Flexible HLS-Based Implementation of the Karatsuba Multiplier Targeting Homomorphic Encryption Schemes

Michael J. Foster, Marcin Łukowiak, Stanisław Radziszowski
Rochester Institute of Technology, Rochester, NY
{mf1656,mxlecc}@rit.edu, spr@cs.rit.edu

***Abstract*—Custom accelerators for high-precision integer arithmetic are increasingly used in compute-intensive applications, in particular homomorphic encryption schemes. This work seeks to advance a strategy for faster deployment of these accelerators using the process of high-level synthesis (HLS). Insights from existing number theory software libraries and custom hardware accelerators are used to develop a scalable implementation of Karatsuba modular polynomial multiplication. The accelerator generated from this implementation by the high-level synthesis tool Vivado HLS achieves significant speedup over the implementations available in the highly-optimized FLINT software library. This is an important first step towards a larger goal of enabling HLS-based homomorphic encryption in the cloud.**

***Index Terms*—High-Level Synthesis, Karatsuba Multiplier, FPGA.**

I. INTRODUCTION

The recent surge in cloud services is revolutionizing the way that data is stored and processed. Providers such as Nimbix [1], IBM [2], and Amazon [3] are now giving anyone with an Internet connection access to massive amounts of storage and computational resources. Of course, this access comes at a cost. Although some cloud *storage* services offer end-to-end data encryption [4], cloud *computing* services require that the user's data be given to the service provider's computers in an unencrypted form. As a result, users are required to trust the integrity of the service provider and the security of its data centers. This may not be a barrier for some users, but for others who wish to work with personal data such as medical records, or to run proprietary algorithms, the risk is significant.

Homomorphic Encryption (HE) schemes can offer a solution to this dilemma, as they allow computations to be performed on encrypted data. For example, as shown in Figure 1, if a user wants to perform an operation with two plaintext operands, the homomorphic version of the operation can be performed using only the encrypted versions of the operands. The homomorphic operation will return an encrypted output. The decryption of this output is equivalent to the output which would have been received if the normal operations were performed on the unencrypted operands. This type of encryption would enable users to utilize a cloud computing service without ever giving the service provider access to the unencrypted inputs or outputs.

The main obstacle in progress on the above is high compute-cost of multiple-precision integer and polynomial multiplication. Recent work on the acceleration of such algorithms using GPUs has produced some promising results [5]. Field-programmable gate arrays (FPGAs) are another family of computing devices which may compete with GPUs or complement them, offering a higher degree of datapath programmability while also enabling rapid execution of highly-parallel computational loads [6][7][8][9][10].

If hardware acceleration of HE with field-programmable field arrays (FPGA) is to be widely used in the cloud, the hardware implementation process must be cost-effective and accessible to both application and system developers [2][11]. Circuit design for FPGAs has traditionally been a time-intensive process which required specialized training in digital circuit design and hardware description languages. A layer of abstraction needs to be added in order to make it more accessible to application software developers and to accelerate the circuit design process in general. This can be accomplished by making use of state-of-the-art high-level synthesis (HLS) tools, which take software descriptions of algorithms (usually written in C or C++) and convert them into hardware accelerators. This is not a new idea. This type of tools has been around for the last 30 years, but previous generations were much more difficult to use, and the results were often not satisfactory [12]. As the third generation of HLS tools evolves, their performance is becoming much more promising.

In this paper, we begin by presenting previous work which has been done in the development of optimized software libraries and HE hardware accelerators. Then, we describe the details of our approach of mapping the classical recursive Karatsuba multiplication algorithm into a flexible HLS-based FPGA hardware implementation. We also discuss synthesis results in the context of performance and computing resource usage for sizes up to a threshold suitable for use in HE systems.

