IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 May 2023, accepted 31 May 2023, date of publication 5 June 2023, date of current version 8 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3282646

RESEARCH ARTICLE

FlexKA: A Flexible Karatsuba Multiplier Hardware Architecture for Variable-Sized Large Integers

BYEONGMIN KANG AND HYUNGMIN CHO
Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Hyungmin Cho (hyungmin.cho@skku.edu)

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grants funded by the Korea Government Ministry of Science and ICT (MSIT) Graduate School of Convergence Security (No. 2022-0-01199) and Development of high speed encryption data processing technology that guarantees privacy based hardware (No. 2021-0-00779). The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

ABSTRACT The Karatsuba algorithm is an effective way to accelerate large integer multiplications through recursive function calls. However, existing hardware implementations of Karatsuba multipliers are limited to fixed operand sizes. To enable their application in diverse domains, including homomorphic encryption with varying multiplicative depths, it is necessary to support variable operand sizes. In this paper, we propose a novel Karatsuba multiplier design, named FlexKA, which supports variable operand sizes through a state machine that manages the dynamic call states of the operation. We evaluate FlexKA on the Xilinx ZynqMP FPGA and demonstrate that it supports variable operand sizes up to 256K bits, achieving a 9.2× speedup compared to a highly-optimized software library running on a CPU. Our results show that FlexKA is an efficient and effective solution for large integer multiplications with flexible operand sizes in hardware.

INDEX TERMS Multiplying circuits, field programmable gate arrays.

I. INTRODUCTION

Large integer multiplication is a fundamental operation in numerous fields including coding theory, digital signal processing, and cryptography. Homomorphic Encryption (HE) enables computation on encrypted data [1]. However, ensuring the necessary multiplicative depth while maintaining the required security level in HE requires increasing operand size [2]. Therefore, it is crucial for the underlying computing system for HE to support adjustable operand sizes. This enables an optimized HE system that can support a wide range of HE applications with varying computational demands while minimizing computational costs. Simply increasing the default, fixed operand size can result in significant overhead, making it impractical for many applications.

While some HE algorithms may not require hardware-based large integer multiplications due to the use of the residue number system (RNS) to break down large integer operations into smaller ones [3], additional operations such as RNS conversions and key-switching operations are necessary for RNS-based HE algorithms. These additional operations can lead to an overall increase in execution time.

Selecting a suitable multiplication algorithm is crucial in reducing the overhead of large integer multiplications. Direct integer multiplication, also known as *Schoolbook* multiplication, has a quadratic increase in computational overhead as the input size grows. Therefore, multiplication algorithms optimized for large operands are essential. Examples of such algorithms include Karatsuba multiplication [4], [5], Toom multiplication [6], and NTT-based multiplications [7]. These algorithms provide much lower asymptotic computational complexity, which can significantly reduce the overhead associated with large integer multiplications.

While implementing large-integer multiplication algorithms on hardware can increase computation performance compared to running the algorithms in software, algorithms with better asymptotic computational complexity may not necessarily be the best candidates for hardware implementation. For example, the Toom and NTT multiplications have lower asymptotic computational complexity than Karatsuba multiplication, but they require much higher hardware overhead [8], [9]. Therefore, Karatsuba multiplication may be a better choice for moderate to large operands, while Toom and

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

NTT multiplications may only be suitable for extremely large operands.

Although the Karatsuba algorithm may be an appropriate choice for large-integer multiplications, there are challenges to implementing the Karatsuba multiplier in hardware, especially if we want to support flexible operand sizes. The main challenge is that the Karatsuba algorithm has a recursive structure that depends on the operand size. The Karatsuba multiplier reduces multiplication overhead by recursively calling itself with smaller operands, and the partial multiplication results are combined to obtain the final result.

High-level synthesis (HLS) tools, which are commonly used for hardware design, have limited support for recursive functions [10], [11]. Typically, only tail-recursive calls or recursive calls with a fixed call depth are supported. Therefore, implementing a Karatsuba multiplier with flexible operand sizes using HLS is not straightforward. While there are techniques to unroll the recursion and implement Karatsuba multiplication using HLS, these approaches may not be efficient for all use cases. To solve the reported issue, an approach in literature uses a variadic template to implement the recursion at compile time [12], with a similar Karatsuba-Comba approach as FlexKA. However, in this approach, once the compilation is completed, the multiplier module becomes fixed to the compiled input size.

There are several hardware implementations of the Karatsuba multiplier, but those designs are often fixed to a specific input operand size [9], [12], [13], [14], [15], [16]. San and At presented a design that supports variable input sizes, but for small operands only [17]. Wong et al. proposed an RLWE Karatsuba multiplier, but the call depth is limited to one [18].

This paper presents FlexKA, a flexible Karatsuba multiplier architecture that supports variable input sizes. FlexKA can process multiplications of two arbitrary-sized integers, as long as the operand memory size permits. The key ideas of FlexKA are to use a state machine to control the recursively invoked Karatsuba multiplications and manage the locations of temporary data items in a fixed set of on-chip memory during computation. We implement FlexKA[1] using System Verilog HDL and evaluate its performance on a Xilinx ZynqMP FPGA using operand sizes up to 256K bits. On average, FlexKA achieved a 9.2× speedup compared to a highly-optimized software library. Our results demonstrate that FlexKA is an effective and efficient solution for implementing a flexible Karatsuba multiplier in hardware.

II. PRELIMINARIES

A. KARATSUBA MULTIPLICATION ALGORITHM

The Karatsuba multiplication algorithm reduces multiplication overhead by dividing the input operands into smaller slices and recursively calling the Karatsuba multiplication on these slices. The resulting partial products are combined using a series of additions and subtractions.

The source code of FlexKA is publicly available at the following repository: https://github.com/hyungmin2/FlexKA

Algorithm 1 The Karatsuba Multiplication

Input: Integers $A[n_A\text{-}1{:}0]$ and $B[n_B\text{-}1{:}0]$
Output: Integer $AB[n_A + n_B - 1 : 0] \leftarrow A \times B$

1 **if** $n_A \le N_{th}$ *and* $n_B \le N_{th}$ **then**
2 **return** $A \times B$ ▷ Direct multiplication
3 $m \leftarrow \lfloor min(n_A, n_B)/2 \rfloor$ ▷ Midpoint
4 $A_l \leftarrow A[m-1:0]$ ▷ Lower part
5 $B_l \leftarrow B[m-1:0]$
6 $A_h \leftarrow A[n_A-1:m]$ ▷ Upper part
7 $B_h \leftarrow B[n_B-1:m]$
8 $AB_l \leftarrow Karatsuba(A_l, B_l)$ ▷ L-call
9 $AB_h \leftarrow Karatsuba(A_h, B_h)$ ▷ H-call
10 $A_{hl} \leftarrow A_h + A_l$ ▷ Operand merging
11 $B_{hl} \leftarrow B_h + B_l$
12 $AB_{hl} \leftarrow Karatsuba(A_{hl}, B_{hl})$ ▷ HL-call
13 $AB \leftarrow AB_h \times 2^{2m} + (AB_{hl} - AB_h - AB_l) \times 2^m + AB_l$ ▷ Partial product combination
14 **return** AB

Algorithm 1 shows the Karatsuba multiplication algorithm, which takes input operands A and B, where n_A and n_B represent their bit-widths. If n_A and n_B are less than a threshold value N_{th}, direct multiplication is used to compute $A \times B$. Otherwise, the input operands are split in half at the midpoint m (e.g., into A_h and A_l).

