant operand sizes. We thus achieve a significant improvement of up to 916× in performance and 281× in area-time product with respect to the state of the art. In addition, we relax memory array constraints and consider the endurance limitations of memory cells.

II. Background

A. Memristive Crossbars

Resistive random access memory (ReRAM) is an emerging non-volatile memory technology using voltage-controlled resistors to store data. Each resistor (also called memristor) stores one bit of information in its resistance: High resistance encodes logic zero and low resistance encodes logic one [5]. Memristors are arranged in a memory grid. Fig. 1a shows a 3×3 grid with horizontal word lines (WL) and vertical bit lines (BL).

For storing one word of information to a certain word in the memory, the word line driver in Fig. 1a selects the target WL and keeps the other WLs idle. Simultaneously, the write circuit applies V_{Set} (or V_{Reset}) to the BL to write logic one (or logic zero) [10]. To read one word from memory, the word line driver applies a small voltage V_{Read} to the target WL. This voltage is below the memristor threshold, ensuring the memristors preserve their initial resistance levels. The resulting current through each bit line, determined by the memristor's resistance, is measured by a sense amplifier.

Endurance: An important aspect of ReRAM design is the limited endurance of the memristors. Memristors face wear and tear effects during writing which limits the lifetime of ReRAM cells between 10^{10} and 10^{11} write cycles per cell [10]–[12]. Hence, a ReRAM-based CIM design must reduce the number of write operations to prolong the cell lifetime. Furthermore, write operations should be equally distributed over the whole array to balance wear and tear across all cells.

B. Memristor-Aided Logic (MAGIC)

Recent works proposed different methods to compute directly in ReRAM memory. Thereby, only simple boolean functions are supported such as MAJORITY [13] or IMPLY [14] due to the constrained memory environment. One of these works is Memristor-Aided Logic (MAGIC) [15], which provides NOR operations across memory rows. The benefit of MAGIC is twofold. First, NOR is a functionally complete gate. This means that any boolean function can be implemented using NOR. Second, MAGIC NOR is computed entirely in the memory array without moving data to the periphery. In addition, MAGIC operations preserve the state of the input memristors, which is not the case in IMPLY [14]. Hence, in MAGIC, multiple sequential operations are possible with the same input memristor.

Fig. 1b shows an example of the MAGIC NOR operation within resistive memory [9]. The two input values to the NOR gate (denoted as a_i and b_i, $i \in \{0, 1, 2\}$) are stored in two separate memory rows but within the same bit line, and the output memristor c_i is initialized to logic one (low resistance). Then, the word line driver applies V_0 to the WLs of the input rows and ground potential to the row of the output memristor (blue in Fig. 1b). This causes a current flow (red in Fig. 1b) through the input memristors and the output memristor. Suppose at least one input memristor has low resistance (logic one). In that case, the current through the output memristor is large enough to force the output memristor into the high resistance state (logic zero). In contrast, if both input memristors show high resistance (logic zero), the current through the output memristor is insufficient to change the memristor state, and it retains logic one. Thus, we obtain a two-input NOR gate.

This NOR operation does not only apply to a single bit line but to all bit lines in parallel, as shown in Fig. 1b. All three $c_i = \text{NOR}(a_i, b_i)$ are evaluated simultaneously corresponding to a single-instruction-multiple-data (SIMD) computation. In addition to NOR, the MAGIC technique also allows performing NOT operations [15]. Thereby, the NOT operation is a special case of NOR with just one input operand.

C. Related works for CIM integer multiplication

Several works propose integer multiplications for in-memory computing [6]–[9], [16], [17]. They mostly focus on integer multiplication with small (8-bit) to medium-sized (up to 64-bit) operands and utilize the simple schoolbook multiplication method. Small integer sizes are well chosen as they fit common machine word sizes in microcontrollers and CPUs. However, many cryptographic applications require significantly larger multiplications reaching up to 384 bits for pairing-based ZKP [2], [18]. Straightforwardly scaling existing in-memory multipliers to such large operand sizes is inefficient.

The schoolbook multiplication has $\mathcal{O}(n^2)$ complexity for n-bit multiplication. Therefore, most of the existing works have either quadratic time complexity [7], [16], [17] or quadratic area complexity [6], [8]. One work with quadratic area complexity is [8], where increased area is traded for low runtime, making it one of the fastest CIM multipliers. The authors use MAJORITY logic to build a Wallace-tree-based multiplier [19], enabling high parallelism at the cost of quadratic area consumption.

The work [9] is exceptional as it has $\mathcal{O}(n \log(n))$ runtime and $\mathcal{O}(n)$ area. This efficiency is achieved by dividing the memory array into independent partitions, where the partitions compute in parallel to boost performance. However, this approach performs the whole multiplication in just a single bit line, leading to a very long bit line, especially in large integer scenarios. For example, a $n = 384$-bit multiplication requires a bit line with 5,369 memristors, which limits the practicability due to increasing parasitic resistances in bit lines [7], [20].