II. Related Work

A. Software Libraries

Many software implementations of HE are built upon several layers of libraries. At the lowest level, mathematical libraries, such as GMP [13] and MPIR [14], are used to perform core mathematical operations. The functions in these core libraries are individually optimized for specific CPU architectures, often at the level of assembly code. The core libraries are then used inside number theory libraries, such as NTL [15] and FLINT [16], as building blocks for polynomial multiplication and other more elaborate mathematical operations. At the top layer, HE software implementations like

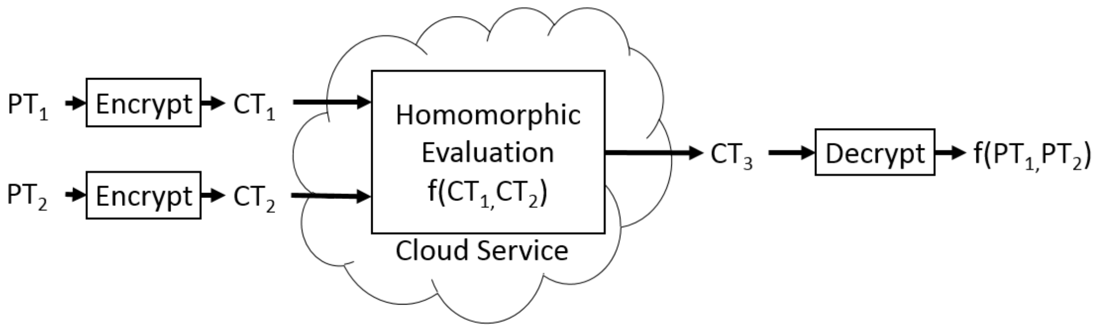

Fig. 1. The process flow for using homomorphic encryption with a cloud service. ”PT” represents plaintext, and ”CT” represents ciphertext. The ”Encrypt,” ”Evaluation,” and ”Decrypt” operations are all defined by the homomorphic encryption scheme being used. Key management is not included in this diagram.

HElib [17], Scarab [18], and other projects [19] use the number theory library functions to implement specific HE schemes.

B. Hardware Accelerators

Since initial work on fully homomorphic encryption was published [20], researchers have been developing custom hardware accelerators for different proposed schemes [6][7][8][9][10] in order to try to boost their performance. Much of this work focused on developing modular designs for the acceleration of the especially expensive operations, such as modular multiplication of very high degree polynomials. There has also been recent works on using HLS tools in the design process of hardware-software HE schemes and Karatsuba multiplication [21][22]. It is not possible, however, to compare them with our results as [21] focuses solely on acceleration of complete HE encryptions, and [22] is targeting polynomials with 32-bit coefficients only.

III. Proposed Methodology

Recent case studies have shown that currently available HLS tools are getting close to being competitive with handwritten Hardware Description Language (HDL) models optimized at Register Transfer Level (RTL) [23][24]. These competitive HLS implementations, however, must be developed with an understanding of the desired final circuit and be carefully optimized with synthesis directives. This notable difference between software written for a CPU and software written for HLS requires creation of HLS-optimized versions of existing software algorithms. This fact has been well understood throughout the history of HLS tool development [25][26].

To support deployment of HE applications (as hardware-software systems where hardware components are implemented on FPGA devices), we are aiming at creating a library of high-level mathematical functions similar to those that are found in software libraries, but tuned for the use with HLS. Once this library is created, the design flow shown in Figure 2 can be used to generate hardware accelerator(s) for hardware-software implementation of the system. The process would simply replace functions which are optimized for CPUs, with functionally equivalent components from our new library.

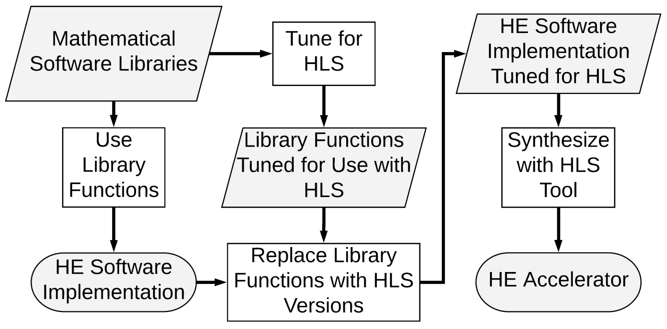
Mathematical Software Libraries
Tune for HLS
HE Software Implementation Tuned for HLS
Use Library Functions
Library Functions Tuned for Use with HLS
Synthesize with HLS Tool
HE Software Implementation
Replace Library Functions with HLS Versions
HE Accelerator

Fig. 2. The HE accelerator design flow proposed by this work.