The partial products AB_h, AB_l, and AB_{hl} are obtained through three recursive Karatsuba calls. AB_l is calculated by multiplying the lower parts of A and B, and we refer to the recursive call that calculates AB_l as the *L-call*. Similarly, the recursive call that computes AB_h using the upper parts of A and B is referred to as the *H-call*.

To calculate AB_{hl}, we first need to create temporary values A_{hl} and B_{hl}, which are obtained by adding the upper and lower parts of each operand (*operand merging*). The *HL-call* then uses these temporary values to compute AB_{hl}.

Using the three partially-computed products, the output AB is obtained by combining the partial products at their respective digit positions (*partial product combination*).

By utilizing the Karatsuba algorithm, the cost of multiplication can be reduced from the $O(n^2)$ required for direct multiplication to $O(n^{log_2 3})$. However, it is important to note that in addition to the multiplication cost, the operand merging and partial product combination steps also contribute to the overall computation time. These steps involve performing additions at every depth of the Karatsuba call, further impacting the total execution time.

B. SIZE OF THE PARTIAL SUMS

Let $|X|$ denote the required bit-width of value X. In cases where $|A|$ and $|B|$ are not identical or are not even numbers, the midpoint, denoted as m, does not divide the input value into two equal-sized chunks. Even when the original input size given to the multiplier is an even number, unless both $|A|$

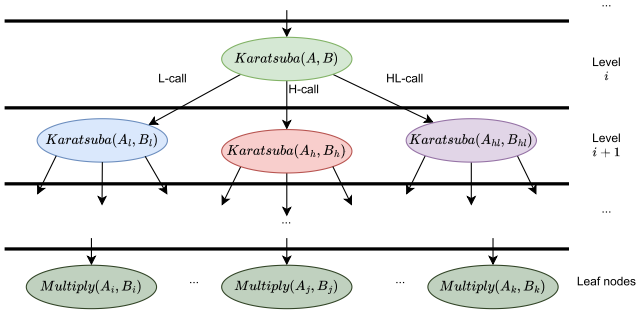


FIGURE 1. Recursive Call Tree of the Karatsuba Multiplication.

and $|B|$ are powers of two, the Karasuba algorithm encounters odd values for $|A|$ or $|B|$ during the recursive call. In such cases, without loss of generality, we consider the upper-bit part to have the bigger chunk, meaning $|A_h| \geq |A_l|$ and $|B_h| \geq |B_l|$. Consequently, $|AB_h|$ is always larger than or equal to $|AB_l|$.

Let's consider AB_{hl}, which is obtained by multiplying A_{hl} and B_{hl}. Due to the possible carry bit generated while computing $A_{hl} = A_h + A_l$, $|A_{hl}|$ can be larger than $|A_h|$. The same applies to B_{hl} as well. Therefore, we have $|AB_{hl}| = |A_{hl}| + |B_{hl}| \geq |A_h| + |B_h| = |AB_h|$

To summarize, we have $|AB_{hl}| \geq |AB_h| \geq |AB_l|$, and it is crucial for a Karatsuba implementation to handle these non-equal partial sum sizes correctly in order to support arbitrary input sizes.

III. FlexKA ARCHITECTURE DESIGN

FlexKA is designed to implement the recursive Karatsuba function calls on fixed hardware architecture. As shown in Fig. 1, the Karatsuba recursive calls form a call tree. The recursive calls form a call tree, with intermediate call nodes responsible for calling lower-level call nodes (Lines 8, 9, and 12 of Algorithm 1), computing operand merging (Lines 10 and 11 of Algorithm 1), and partial product combination (Line 13 of Algorithm 1). The leaf call nodes perform direct multiplications (Line 2 of Algorithm 1).

Figure 2 shows the overall architecture of FlexKA. To accommodate arbitrary recursive call depth on fixed hardware, FlexKA employs the following strategies:

- The Karatsuba algorithm traverses the recursive call tree in Fig. 1 in a depth-first order. FlexKA follows the same depth-first order.
- The main finite state machine (FSM) in the controller module manages the recursive call invocations and returns. It also controls the sub-modules for each operation to dispatch the sub-operations within a call node in the order dictated by the Karatsuba algorithm.
- FlexKA has several sub-modules responsible for performing each sub-operation within a call node. These sub-modules include operand merging, partial product combination, and direct multiplication
- While traversing the call tree, the node parameters utilized at the current call node are stored in the stack memory (SM) and subsequently restored once the lower-level node returns.
- Operands and intermediate values are stored in a fixed set of memory modules `Mem_A`, `Mem_B`, and `Mem_C`). The same memory modules are used regardless of the call depth.
- At a leaf node, the base multiplier module performs direct multiplication for operands with a size not larger than N_{th}.

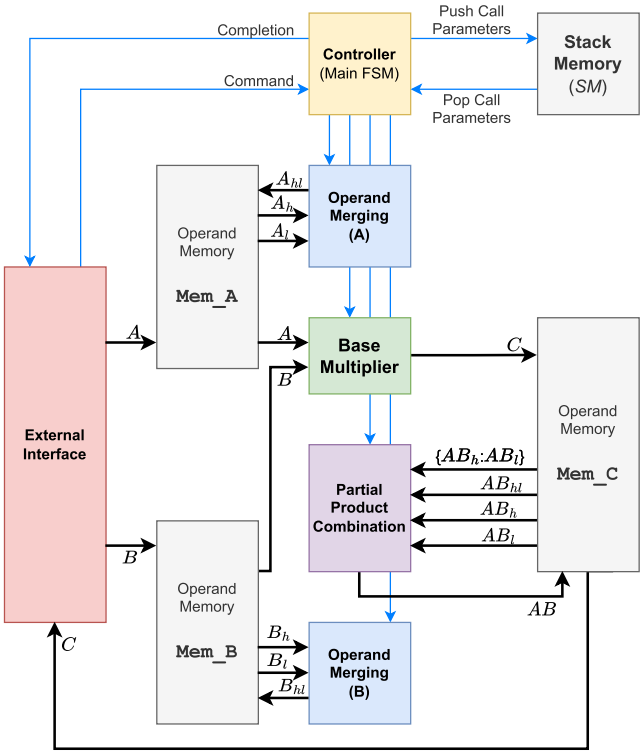


FIGURE 2. FlexKA architecture diagram.