III. ALGORITHM EXPLORATION FOR CIM-BASED LARGE INTEGER MULTIPLICATIONS

Selecting the multiplication algorithm for large integer multiplication is a crucial aspect of CIM designs. Most importantly, the algorithm must scale well for large operands while allowing a simple control flow with in-memory-friendly operations. The operations should be easily decomposable into NOR gates and allow a high level of parallelism to profit from CIM designs.

This section presents the different multiplication methods such as schoolbook [21], Toom-Cook [22], and Karatsuba [23] algorithms. Furthermore, we discuss the suitability of these algorithms for efficient in-memory implementation.

A. CIM Efficiency Exploration of Schoolbook Multiplication

The schoolbook method [21] is the simplest approach for integer multiplication. It multiplies each bit of one operand with all bits of the other operand. These multiplications at bit-level correspond to bit-wise AND operations. The results of the partial multiplications are added to yield the final result.

Suitability for CIM: The schoolbook method contains just AND operations and additions of partial results. Both operations are CIM-friendly due to a regular data flow. Further, the additions can be parallelized by, for example, applying a Wallace tree structure [8]. However, the schoolbook method does not scale well for large operands due to the quadratic growth of AND operations with the bit-size of operands. Hence, the schoolbook method is not ideal for large integer multiplication.

B. CIM Efficiency Exploration of Toom-Cook Multiplication

The Toom-Cook method [22], [24] has $\mathcal{O}(n^c)$ complexity, with $1<c<2$. It splits each operand into k smaller chunks, treating these chunks as the coefficients of polynomials. These polynomials are then evaluated at a set of strategically important points, and after that, the evaluations are point-wise multiplied. An interpolation step is used to reconstruct the product polynomial. Finally, the product polynomial's coefficients are joined to obtain the integer multiplication result.

Suitability for CIM: Selecting a larger parameter k is advantageous in the point-wise product computation since the operands involved are smaller. However, a large k is a significant drawback in the interpolation due to the quadratic growth of the Vandermonde matrix [25]. This quickly leads to a high number of constant multiplications for interpolation. For example, interpolation requires 25, 49, and 81 multiplications for $k=3$, 4, and 5, respectively. Although these multiplications can be parallelized in the CIM paradigm, they cause a significantly increased area consumption.

In addition to that, selecting $2k-1$ evaluation points supporting both, efficient evaluation and efficient interpolation, is challenging. For example, choosing evaluation points as powers of two allows evaluation to just use additions of shifted coefficients. Yet, in this case, interpolation needs multiplications with non-powers-of-two and potentially fractional values. Supporting these multiplications with sufficient precision in CIM is highly inefficient. Therefore, generic Toom-Cook with large k is not an ideal choice for CIM multiplication. Yet, we select a special case of Toom-Cook, namely the Karatsuba method [23], for our implementation. Karatsuba resolves several issues in the generic Toom-Cook, as discussed in the next subsection.

C. CIM Efficiency Exploration of Karatsuba Multiplication

The Karatsuba multiplication method [23] is a special case of Toom-Cook multiplications with $k=2$. This simplifies Toom-Cook and leads to an algorithmic complexity of $\mathcal{O}(n^{1.58\cdots})$ [23], which is lower than schoolbook multiplication.

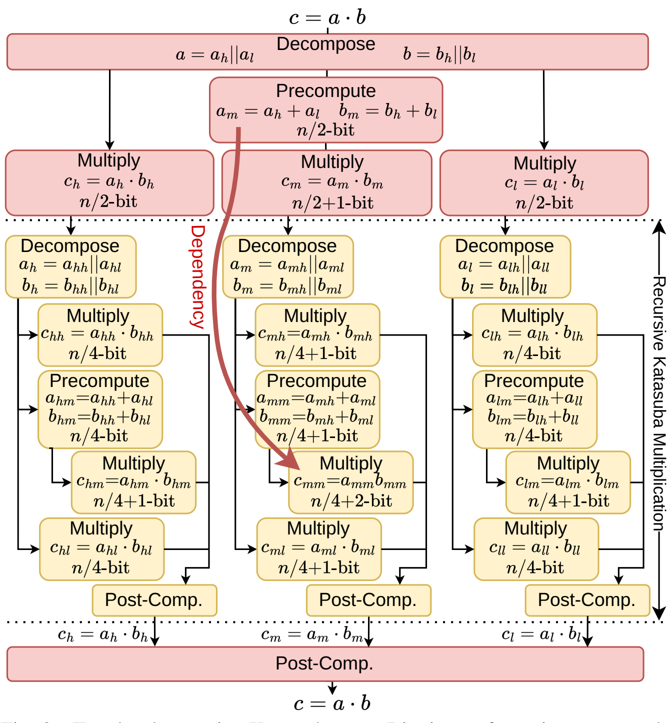
The Karatsuba method computes $c=a\cdot b$ with n-bit wide operands by splitting each operand into two $\frac{n}{2}$-bit wide chunks a_h, a_l and b_h, b_l, respectively, as in (1). Based on these chunks, three multiplications are performed in (2) to compute partial results c_h, c_l, and c_m. Therein, the suffixes stand for $h$igh, $l$ow, and $m$id, respectively. Finally, the partial multiplication results c_h, c_l, and c_m are combined according to (3) to yield the overall result ($\cdot||\cdot$ denotes bit-wise appending). For the remainder of this paper, we refer to the (a_h+a_l) and (b_h+b_l) additions in (2) as precomputation and to (3) as postcomputation.