As significant portion of the custom hardware accelerator work mentioned earlier has focused on the large size multiplication operations, we decided to make Karatsuba polynomial multiplication the first algorithm for our proof-of-concept library. The biggest challenge here was in mapping a recursive algorithm to hardware.

IV. Karatsuba Multiplication and HLS

In low-level software libraries, such as GMP [13] and MPIR [14], several multiplication algorithms are implemented. When the high-level, generic multiplication function is called, one of these algorithms is selected based upon the size of the operands. Table I shows the operand bit-length thresholds used by MPIR to determine which algorithm should be used for

TABLE I
Operand size thresholds for multiplication using MPIR for CPU architectures [27].

Algorithm	Threshold for a Generic Architecture with 32-bit Limbs (bits)	Threshold for x86_64 Haswell with 64-bit Limbs (bits)
Schoolbook	N/A	N/A
Karatsuba	1024	1024
Toom-3	4096	6720
Toom-4	9600	15744
Toom-8.5	12832	19392
FFT	128320	249856

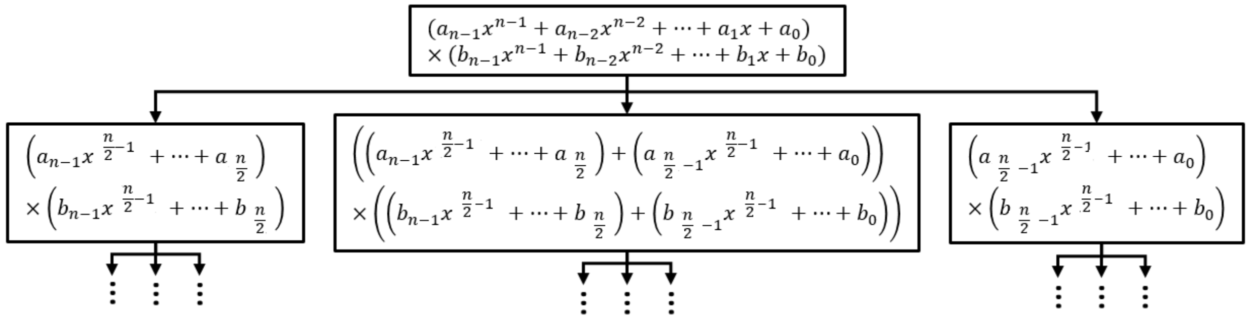
$(a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0)$
$\times (b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \cdots + b_1x + b_0)$
$\left(a_{n-1}x^{\frac{n}{2}-1} + \cdots + a_{\frac{n}{2}}\right) \times \left(b_{n-1}x^{\frac{n}{2}-1} + \cdots + b_{\frac{n}{2}}\right)$
$\left(\left(a_{n-1}x^{\frac{n}{2}-1} + \cdots + a_{\frac{n}{2}}\right) + \left(a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + \cdots + a_0\right)\right) \times \left(\left(b_{n-1}x^{\frac{n}{2}-1} + \cdots + b_{\frac{n}{2}}\right) + \left(b_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + \cdots + b_0\right)\right)$
$\left(a_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + \cdots + a_0\right) \times \left(b_{\frac{n}{2}-1}x^{\frac{n}{2}-1} + \cdots + b_0\right)$

Fig. 3. Top layers of tree structure created by the Karatsuba algorithm with depth $m = \log_2 n$ and 3^m leaves. Three branches evaluate $A_H B_H, A_L B_H + A_H B_L$ and $A_L B_L$, respectively.

integer multiplication. A similar set of algorithms is used by higher-level libraries for polynomial multiplication. The algorithms and thresholds shown in Table I were selected for CPU architectures. These selections may not be the same for much different devices, such as field-programmable gate array (FPGA). This work specifically focuses on implementing the Karatsuba polynomial multiplication algorithm. This algorithm was selected because of some promising Field-programmable gate array (FPGA) implementation results presented in [28].