A. EXTERNAL INTERFACE

Figure 2 also illustrates the interface to FlexKA. Through this interface, the host module (such as a CPU) can initiate computations using FlexKA. The process involves several steps. First, the size of the input operands is provided to the FlexKA controller module via the command interface. Next, the input operands A and B are transferred to FlexKA and stored in the operand memory modules. Computation can commence once the operands are transferred. The interface also allows the host to receive the completion status and obtain the computed results after the computation has concluded.

The FlexKA interface can be connected to the host using various interface protocols. In our implementation, the interface is connected to the host CPU via the AXI interface. The input operands and results are transferred with a granularity of 64 bits per word.

B. STATE TRANSITION

Figure 3 illustrates the state transition of the main FSM in the FlexKA controller. Although there are a finite number of

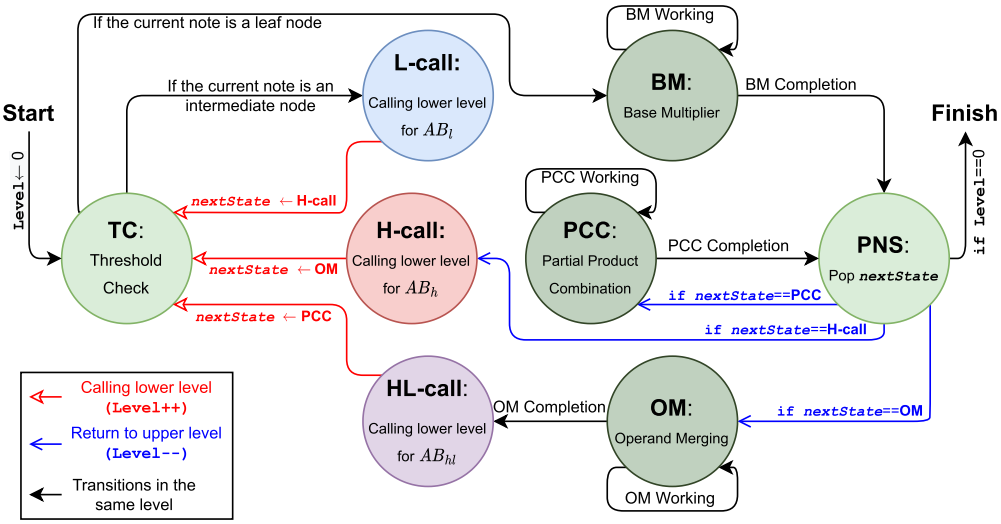


FIGURE 3. State transition diagram of the main FSM in the FlexKA controller module.

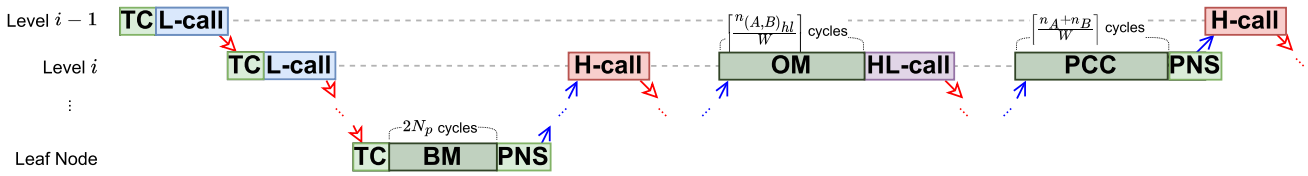


FIGURE 4. FlexKA main FSM state transition example.

states, the FSM can represent an arbitrary depth of Karatsuba calls using the additional `Level` value. This value increases when FlexKA descends to a lower-level node and decreases upon returning to the upper level. The `Level` value is also utilized to index the stack memory module (`SM`), which enables the memory pointers to be pushed and popped.

The **`TC`** state is the initial state of a Karatsuba call and determines whether the current node is a leaf or an intermediate node. If the node is a leaf node, it transitions to the **`BM`** state, which performs direct multiplication using the base multiplier module. Otherwise, if the node is an intermediate node, FlexKA initiates lower-level call nodes. To accomplish this, FlexKA saves the current call parameters and transitions to one of the three states: **`L-call`**, **`H-call`**, or **`HL-call`**. These states prepare the new call parameters for the lower-level node.

Once FlexKA has computed the new parameters, it transitions back to the **`TC`** state to initiate a new call node (indicated by the red arrows in Fig. 3). When descending to a lower-level call node, FlexKA must also specify the subsequent state to be executed upon returning from the lower-level node, which is referred to as the `nextState`.

The value of `nextState` depends on the type of call being executed. For example, after an L-call returns, FlexKA transitions to the **`H-call`** state. Once the H-call returns, FlexKA transitions to the **`OM`** state, which executes the operand merging. Next, the HL-call state is initiated, followed by the **`PCC`** state. The **`PCC`** state is responsible for handling the partial product combination.

The **`PNS`** state serves as the concluding state of a call node and manages the return to the upper-level node. In the **`PNS`** state, `nextState` is removed from SM and utilized to transition to the corresponding state while simultaneously decrementing `Level` (as indicated by the blue arrows in Fig. 3). Additionally, the call parameters saved for the upper-level nodes are retrieved from `SM` and restored.

Figure 4 depicts the state transition timeline at a leaf node and at call depth i. The **`TC`**, **`L-call`**, **`H-call`**, **`HL-call`**, and **`PNS`** states manage the call and return procedures, each immediately transitioning to the subsequent state. On the other hand, the **`BM`**, **`OM`**, and **`PCC`** states handle the actual computation steps, and Fig. 4 denotes the number of computation cycles elapsed in these states.

C. OPERAND MEMORY

When implementing a Karatsuba multiplier in hardware, replicating the memory usage pattern of software increases the design overhead. This is because various data objects are temporarily created during the computation, such as those for A_{hl} and B_{hl} in Lines 10 and 11 of Algorithm 1. Similarly, several intermediate data objects are required to perform the partial product combination step. This approach not only increases the design complexity but also limits the possible

recursive call depth. Therefore, an HLS-based Karatsuba implementation, which automatically generates corresponding hardware from software, usually results in high memory usage and is fixed to a certain input operand size. For instance, an HLS-based implementation of the Karatsuba algorithm has shown that the BRAM usage increases almost quadratically as the input operand size grows [14].

Instead, FlexKA uses only three fixed memory modules to handle the entire recursive call process of an arbitrary-sized multiplication. Operand memory `Mem_A` and `Mem_B` contain the input operands, and the multiplication results will be written into operand memory `Mem_C`. Additionally, `Mem_A`, `Mem_B`, and `Mem_C` may temporarily hold intermediate values. The size of the operand memory can be increased if more on-chip memory resources are available to support larger operand sizes.