$$a\cdot b=(a_h||a_l)\cdot(b_h||b_l)=(2^{\frac{n}{2}}a_h+a_l)\cdot(2^{\frac{n}{2}}b_h+b_l) \quad (1)$$

$$c_h=a_h\cdot b_h,\quad c_l=a_l\cdot b_l,\quad c_m=(a_h+a_l)\cdot(b_h+b_l) \quad (2)$$

$$a\cdot b=c=(c_h||c_l)+(c_m-c_h-c_l)\cdot 2^{\frac{n}{2}} \quad (3)$$

Suitability for CIM: One level of Karatsuba decomposition requires two $\frac{n}{2}$-bit additions (2), two $\frac{n}{2}$-bit multiplications (for c_h and c_l in (2)), one $\frac{n}{2}+1$-bit multiplication (for c_m in (2)), two $n+2$-bit subtractions (3) and one $2n$-bit addition (3). Hence, the Karatsuba multiplication is more straightforward than the Toom-Cook approach described in Sec. III-B. We therefore choose Karatsuba multiplication for our CIM multiplier and explain the design aspects for CIM in detail.



$c=a\cdot b$
Decompose
$a=a_h||a_l$
$b=b_h||b_l$
Precompute
$a_m=a_h+a_l$ $b_m=b_h+b_l$
$n/2$-bit
Multiply $c_h=a_h\cdot b_h$ $n/2$-bit
Multiply $c_m=a_m\cdot b_m$ $n/2+1$-bit
Multiply $c_l=a_l\cdot b_l$ $n/2$-bit
Dependency
Decompose $a_h=a_{hh}||a_{hl}$ $b_h=b_{hh}||b_{hl}$
Multiply $c_{hh}=a_{hh}\cdot b_{hh}$ $n/4$-bit
Precompute $a_{hm}=a_{hh}+a_{hl}$ $b_{hm}=b_{hh}+b_{hl}$ $n/4$-bit
Multiply $c_{hm}=a_{hm}\cdot b_{hm}$ $n/4+1$-bit
Multiply $c_{hl}=a_{hl}\cdot b_{hl}$ $n/4$-bit
Decompose $a_m=a_{mh}||a_{ml}$ $b_m=b_{mh}||b_{ml}$
Multiply $c_{mh}=a_{mh}\cdot b_{mh}$ $n/4+1$-bit
Precompute $a_{mm}=a_{mh}+a_{ml}$ $b_{mm}=b_{mh}+b_{ml}$ $n/4+1$-bit
Multiply $c_{mm}=a_{mm}b_{mm}$ $n/4+2$-bit
Multiply $c_{ml}=a_{ml}\cdot b_{ml}$ $n/4+1$-bit
Decompose $a_l=a_{lh}||a_{ll}$ $b_l=b_{lh}||b_{ll}$
Multiply $c_{lh}=a_{lh}\cdot b_{lh}$ $n/4$-bit
Precompute $a_{lm}=a_{lh}+a_{ll}$ $b_{lm}=b_{lh}+b_{ll}$ $n/4$-bit
Multiply $c_{lm}=a_{lm}\cdot b_{lm}$ $n/4+1$-bit
Multiply $c_{ll}=a_{ll}\cdot b_{ll}$ $n/4$-bit
Post-Comp.
Post-Comp.
Post-Comp.
Recursive Katasuba Multiplication
$c_h=a_h\cdot b_h$
$c_m=a_m\cdot b_m$
$c_l=a_l\cdot b_l$
Post-Comp.
$c=a\cdot b$

Fig. 2. Two-level recursive Karatsuba tree. Bit sizes refer to input operands.

1) Recursive Karatsuba: The Karatsuba decomposition requires three partial multiplications at each level shown in (2). The cost of these multiplications can further be reduced by recursively applying Karatsuba decomposition again. A two-level recursive Karatsuba decomposition is shown in Fig. 2. The red operations show the first Karatsuba level, whereas the three partial multiplications are computed by another layer of Karatsuba (yellow). The second layer of Karatsuba again contains precomputation steps and nine actual multiplications.

Fig. 2 also highlights a data dependency (red arrow) across the recursion: The precomputation of a_m and b_m in the first level must be performed before handing the results over to the second Karatsuba level for multiplication. This means that two $\frac{n}{2}$-bit additions are required in precomputation of the first level. In level 2, the $\frac{n}{2}+1$-bit wide results a_m, b_m are again split and fed to $\frac{n}{4}+1$-bit additions. These additions compute a_{mm}, b_{mm} as shown in Fig. 2. Hence, in recursive Karatsuba, each level requires additions with different operand sizes during the precomputation step.