D. MEMORY POINTERS

The data objects in the operand memory are located by the following memory pointers.

- i_A, i_B, **and** i_C: The starting address of A, B, and C in `Mem_A`, `Mem_B`, and `Mem_C` at the current call node of the Karatsuba multiplication, respectively.
- t_A **and** t_B: The starting address for temporarily storing A_{hl} and B_{hl} in `Mem_A` and `Mem_B`, respectively.

Figure 5 shows how the data objects are placed in the operand memory at an arbitrary intermediate call level. At the beginning of the current call node, the input operands A and B are located from the current memory pointer i_A and i_B. At the end of the current call node, before returning to the upper node, the goal of the current node is to complete the $A \times B$ computation and place the resulting value AB at memory pointer i_c.

In order to compute the final output AB, three recursive calls are required to calculate AB_l, AB_h, and AB_{hl}, respectively. These values are temporarily stored in `Mem_C` before being merged into the final output, which is also stored in `Mem_C`. The use of the same memory module to store both the temporary partial products and the output is due to the fact that these partial products are also considered outputs from the perspective of lower-level call nodes. In other words, FlexKA writes the output to `Mem_C` regardless of the call level, simplifying the design by eliminating the need for a separate memory module for writing the output based on the call level.

The L-call, which calculates AB_l, utilizes A_l and B_l as its operands. As A_l and B_l correspond to the lower parts of A and B, respectively, the memory pointers for locating these input operands remain the same as those for the current call node, denoted by i_A and i_B. The resulting output of the L-call, AB_l, is temporarily placed at the address $i_C + m$ to prevent conflicts between the partial products and the final output AB. Further discussion regarding this offset value can be found in Section III-G.

The H-call takes A_h and B_h as its operands, which can be located by adding an offset of m from i_A and i_B, respectively. The output of the H-call, AB_h, should be placed in a non-overlapping position with the AB_l. Therefore, the H-call uses the output position of $i_C + 3m$, since the size of AB_l is $2m$

Before the HL-call, temporary values A_{hl} and B_{hl} have to be calculated. Inside the HL-call at the lower level, A_{hl} and B_{hl} can be seen as normal A and B operands. Therefore, we place A_{hl} and B_{hl} in operand memory `Mem_A` and `Mem_B`, respectively. By doing so, we can just change the memory pointer i_A and i_B to the locations of A_{hl} and B_{hl} when calling the HL-call. With the updated i_A and i_B pointers, the lower-level call node can compute AB_{hl} using the same logic as other calls.

To avoid overwriting the input operands in `Mem_A` and `Mem_B`, the temporary locations of A_{hl} and B_{hl} should be determined in a way that they do not overlap with the input operands. At the root node, t_A and t_B are set to n_A and n_B, respectively, to avoid overlap with the original input operands A and B. For the L-call and H-call, t_A and t_B do not need to be changed since these pointers are already positioned in a location that avoids collision with the input operands for those calls. In the case of the HL-call, relocation of t_A and t_B is necessary, as this call utilizes the temporary A_{hl} and B_{hl} values (which are currently stored at t_A and t_B, respectively) as its input operands. Consequently, t_A and t_B must be adjusted to $t_A + n_{A_{hl}}$ and $t_B + n_{B_{hl}}$ for the HL-call, where $n_{A_{hl}}$ and $n_{B_{hl}}$ correspond to the sizes of A_{hl} and B_{hl}, respectively.

The output of the HL-call, AB_{hl} also needs to avoid collision with the previously produced AB_l and AB_h. Therefore, the memory pointer i_C for the HL-call is set to $i_C + n_A + n_B + m$, since the combined size of AB_h and AB_l is $n_A + nB$.

Each call node is responsible for writing its output within a designated range of `Mem_C`. Because the partial products and the output of the nodes share the same memory module, there is a risk of overwriting the output of another call node when computing the output of a node. This is because the combined size of the partial products temporarily occupies more space than the output range of the node.

To avoid this conflict, we only allow each call node to write to the left of the memory pointer i_C. Additionally, we ensure that lower-level call nodes are called from the L-call node first, which will write its output at the right-most part of `Mem_C`. When the H-call and its child nodes write their outputs above their pointer location, they will not overwrite the output of the L-call since it is located below the output pointer of the H-call. The same principle applies to the HL-call.

E. STACK MEMORY FOR THE CALL PARAMETERS

In order for FlexKA to process an intermediate call node, five memory pointers (as described in the previous subsection) and the sizes of the two input operands are required. Table 1 summarizes the rules for Karatsuba call parameters and memory pointers for each of the three lower-level calls.

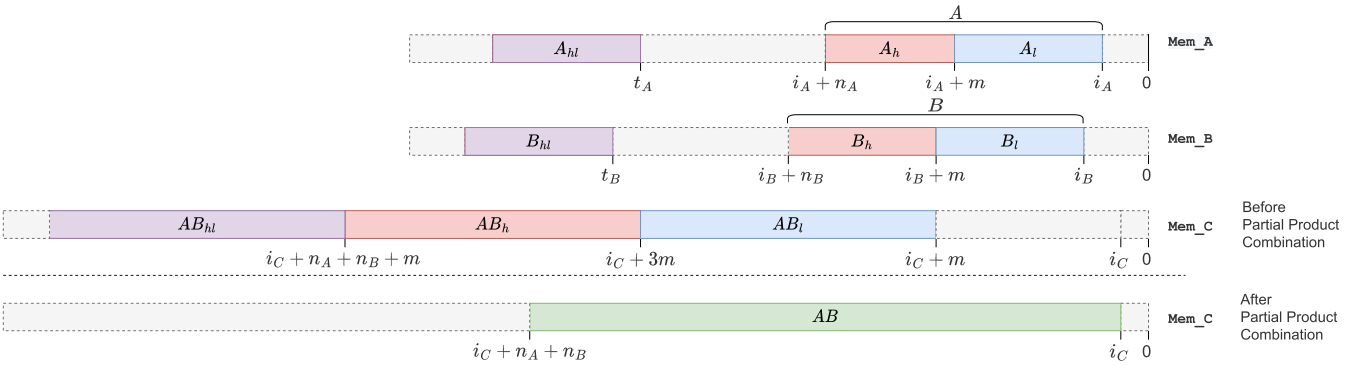


FIGURE 5. Operands and temporary values stored in Mem_A, Mem_B, and Mem_C during the computation.

TABLE 1. Karatsuba call parameter change rule for each lower-level node.

Parameter	Level i	Level $i+1$		
		L-call	H-call	HL-call
Size of operand A	n_A	$m = \lfloor min(n_A, n_B)/2 \rfloor$	$n_A - m$	$n_{A_{hl}} = n_A - m + carry$
Size of operand B	n_B	$m = \lfloor min(n_A, n_B)/2 \rfloor$	$n_B - m$	$n_{B_{hl}} = n_B - m + carry$
Starting address of input parameter A	i_A	i_A	$i_A + m$	t_A
Starting address of input parameter B	i_B	i_B	$i_B + m$	t_B
Starting address of output value C	i_C	$i_C + m$	$i_C + 3m$	$i_C + n_A + n_B + m$
Starting address of temporary value A_{hl}	t_A	t_A	t_A	$t_A + n_{A_{hl}}$
Starting address of temporary value B_{hl}	t_B	t_B	t_B	$t_B + n_{B_{hl}}$

Upon returning to the upper-level call node, it's necessary to restore the memory pointers that were altered during the lower-level node computations. To accomplish this, FlexKA utilizes a stack memory module (SM) that stores these memory pointers. Specifically, the memory pointers are pushed to SM when transitioning to a lower-level call node and subsequently popped from SM when the lower-level operation has been completed. Additionally, SM maintains the next state of the FSM, which is necessary to handle call returns (as discussed in Section III-B).

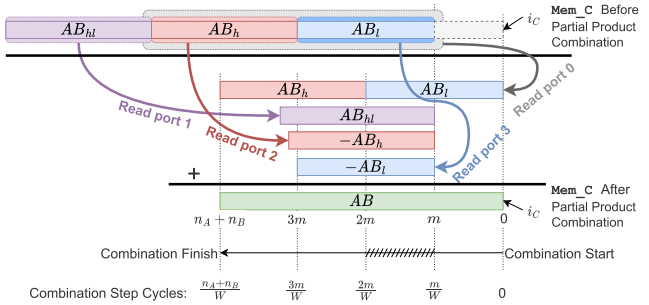


FIGURE 6. Mem_C access during the partial product combination step.

F. OPERAND MERGING

The temporary values A_{hl} and B_{hl} are obtained by merging the lower part and the upper parts of the input operands. The following explanations on operand merging focus on operand A, but the same concepts and methods apply to operand B as well.

The sizes of A_h and A_l are $n_A - m$ bits and m bits, respectively. Since $n_A - m \geq m$ (Line 3 of Algorithm 1), the size of $A_h + A_l$ is $n_{A_{hl}} = n_A - m + carry$ bits. Here, $carry$ is the possible carry bit generated from the addition.

To handle the addition, FlexKA uses adders with a fixed width of W bits, which allows the addition to be performed in multiple cycles. The number of cycles required for addition is $\left\lceil \frac{n_{A_{hl}}}{W} \right\rceil$. In FlexKA, W is set to 64 bits to match the width of the UltraRAM modules in ZynqMP. To minimize the number of cycles required for operand merging, FlexKA simultaneously performs operand merging for A and B using two adders (Adder A and B in Fig. 2).

G. COMBINING PARTIAL MULTIPLICATION RESULTS

Figure 6 illustrates the memory access pattern during the partial product combination step. As discussed in Sec. II-B, the sizes of the partial sums, AB_{hl}, AB_h and AB_l, may differ. In FlexKA, the partial product combination step is specifically designed to accommodate these differences. To minimize the number of cycles required for this step, Mem_C provides four read ports.

When reading from port 0, the values of AB_h and AB_l represent $AB_h \times 2^{2m} + AB_l$. AB_h and AB_l are reused to represent $(AB_{hl} - AB_h - AB_l) \times 2^m$, and this time they are read by read ports 2 and 3, respectively. Similar to the operand merging step, the combination step employs an adder that can handle W-bit inputs per cycle.

The combination process starts from the lower bits of AB and progresses to the upper bits. Overall, the combination step takes $\lceil (n_A+n_B)/W \rceil$ cycles per level.

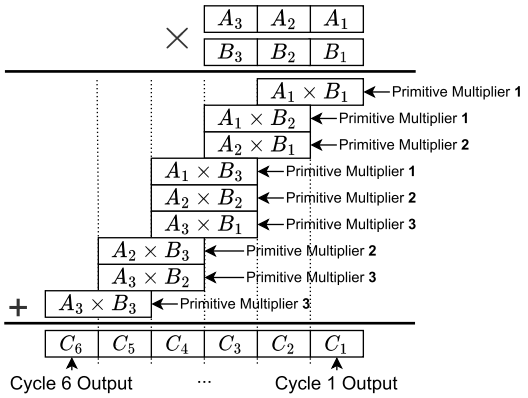


FIGURE 7. Base multiplier computation timeline when $N_{th} = 3W$.

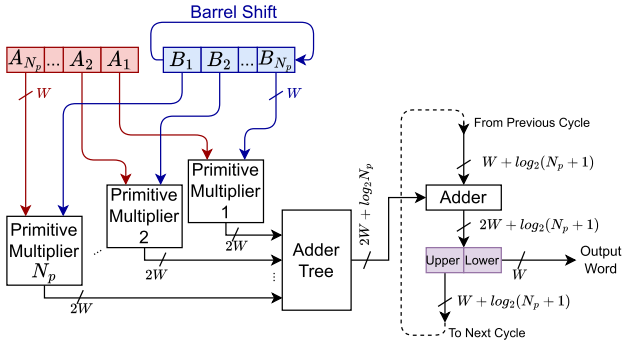


FIGURE 8. Base multiplier architecture with N_p primitive multipliers.

As discussed in Section III-D, the combination step uses the same operand memory to read the partial products and write the combined output. To prevent conflicts during this step, FlexKA places the partial products at a slightly shifted position from the base address of the output. Specifically, the partial products, starting from AB_l, are placed m-bits away from the base location of AB indicated by the current value of i_C.

Without such an offset, the combination step would overwrite some of the partial products before they are fully used. In particular, conflicts would arise for AB_l during the combination cycles from $\frac{m}{W}$ to $\frac{2m}{W} - 1$ (indicated by the hatched pattern in Fig. 6). During this period, FlexKA produces a part of AB, namely $AB[2m-1:m]$, which should be written into the range of $[i_C + 2m - 1 : i_C + m]$ in `Mem_C`.

If we had not added an offset when computing AB_l, the upper m-bits of AB_l would have been overwritten with $AB[2m-1:m]$ during this period. However, the upper part of AB_l is needed again during the combination cycles from $\frac{2m}{W}$ to $\frac{3m}{W} - 1$ to compute $AB[3m-1:2m]$.

As shown in Fig. 6, the added offset enables us to avoid the conflict during the period from $\frac{m}{W}$ to $\frac{2m}{W} - 1$. The partial product that gets overwritten during this period is the lower m-bit part of AB_l, which is no longer used afterward.