These varying addition sizes are easily implementable in software on von Neumann systems. In contrast, in CIM architectures, each addition size requires a dedicated array dimension, leading to two potential design approaches: (i) either instantiate multiple addition arrays of different sizes, which results in a high area overhead; or (ii) reuse a single addition array of the largest dimension for all additions, which causes underutilization of the array and limits efficiency. Consequently,

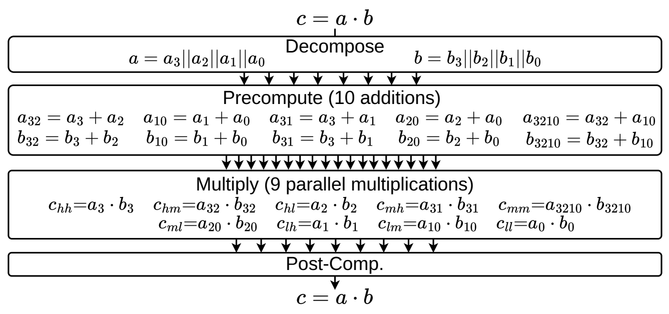

$c = a \cdot b$
Decompose
$a = a_3||a_2||a_1||a_0$
$b = b_3||b_2||b_1||b_0$
Precompute (10 additions)
$a_{32} = a_3 + a_2$ $a_{10} = a_1 + a_0$ $a_{31} = a_3 + a_1$ $a_{20} = a_2 + a_0$ $a_{3210} = a_{32} + a_{10}$
$b_{32} = b_3 + b_2$ $b_{10} = b_1 + b_0$ $b_{31} = b_3 + b_1$ $b_{20} = b_2 + b_0$ $b_{3210} = b_{32} + b_{10}$
Multiply (9 parallel multiplications)
$c_{hh}=a_3 \cdot b_3$ $c_{hm}=a_{32} \cdot b_{32}$ $c_{hl}=a_2 \cdot b_2$ $c_{mh}=a_{31} \cdot b_{31}$ $c_{mm}=a_{3210} \cdot b_{3210}$
$c_{ml}=a_{20} \cdot b_{20}$ $c_{lh}=a_1 \cdot b_1$ $c_{lm}=a_{10} \cdot b_{10}$ $c_{ll}=a_0 \cdot b_0$
Post-Comp.
$c = a \cdot b$

Fig. 3. $L = 2$-level unrolled Karatsuba multiplication.

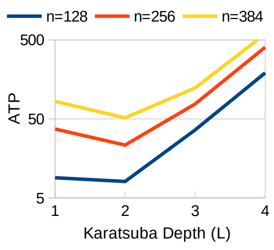


Fig. 4. ATP estimation.

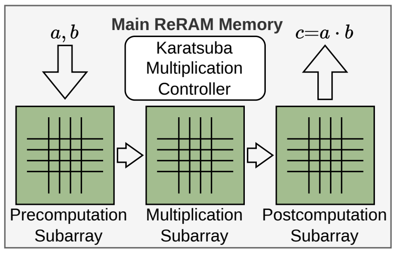


Fig. 5. Our pipelined Karatsuba multiplier.

the non-uniformity of precomputation presents challenges for the CIM implementation of the recursive Karatsuba method.

2) Unrolled Karatsuba: For our implementation, we improve uniformity by unrolling the recursive Karatsuba multiplication. For a Karatsuba depth of L, we directly decompose a, b into 2^L chunks each. We then merge the precomputation stages of all levels into a single precomputation stage, as shown in Fig. 3. Thereby, the precomputation stage only requires between $\frac{n}{2^L}$-bit and $\frac{n}{2^L}+L-1$-bit additions for a Karatsuba depth of L. This increased uniformity in the addition operation allows a more efficient CIM design. Furthermore, the bit size of the addition operands is smaller compared to recursive Karatsuba, which makes unrolled Karatsuba better suited for our design.

The Karatsuba depth L is a central parameter in our design. Increasing the value of L reduces the bit sizes of precomputation and multiplication operands, as discussed above. This has a positive effect on the area and latency of the individual suboperations. However, the overall number of additions and multiplications to be performed significantly increases with larger L. For example, we need 9, 27, and 81 multiplications and 10, 38, and 140 additions in precomputation for $L = 2$, 3, and 4, respectively. This increase counteracts the benefit of smaller suboperations. To find the optimal depth L, we computed the area-time product (ATP) for different depths. The obtained results are presented in Fig. 4 and show that $L = 2$ leads to the lowest ATP across cryptographically relevant multiplication sizes n. Therefore, we choose $L = 2$ depth for our unrolled Karatsuba CIM multiplier design.

IV. Our Karatsuba Multiplier Design for CIM

Our CIM Karatsuba design focuses on efficient large integer multiplication for data-intensive cryptographic schemes, e.g., 64-bit integers for FHE [4] and up to 384-bit for pairing-based ZKP [2], [18]. Achieving a balance between throughput and area consumption remains essential in our investigation.

A. High-Level Design

The unrolled Karatsuba multiplication method, as discussed in Sec. III-C, can be split into three main steps. The first step is the precomputation step, which consists of additions. The second step performs the partial multiplications. Finally, the third step is the postcomputation, which again consists of additions. We realize this three-step process as a three-stage pipelined design for CIM, which is shown in Fig. 5. Each stage of the pipeline uses a dedicated memory subarray. Similar heterogeneous approaches have been proposed for AI applications [26] but are unexplored for large integer multiplication.

Our CIM Karatsuba multiplier is controlled by the *Karatsuba Multiplication Controller* shown in the center of Fig. 5. The controller is responsible for providing the input operands a and b to the precomputation subarray. After precomputation is completed, the intermediate results are passed on to the multiplication subarray. Finally, the partial products are aggregated in the postcomputation stage yielding the multiplication result c, which is stored back to the desired main memory location.

The latency of one such multiplication is the sum of the stage latencies. Yet, the proposed pipelining allows increased throughput of our design by simultaneously operating on three multiplications. The throughput is given by the maximum latency across the three stages. Therefore, balancing the stage's latency is beneficial: Area can be saved in stages with low computational effort, whereas stages with high computational effort should consume more area to reduce their latency. In our pipelined design, the precomputation stage shows the lowest computational effort which allows us to save area resources in the precomputation array. In contrast, the multiplication and postprocessing stages consume more area to lower their latency.

B. Our Kogge-Stone Adder Implementation

The Karatsuba pre- and postcomputation heavily rely on addition operations. To efficiently perform these additions in memory, we implement a Kogge-Stone adder. The Kogge-Stone method [27] is a type of carry lookahead adder, which allows $\mathcal{O}(\log(n))$ latency for n-bit additions. This low latency is achieved through bit-level parallelism which perfectly suits the SIMD paradigm of CIM. In addition, the Kogge-Stone adder has a regular data flow making it an attractive choice for CIM.

An example of a 4-bit Kogge-Stone adder is shown in Fig. 6 to explain our adder design. The 4-bit addends x and y are stored in one memory row each. Then, the so-called propagate bits p_i and generate-bits g_i are computed via several MAGIC Nor and Not operations emulating the Xor and And gates (blue region in Fig. 6). This computation requires 8 clock cycles (cc), independent of the bit width of operands as all bits are processed simultaneously.

The resulting p_i and g_i serve as input to the Kogge-Stone prefix graph, as shown in red in Fig. 6. The prefix graph computes the carry bits g_i^{out} in $\lceil \log_2(n) \rceil$ levels. In our $n = 4$ example, two levels are needed, whereby each node performs computations according to Fig. 6 right. These computations are again implemented using multiple Nor and Not gates. In addition, data needs to be shifted across columns (indicated

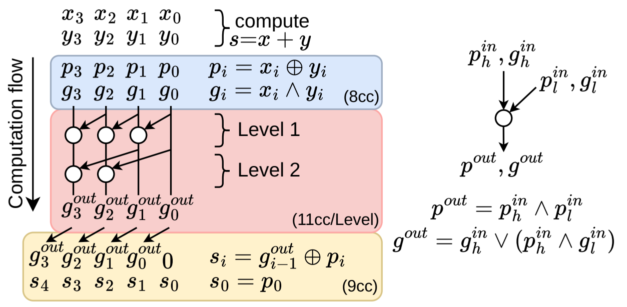


Fig. 6. Schematic of a 4-bit Kogge-Stone adder [28].

by angled edges), which is not supported within the MAGIC array structure [15]. Therefore, we read the values to be shifted from memory, perform the cheap shift operation in a dedicated periphery circuit, and store the shifted value back to memory. Each level of the Kogge-Stone adder has a latency of 11cc (2×2cc for shifting and 7cc for NOR/NOT).

The result of the Kogge-Stone prefix graph $g_i^{(out)}$ is finally used to compute the sum $s = x + y$ (yellow in Fig. 6). This again incorporates a 1-bit shift followed by 5 NOR/NOT to emulate the XOR operation. After that operation, the memory grid is reset to be ready for further computations. The sum computation and resetting takes 9cc. Hence, our n-bit Kogge-Stone adder has an overall latency of $8 + 11\lceil\log_2(n)\rceil + 9$cc.

Our Kogge-Stone design for n-bit addition requires a memory grid with $n+1$ columns. The number of rows is independent of n and amounts to 12 rows for storing intermediate results (scratch region). We achieve the constant number of rows by re-using the same rows for all Kogge-Stone levels. This leads to a total of $2\lceil\log_2(n)\rceil$ write operations to each cell during one addition. These repeated writes slightly degrade the endurance of cells (see Sec. II-A). We address this concern by using wear-leveling [7]. In wear-leveling, the scratch region and the region for input operands and results are constantly exchanged. This balances the write occurrences across the memory array and approximately halves the wear effects. Wear leveling does not lower performance and only has a small control logic overhead. Thus, we achieve a compact and enduring design.

C. Precomputation Stage

The first stage of our n-bit Karatsuba multiplier is the precomputation stage. This stage performs the additions of chunks a_i and b_i as shown in Fig. 3. Since we unroll the Karatsuba tree for $L = 2$ times, a total of 10 additions must be performed during precomputation. Thereby, the additions for a_{3210} and b_{3210} in Fig. 3 operate on $\frac{n}{4} + 1$ bit-wide input operands. Yet, the remaining additions have $\frac{n}{4}$-bit input operands. Thus, we instantiate our Kogge-Stone adder for $\frac{n}{4} + 1$-bit addition to utilize the same array for all additions in precomputation stage.