H. BASE MULTIPLIER

As the Karatsuba algorithm progresses to lower levels in its call tree, the size of the operands decreases. However, at some point, the cost of managing the call tree can become greater than the benefit gained from smaller operands. In FlexKA, the base multiplier performs direct multiplication when the operand size is less than the threshold N_{th}.

To minimize the computation cycles required for direct multiplication and provide flexibility in increasing N_{th}, the base multiplier has been designed with multiple primitive multipliers. These are pipelined integer multipliers that each accept two W-bit inputs. There are a total of $N_p = \frac{N_{th}}{W}$ primitive multipliers in the base multiplier.

The base multiplier is based on the principle of Comba multiplication [19]. Figure 7 shows the partial products computed during Comba multiplication when $N_p = 3$. The input operands A and B are divided into W-bit words, A_1-A_3 and B_1-B_3. Unlike the original Comba multiplication, which computes one partial product at a time, the base multiplier simultaneously calculates all partial products of the same digit position using N_p primitive multipliers. These partial products are then combined, resulting in one W-bit result word at each cycle. Using this method, the base multiplier performs the multiplication of two N_{th}-bit values in $2N_p$ cycles.

The architecture of the base multiplier is illustrated in Figure 8. The operand A words are directly connected to each of the primitive multipliers. For example, in Figure 7, word A_1 is always connected to the primitive multiplier 1. On the other hand, the operand B words are connected in reverse order and are barrel shifted every cycle to be multiplied with the corresponding word from operand A. The shifted operand B words at cycle N_p are shown in Figure 8.

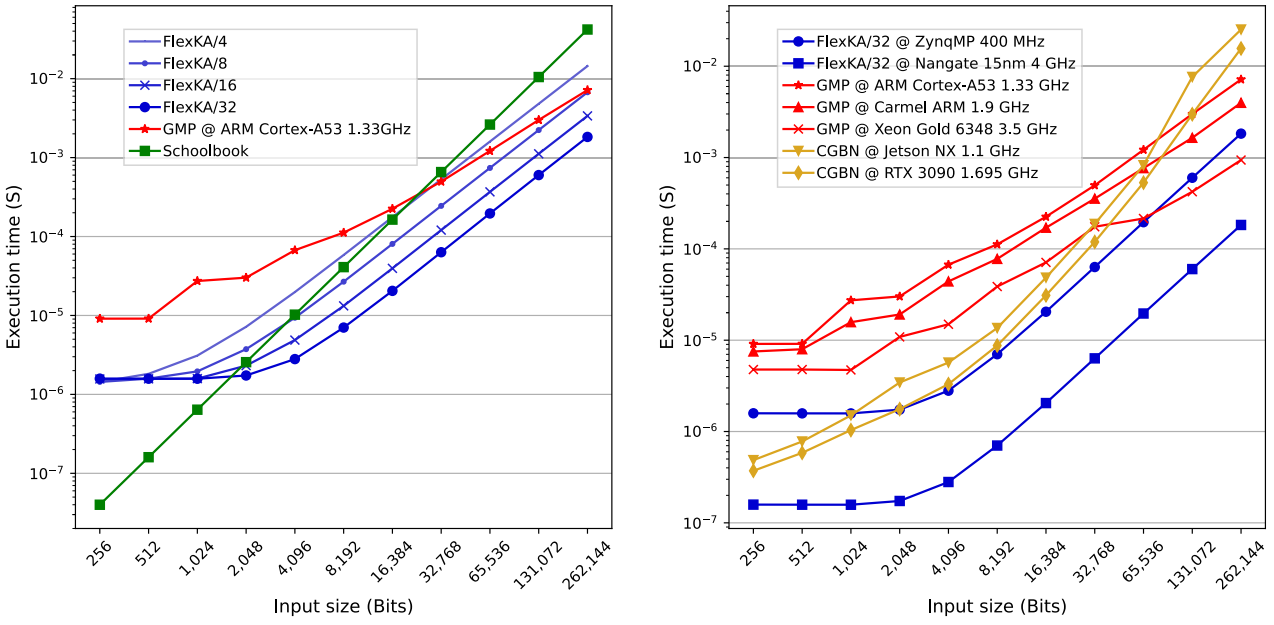
The computed partial products are combined using an adder tree. Since the digit positions of the partial products overlap with those of the adjacent cycles, the combined partial products are added with the carry from the previous cycle to produce the W-bit output word of the current cycle. The upper part of the added result, excluding the W-bit output from the lower part, is carried over to the next cycle.

IV. EVALUATION

We compare the large-integer multiplication performance of FlexKA against the optimized software implementations of multiplication algorithms on CPU to demonstrate the performance advantages of dedicated multiplication hardware.

Additionally, we compare the resource utilization of FlexKA with existing Karatsuba multiplier implementations on FPGA. Our results show that FlexKA provides better resource utilization compared to existing implementations while also offering flexibility in input sizes.

We evaluate FlexKA by implementing the multiplier architecture using the Xilinx Zynq UltraScale+ MPSoC XCK26 FPGA. During our evaluation of FlexKA, we assessed its performance while varying the base multiplier threshold sizes (N_{th}). Specifically, we denote a FlexKA



(a) Comparison of FlexKA on FPGA

(b) Comparison of FlexKA with CPUs and GPUs

FIGURE 9. Computation time comparison.

configuration with $N_{th} = nW$ as FlexKA/n. Notably, FlexKA operates at 400 MHz for all N_{th}.

We also performed synthesis of FlexKA using the Synopsys Design Compiler to project its performance in ASIC implementations. By utilizing the Nangate 15nm open-cell library [20], FlexKA achieved a maximum frequency of 4 GHz.

We evaluate FlexKA up to 256K bits since the performance benefit of the Karatsuba multiplier diminishes for larger sizes. However, 256K is not a hard limit for FlexKA. We can expand the size range by increasing the operand memory.

We compare the performance of FlexKA with the following implementations for large-integer multiplication.

- **GMP:** Multiplication is performed using the GNU Multiple Precision Arithmetic Library [21], which serves as the foundation for numerous number theory libraries, such as FLINT [22] or NTL [23].
- **CGBN:** Multiplication is performed using the NVIDIA CGBN[2], which is a CUDA-based library designed to accelerate multi-precision arithmetic on GPUs.
- **Schoolbook:** Multiplication is performed using the traditional schoolbook multiplier implemented on FPGA running at 400 MHz, which is the same frequency as FlexKA. The schoolbook multiplier performs a W-bit $\times$ W-bit multiplication per cycle and completes a multiplication in $\frac{N}{W}^2$ cycles.

The computation performance is measured using the wall-clock time, which is the duration from the invocation of the multiplication process until its completion. To accurately measure the computation time, we employ different methodologies depending on the platform.

- For the GMP library, we utilize the POSIX `clock_gettime` API to measure the computation time. This API allows us to capture the elapsed time accurately.
- In the case of CGBN, the CUDA `clock` API is used to measure the computation time. This API enables us to measure the elapsed GPU clock cycles within the GPU kernel, providing precise timing information.
- For the FPGA implementations, we measure the number of FPGA clock cycles during the computation.

We focus on measuring the pure computation time without including the data loading or unloading time. The reason for excluding these times is that they can be influenced by other factors such as I/O bandwidth, which are not directly related to the performance of the multiplier itself.

A. COMPUTATION PERFORMANCE ON FPGA

In Fig 9a, we compare the performance of FlexKA on the ZynqMP FPGA. In general, for small integers with less than 1,024 bits, the schoolbook multiplier provides the most efficient performance (i.e., the lowest execution time), whereas Karatsuba or Toom multiplications perform better for larger input sizes. The performance of FlexKA is significantly influenced by the chosen N_{th} configuration. For instance, FlexKA/4 is slower than both the schoolbook multiplier and GMP across the evaluated input size range. However, larger N_{th} configurations result in substantial performance improvements. For example, FlexKA/32 is 23$\times$ faster than GMP when the input size is 4,096 bits and 4$\times$ faster when the

[2] https://github.com/NVlabs/CGBN

TABLE 2. Resource Usage Comparison of FPGA-based Karatsuba Multiplier implementations.

Karatsuba multiplier	FPGA	Input Size (bits)	Clock Freq. (MHz)	BRAM	DSP	Flip-flop	LUT
Foster et al. [14]	UltraScale VU190	4,096	200	869	1,296	171K	139K
		8,192		1,398	1,296	386K	268K
		16,384		1,852	1,296	322K	225K
		32,768		2,268	1,296	408K	279K
		65,536		2,721	1,296	472K	322K
		131,072		8,429	1,296	558K	375K
		262,144		21,772	1,296	644K	418K
Zoni et al. [16]	Artix-7 A200T	8,467	143	160	-	13K	63K
		9,643		160		13K	63K
		14,717		160		13K	63K
		15,013		160		13K	63K
		17,827		160		13K	63K
		22,853		160		13K	63K
		24,533		160		13K	63K
		28,477		160		13K	63K
		37,619		160		13K	64K
Rafferty et al. [9]	Virtex-7 VX980T	128	351	-	13	4K	3K
		256	291		25	7K	7K
		512	236		49	17K	11K
		1024	160		97	34K	22K
		2048	96		193	67K	36K
		4096	50		385	135K	82K
Jayet-Griffon et al. [13]	Virtex-7 VX1140T	16,384	234	769	3,072	373K	323K
Andre [15]	Kintex-7 K325T	32	N/A	-	-	64	1.8K
		128				256	18K
		512				1024	179K
San and At [17]	Virtex-5 LX50	32 - 128	372	1	1	-	124
FlexKA/4	ZynqMP K26	64 - 262,144	400	256	64	7K	5K
FlexKA/8					128	12K	8K
FlexKA/16					256	21K	15K
FlexKA/32					512	38K	26K
FlexKA/4	Kintex-7 K160T	64 - 262,144	320	256	64	8K	5K
FlexKA/8					128	12K	7K
FlexKA/16					256	21K	11K
FlexKA/32					512	38K	19K

input size is 256K bits. Across the assessed operand sizes, the geometric mean of the speedup over GMP is 9.2× when $N_{th} = 32W$.

As the input size grows, the performance gap between GMP and FlexKA diminishes. The primary reason for this is that GMP utilizes various multiplication algorithms based on the size of the input operands. The threshold for selecting the algorithm may differ depending on the CPU architecture. For example, on ARM Cortex-A53, GMP utilizes Karatsuba multiplication for input sizes ranging from 896 to 3,136 bits. Beyond that, GMP switches to Toom multiplication. For extremely large integers beyond 200K bits, GMP employs FFT multiplication. When handling input sizes greater than 200K bits, a multiplier architecture based on Toom or NTT may be preferable due to their better asymptotic complexity. Nonetheless, it should be noted that implementing these multiplication algorithms in hardware leads to significantly higher resource consumption [8], [9].

B. COMPUTATION PERFORMANCE ON ASIC

Figure 9b presents the performance comparison between FlexKA/32 on FPGA and ASIC with various CPU and GPU platforms. Even when compared to higher-performance CPUs and GPUs, FlexKA continues to exhibit its performance benefits. In addition to the Cortex-A53 CPU on ZynqMP, we also evaluated the performance of FlexKA on the Carmel ARM CPU in NVIDIA Jetson NX and the Xeon 6348 CPU. The performance of the CGBN library was measured on two different GPUs: the NVIDIA Jetson NX embedded device and the NVIDIA GeForce RTX 3090.

FlexKA on FPGA demonstrated comparable performance to high-performance CPUs and GPUs, even at a clock frequency of 400 MHz. When the performance of FlexKA is measured at the 4 GHz clock frequency, FlexKA outperforms other baseline architectures.

C. FPGA RESOURCE CONSUMPTION

Table 2 compares the FPGA resource utilization of various Karatsuba implementations published in the literature. In general, existing Karatsuba multipliers are designed for fixed input sizes, and the resulting resource usage increases as the input operand size grows. For instance, the HLS-based implementation in [14] exceeds the available FPGA resources for input sizes of 128K bits or greater on the UltraScale

TABLE 3. Karatsuba computation time comparison in microseconds.

Input size (bits)	8,467	14,717	22,853	28,477	37,619	2,048	4,096	16,384
Computation time (μs) of other implementations	Zoni et al. [16]					Rafferty et al. [9]		Jayet-Griffon et al. [13]
	6.06	12.12	24.24	30.3	48.48	1.68	5.78	5.21
Computation time (μs) of **FlexKA/32** on ZynqMP 400 Mhz	11.34	15.97	40.21	46.85	107.20	1.58	2.80	20.49
Computation time (μs) of **FlexKA/32** on Kintex-7 320 Mhz	14.18	19.96	50.26	58.56	134.00	1.98	3.50	25.61

VU190 device. The implementation presented by Rafferty et al. not only has increased resource utilization but also has a decreasing clock frequency as the input size grows [9]. The resource usage in [16] is nearly identical across input sizes, but this is mainly due to the similar operand sizes evaluated.

In contrast, FlexKA supports flexible input sizes while consuming only modest hardware resources and maintaining a high clock frequency. Depending on the N_{th} configuration, the resource utilization differs, but the supported operand size ranges remain identical.

The FlexKA/32 configuration, which is the largest among the evaluated setups, requires 256 BRAM units (or 32 URAM units) and 512 DSP Units. The number of utilized flip-flops and LUTs is less than 38K and 26K, respectively. When comparing these results with the implementation presented by Foster et al. [14], which evaluated a similar input range to FlexKA, the resource utilization of FlexKA is substantially lower.