Within the precomputation array, we store the eight input chunks a_i and b_i, $i \in \{0, 1, 2, 3\}$ in Fig. 3 to the addresses 0 to 7. Furthermore, we reserve addresses 8 to 17 for the 10 addition results. Finally, from address 18 on, the 12-row deep scratch region for our Kogge-Stone adder is placed. This leads to a precomputation array dimension of $(8{+}10{+}12) \times (\frac{n}{4}{+}2)$ for n bit multiplication. For example, in $n = 256$-bit multiplication, the precomputation array consumes $1,980$ memristors.

The latency of the precomputation stage for a n-bit Karatsuba multiplication consists of three main parts. These parts are: (i) writing the 8 inputs into the memory array taking 8cc, (ii) performing 10 Kogge-Stone additions each taking $17 + 11\lceil\log_2(\frac{n}{4} + 1)\rceil$cc, and (iii) resetting the memory array to the initial state in 1cc. Therefore, the preprocessing stage has a total latency of $8 + 10(17 + 11\lceil\log_2(\frac{n}{4} + 1)\rceil) + 1$cc.

D. Multiplication Stage

The multiplication stage takes 18 input operands and performs 9 *small* multiplications, as shown in Fig. 3. To perform these small multiplications, we adopt the method from [9] which also targets the MAGIC technology. In addition, the work [9] shows a proper area-time tradeoff through its $\mathcal{O}(n)$ area and $\mathcal{O}(n\log(n))$ time complexity (see Sec. II-C). We further optimize the multiplier from [9] to lower the area consumption by sharing the memory between input and output operands.

In [9], each small multiplication is performed in a single memory row. This allows to parallelize multiplications by instantiating multiple rows in the memory grid. We use this fact and parallelize the 9 required small multiplications. Thereby, the largest multiplication in our n-bit Karatsuba multiplier computes c_{mm} and has $\frac{n}{4} + 2$ bit wide operands. Based on that, the area consumption of the multiplication stage, which performs 9 parallel multiplications, is $9 \times 12(\frac{n}{4} + 2)$. The latency of the whole multiplication stage is $(\frac{n}{4}{+}2)\cdot(\lceil\log_2(\frac{n}{4}{+}2)\rceil{+}14){+}3$cc.

E. Postcomputation Stage

After the multiplication stage, the postcomputation stage is performed. The postcomputation takes the 9 partial multiplication results which are between $\frac{n}{2}$ and $\frac{n}{2} + 4$ bits wide. Based on these results, the computation of the $2n$ bit-wide overall multiplication result is done following (3).

The postcomputation consists of additions and subtractions which allows us to use another $1.5n$ bit-wide instance of our Kogge-Stone adder. Similar as described in Sec. IV-C, we store the 9 inputs to the postprocessing in memory rows 0 to 5, as shown in Fig. 7a. Therein, the naming of partial multiplication results complies with Fig. 3. Using this memory layout, we perform the required additions and subtractions using the adder scratch area (grey in Fig. 7). We start with computing $\tilde{c}_{lm} = c_{lm} - (c_{ll} + c_{lh})$. Simultaneously, we compute $\tilde{c}_{hm}$ via a batched computation approach. After that, we compute $\tilde{c}_{mm}$.

In the next step, the data in the memory is reordered via read and write operations. This yields the memory layout as shown in Fig. 7b. There, c_h and c_l are computed using one addition each. Unlike in c_h and c_l, computing c_m requires two additions. This is caused by the $\frac{n}{2} + 2$-bit wide c_{ml} value. The two extra bits in c_{ml} prevent a simple appending during computing c_m. Hence, another addition is required.

The computation of c_h, c_l, and c_m is interleaved with reset and reorder operations to fit into the 8 available memory rows. These operations yield the memory content as in Fig. 7c. Next, parts of the array are reset, and $\tilde{c}_m$ is computed. Finally, the overall result $c = a{\cdot}b$ is computed in Fig. 7d. There, we propose another optimization. According to (3), the $\frac{n}{2}$ least significant bits of c_l are directly the LSBs of the overall result c. Hence,

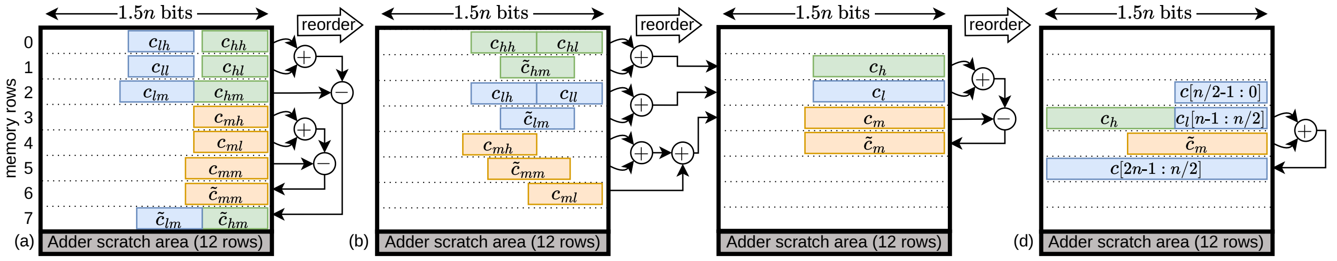


Fig. 7. Execution flow of our Karatsuba postcomputation.

we only perform the addition on the most significant $1.5n$ bits, as shown in Fig. 7d. This reduction of addition size saves 25% of the postprocessing stage area. Therefore, our postprocessing stage has an area consumption of $(8+12) \times 1.5n$ memory cells.