We also conducted evaluations of FlexKA on the Xilinx Kintex-7 K160T FPGA device, which belongs to the same FPGA generation as the FPGAs utilized in [9], [13], [15], [16]. Although the maximum achievable frequency on the Kintex-7 FPGA is reduced to 320 MHz compared to 400 MHz on the ZynqMP platform, the resource usage remains relatively similar without significant differences.

Table 3 provides a comparison of the computation time between FlexKA and a subset of FPGA-based Karatsuba multiplier implementations listed in Table 2. Only the implementations that reported their computation time are included in Table 3. To ensure a fair comparison, we set the input size of FlexKA to be the same as the input sizes of the listed implementations. In some cases, FlexKA was up to 4 times slower than the existing methods. However, Table 3 shows that overall, FlexKA exhibits performance similar to other implementations that are fixed to a specific input size, despite the fact that FlexKA supports flexibility in input sizes.

V. CONCLUSION

Supporting variable input sizes is crucial for enabling the use of a Karatsuba multiplier in a wide range of application domains. In this paper, we introduced FlexKA, a Karatsuba multiplier design that supports variable input sizes on fixed hardware.

When implementing a large-integer multiplier module on an FPGA, previous approaches commonly use a Karatsuba multiplier that is fixed to a specific input size, provided that the implementation supports a parameterized configuration. This approach is feasible because FPGAs can be reprogrammed to accommodate different input size requirements. However, in the context of ASIC implementation, where the hardware is permanently fixed, the input size flexibility of FlexKA becomes even more valuable. FlexKA can maintain high performance for various input sizes using the same hardware, making it well-suited for a wide range of applications with different requirements. Thus, with FlexKA, it is possible to support diverse applications efficiently, utilizing the same hardware resources.

When compared to a highly-optimized software-based large integer multiplication library running on a CPU, FlexKA achieved more than a 9× speedup. The efficiency of FlexKA can be compared to existing Karatsuba multiplier implementations that are fixed to a certain input size. FlexKA consumes a similar or lower amount of FPGA resources compared to other Karatsuba multipliers that are fixed to a smaller size. Additionally, the computation performance of FlexKA falls within the similar range of other multipliers.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Theory Comput.*, May 2009, pp. 169–178.

[2] H. Chen, K. Laine, R. Player, and Y. Xia, "High-precision arithmetic in homomorphic encryption," in *Topics in Cryptology—CT-RSA*. Springer, 2018, pp. 116–136.

[3] J. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Proc. Int. Conf. Sel. Areas Cryptogr. (SAC)*, 2018, pp. 347–368.

[4] A. Karatsuba and Y. Ofman, "Multiplication of multi-digit numbers on automata," *Sov. Phys. Doklady*, vol. 7, pp. 595–596, Jun. 1963.

[5] Y. Li, S. Sharma, Y. Zhang, X. Ma, and C. Qi, "On the complexity of hybrid *n*-term Karatsuba multiplier for trinomials," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 3, pp. 852–865, Mar. 2020.

[6] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Math. Doklady*, vol. 3, no. 4, pp. 714–716, 1963.

[7] A. Schönhage and V. Strassen, "Schnelle multiplikation Großer Zahlen," *Computing*, vol. 7, nos. 3–4, pp. 281–292, Sep. 1971.

[8] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI design of a large-number multiplier for fully homomorphic encryption," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 9, pp. 1879–1887, Sep. 2014.

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

[10] (2021). *Intel High Level Synthesis Compiler Pro Edition: User Guide*. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683456/21-4/pro-edition-restrictions.html

[11] (2021). *Vivado Design Suite User Guide*. [Online]. Available: https://docs.xilinx.com/v/u/en-U.S./ug902-vivado-high-level-synthesis

[12] E. Vitali, D. Gadioli, F. Ferrandi, and G. Palermo, "Parametric throughput oriented large integer multipliers for high level synthesis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 38–41.
[13] C. Jayet-Griffon, M.-A. Cornelie, P. Maistri, P. Elbaz-Vincent, and R. Leveugle, "Polynomial multipliers for fully homomorphic encryption on FPGA," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2015, pp. 1–6.
[14] M. J. Foster, M. Lukowiak, and S. Radziszowski, "Flexible HLS-based implementation of the Karatsuba multiplier targeting homomorphic encryption schemes," in *Proc. 26th Int. Conf. Mixed Design Integr. Circuits Syst. (MIXDES)*, Jun. 2019, pp. 215–220.
[15] W. Andre, "Efficient adaptation of the Karatsuba algorithm for implementing on FPGA very large scale multipliers for cryptographic algorithms," *Int. J. Reconfigurable Embedded Syst. (IJRES)*, vol. 9, no. 3, p. 235, Nov. 2020.
[16] D. Zoni, A. Galimberti, and W. Fornaciari, "Flexible and scalable FPGA-oriented design of multipliers for large binary polynomials," *IEEE Access*, vol. 8, pp. 75809–75821, 2020.
[17] I. San and N. At, "On increasing the computational efficiency of long integer multiplication on FPGA," in *Proc. IEEE 11th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Jun. 2012, pp. 1149–1154.
[18] Z. Wong, D. C.-K. Wong, W. Lee, and K. Mok, "High-speed RLWE-oriented polynomial multiplier utilizing Karatsuba algorithm," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 6, pp. 2157–2161, Jun. 2021.
[19] P. G. Comba, "Exponentiation cryptosystems on the IBM PC," *IBM Syst. J.*, vol. 29, no. 4, pp. 526–538, 1990.
[20] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15 nm FreePDK technology," in *Proc. Symp. Int. Symp. Phys. Design*, Mar. 2015, pp. 171–178.
[21] (2021). *The GNU Multiple Precision Arithmetic Library*. [Online]. Available: https://gmplib.org/
[22] W. B. Hart, "Fast library for number theory: An introduction," in *Proc. 3rd Int. Congr. Math. Softw. (ICMS)*, 2010, pp. 88–91.
[23] (2021). *NTL: A Library for Doing Number Theory*. [Online]. Available: https://libntl.org/

BYEONGMIN KANG received the B.S. degree in information and communication engineering from Myongji University, South Korea, in 2017. He is currently pursuing the M.S. degree in computer science and engineering with Sungkyunkwan University (SKKU), South Korea. His research interest includes accelerator architectures in FPGA.

HYUNGMIN CHO received the B.S. degree in computer science engineering from Seoul National University, South Korea, in 2005, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, in 2010 and 2015, respectively. He was a Research Scientist with Intel Labs, Santa Clara, CA, USA. From March 2017 to August 2019, he was an Associate Professor with Hongik University, South Korea. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU), South Korea. His research interests include reliable computer systems and accelerator architectures for high-performance computing.