The latency of one postcomputation stage includes the delay for the additions/subtractions, the delay for reset cycles, and the reordering of the operands. The delay for the 11 additions/subtractions is $11 \cdot (11\lceil \log_2(1.5n) \rceil + 17)$cc whereas the delay for reordering/resetting is 18cc. This totals an overall latency of $121\lceil \log_2(1.5n) \rceil + 187 + 18$cc for postcomputation.

F. Application of our CIM Multiplier in Cryptography

Many cryptographic schemes operate over finite rings or fields, needing a modular reduction after integer multiplication. Modular reduction algorithms such as Montgomery [29] and Barrett [30] internally use multiplications readily supported by our multiplier design. Moreover, reduction by a sparse modulus [31] requires additions supported by our Kogge-Stone adder. Thus, our design covers the main building blocks for modular multiplications in cryptography.

V. Implementation Results

Our paper introduces the first large integer multiplier for CIM platforms. We evaluated its performance and verified its correctness using a cycle-accurate simulator. Tab. I presents key metrics such as throughput, area consumption, area-time product (ATP = cells/throughput), and the number of memory cell writes for large bit sizes relevant to data-intensive FHE and ZKP applications. Additionally, we compare our results with scaled-up CIM integer multipliers in the literature. The scaling is necessary as existing CIM designs are limited to short integers only.

The work [6] uses a Imply-based adder to implement an integer multiplier based on the schoolbook approach. Compared to [6], our Karatsuba approach increases the throughput between 3.8× and 17× while lowering the area consumption by up to 11.8×. The ATP improves by a factor of 7× to 204× as the integer size n increases. The authors of [7] use MAGIC Nor gates and present results for a schoolbook multiplier used in image processing. Compared to [7], we achieve between 49× and 916× higher throughput at the cost of 3.5× more area; hence our design yields a 14× to 281× better ATP. In addition, the maximum write operations to one cell are 1.6× to 5.2× less in our Karatsuba design than in [7]. The work presented in [8] uses Majority gates to implement a Wallace-Tree-based multiplier. The work trades low latency for large area consumption thus requiring up to 1.2 million memory cells for multiplication, which is 47× larger than

TABLE I
Comparison of area and throughput to related works.

Work	n	Throughput Multipl./Mcc[†]	Area Cells	ATP Cells/Throughp.	Max. Writes
[6]	64*	243 (3.8×)	8,258	34 (7×)	n.r.
	128*	105 (7.9×)	32,898	312 (30×)	n.r.
	256*	46 (15×)	131,330	2.8k (119×)	n.r.
	384*	28 (17×)	295,298	10.7k (204×)	n.r.
[7]	64	19 (49×)	1,275	67 (14×)	128
	128*	5 (176×)	2,555	540 (53×)	256
	256*	1.2 (599×)	5,115	4.3k (183×)	512
	384*	0.5 (916×)	7,675	14.7k (281×)	1,024
[8]	64	2,475 (0.37×)	32,960	13 (3×)	2
	128*	1,155 (0.72×)	131,312	114 (11×)	2
	256*	525 (1.3×)	524,576	999 (42×)	2
	384*	313 (1.5×)	1,18M	3.8k (72×)	2
[9]	64*	779 (1.2×)	889	1.1 (0.2×)	256
	128*	372 (2.2×)	1,785	4.8 (0.5×)	512
	256*	177 (4.0×)	3,577	20 (0.8×)	1,024
	384*	115 (4.2×)	5,369	47 (0.9×)	1,536
Our	64	927 (1×)	4,404	4.8 (1×)	81
	128	833 (1×)	8,532	10 (1×)	92
	256	706 (1×)	16,788	24 (1×)	134
	384	479 (1×)	25,044	52 (1×)	198

*scaled-up results; [†]Multiplications in 10^6 clock cycles; n.r.: not reported

our design for $n = 384$. This high area consumption allows [8] to distribute write occurrences over more memory cells leading to higher endurance. However, our design has a smaller crossbar dimension and improves the ATP between 3× and 72×, compared to [8]. Finally, we compare with [9] which performs integer multiplication in a single row. For $n = 384$-bit multiplication, [9] needs $5,369$ memristors in one memory row. Such wide rows are less practical due to increasing parasitic effects as reported in [7], [20]. Moreover, [9] performs between 256 and $1,536$ write operations to the same memory cells for $n = 64$ to 384. In comparison, our design reduces the memory row length by 4× and decreases write operations by up to 7.8×.

VI. Conclusion

This paper addressed the challenge of large integer multiplications in CIM which are the fundamental operations in data-intensive ZKP and FHE schemes. We presented a comprehensive algorithm exploration covering Toom-Cook and Karatsuba multiplication and found Karatsuba as most effective for CIM designs. Thus, we designed the first Karatsuba large integer multiplier for CIM which incorporates our Kogge-Stone adder and a three-stage pipeline to enhance throughput. In addition, our design was optimized to reduce crossbar sizes while increasing memory endurance. Compared to prior work, the proposed design shows up to 916× and 281× improvement in throughput and ATP, thereby contributing to the efficient and scalable CIM support of data-centric ZKP and FHE schemes.

References

[1] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252. [Online]. Available: https://doi.org/10.1145/3466752.3480070

[2] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 416–428, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235386321

[3] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao, "Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024, pp. 33–39.

[4] "OpenFHE: Open-Source Fully Homomorphic Encryption Library," https://github.com/openfheorg, 2023, accessed: 2024-09-19.

[5] D. Ielmini and H.-S. P. Wong, "In-memory computing with resistive switching devices," *Nature Electronics Vol 1*, pp. 333–343, 2018. [Online]. Available: https://doi.org/10.1038/s41928-018-0092-2

[6] D. Radakovits, N. Taheri Nejad, M. Cai, T. Delaroche, and S. Mirabbasi, "A memristive multiplier using semi-serial imply-based adder," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 5, pp. 1495–1506, 2020.

[7] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, "Imaging: In-memory algorithms for image processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4258–4271, 2018.

[8] V. Lakshmi, J. Reuben, and V. Pudi, "A novel in-memory wallace tree multiplier architecture using majority logic," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 3, pp. 1148–1158, 2022.

[9] O. Leitersdorf, R. Ronen, and S. Kvatinsky, "Multpim: Fast stateful multiplication for processing-in-memory," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1647–1651, 2022.

[10] 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, 2016.

[11] J. J. Yang, M.-X. Zhang, J. P. Strachan, F. Miao, M. Pickett, R. Kelley, G. Medeiros-Ribeiro, and S. Williams, "High switching endurance in taox memristive devices," *Applied Physics Letters*, vol. 97, pp. 232 102–232 102, 12 2010.

[12] H. Y. Lee, Y. S. Chen, P. S. Chen, P. Y. Gu, Y. Y. Hsu, S. M. Wang, W. H. Liu, C. H. Tsai, S. S. Sheu, P. C. Chiang, W. P. Lin, C. H. Lin, W. S. Chen, F. T. Chen, C. H. Lien, and M.-J. Tsai, "Evidence and solution of over-reset problem for hfox based resistive memory with sub-ns switching speed and high endurance," in *2010 International Electron Devices Meeting*, 2010, pp. 19.7.1–19.7.4.

[13] J. Reuben and S. Pechmann, "Accelerated addition in resistive ram array using parallel-friendly majority gates," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 6, pp. 1108–1121, 2021.

[14] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2014.

[15] 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, 2014.

[16] L. Guckert and E. E. Swartzlander, "Optimized memristor-based multipliers," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 2, pp. 373–385, 2017.

[17] Z. Lu, M. T. Arafin, and G. Qu, "Rime: A scalable and energy-efficient processing-in-memory architecture for floating-point operations," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 120–125.

[18] G. Luo, S. Fu, and G. Gong, "Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 358–380, 2023.

[19] D. R. Gandhi and N. N. Shah, "Comparative analysis for hardware circuit architecture of wallace tree multiplier," in *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*, 2013, pp. 1–6.

[20] N. Lepri, M. Baldo, P. Mannocci, A. Glukhov, V. Milo, and D. Ielmini, "Modeling and compensation of ir drop in crosspoint accelerators of neural networks," *IEEE Transactions on Electron Devices*, vol. 69, no. 3, pp. 1575–1581, 2022.

[21] C. Rafferty, M. O'Neill, and N. Hanley, "Evaluation of large integer multiplication methods on hardware," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1369–1382, 2017.

[22] M. Bodrato, "Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0," in *WAIFI 2007 proceedings*, 06 2007, pp. 116–133.

[23] A. A. Karatsuba, "The complexity of computations," *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, vol. 211, pp. 169–183, 1995.

[24] M. Bodrato and A. Zanoni, "Integer and polynomial multiplication: towards optimal toom-cook matrices," in *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 17–24. [Online]. Available: https://doi.org/10.1145/1277548.1277552

[25] D. Kalman, "The generalized vandermonde matrix," *Mathematics Magazine*, vol. 57, no. 1, pp. 15–21, 1984.

[26] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 715–731. [Online]. Available: https://doi.org/10.1145/3297858.3304049

[27] P. Chakali and M. K. Patnala, "Design of high speed kogge-stone based carry select adder," *International Journal of Emerging Science and Engineering (IJESE)*, vol. 1, no. 4, pp. 2319–6378, 2013.

[28] K. Vitoroulis, "Parallel prefix adders," Online (accessed on Sept. 5th, 2024), 2006, https://www.slideshare.net/peeyushpashine/parallel-prefix-adders-presentation.

[29] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comp.*, vol. 44, no. 170, pp. 519–521, 1985.

[30] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[31] W. Tan, B. M. Case, A. Wang, S. Gao, and Y. Lao, "High-speed modular multiplier for lattice-based cryptosystems," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 8, pp. 2927–2931, 2021.