A. Karatsuba Multiplication Algorithm

The Karatsuba algorithm [29] offers an approach for polynomial multiplication that allows it to outperform the schoolbook method for sufficiently large operands. Let A and B be two polynomials of degree $n-1$ with integer coefficients, where $n = 2^m$. The first step in the algorithm is to divide A and B into polynomials half their size referred to as A_H, A_L, B_H, and B_L, respectively. The operands can therefore be written as:

$$A = A_L + A_H x^{n/2}, \quad B = B_L + B_H x^{n/2}.$$

When multiplied out, they produce:

$$A \times B = A_H B_H x^n + (A_L B_H + A_H B_L) x^{n/2} + A_L B_L.$$

Notice that the term $(A_L B_H + A_H B_L)$ requires two multiplications and one addition. With some rearranging we have

$$A_L B_H + A_H B_L = (A_L + A_H)(B_H + B_L) - A_L B_L - A_H B_H.$$

One multiplication has been removed (since multiplications $A_L B_L$ and $A_H B_H$ once computed can be reused), but one addition and two subtractions have been added. When the operands are large enough that one addition and two subtractions are faster to execute than one multiplication. This is a better choice than the schoolbook method. In terms of complexity, Karatsuba algorithm is of the order $O(n^{1.58})$ [29]. In the remainder of this paper we consider a commonly studied case of polynomials A, B in $Z_q[x]$, where all coefficients are reduced modulo q.

B. Karatsuba Tree and Recursion

As shown in Figure 3, the structure of the Karatsuba equation creates a ternary tree of products. The left, middle, and right branches in the figure represent the $A_H B_H$, $(A_L + A_H)(B_L + B_H)$, and $A_L B_L$ products, respectively. Each product in the tree is further broken down into three sub-products until the leaf nodes of the tree contain integer products of coefficients instead of polynomial products.

C. Karatsuba Tree Evaluation

A C-language implementation was written to follow Karatsuba trees. Software implementations of Karatsuba included in libraries such as FLINT [16] evaluate the tree recursively. Since Vivado HLS does not support recursive implementations, a non-recursive implementation was needed. The process of converting a recursive function into a non-recursive function is straightforward in software when high-level data structures, such as stacks, are readily available, but this process is more complex when the software is being prepared for HLS.

Recursion was replaced by a tree evaluation strategy which simply started at the leaf nodes and worked up the tree computing all of the intermediate products that shared the same level before moving further up the tree. In software, this was implemented using 'for' loops iterating across the width of the tree for each level. By default, each loop was synthesized into its own state machine. Each loop could be unrolled using the UNROLL synthesis directive in order to create parallel instances of the loop logic. This approach was used in order to leverage an FPGA's capabilities by computing intermediate products in parallel.

In most cases pipelining proved to be more effective in boosting performance than loop unrolling. Each 'for' loop was given its own memory structure (array in C) and was pipelined using PIPELINE synthesis directives. This synthesized to several consecutive high-throughput independent processors. DATAFLOW directives were used to place FIFOs between each of these processors. This configuration allowed the series of loop level processors to act as a pipeline at the system level. The throughput of the system is equal to the lowest throughput of any loop level processor in the pipeline. The result was a Karatsuba accelerator with a large latency but high throughput.

The throughput and resource usage of the accelerator can be adjusted by choosing the number of memory ports in each loop level processor's memory structure and changing the number of operations which are performed per loop. The throughput of a level loop processor is often related to the number of iterations of the 'for' loop from which it was derived. If the operations performed in a 'for' loop can be divided into multiple consecutive 'for' loops with fewer iterations than the original loop, the result in hardware would likely have a higher throughput at the expense of a higher latency.

D. Karatsuba Processor Structure

The final Karatsuba processor design, shown in Figure 4, breaks the lower levels of the Karatsuba tree into nine subtrees and evaluates these subtrees in parallel to maximize throughput. In this figure, the parameter n represents the maximum number of coefficients that can be in each multiplicand, and c is the maximum bit width of each coefficient. The parameter k is the number of subtrees which are evaluated in parallel. In the final design, k was equal to 9. The final Subproduct Combine function performed the product combination steps (i.e. additions and subtractions) for the last few levels of the tree and produced the final product.

Figure 5 shows the internal structure of the Karatsuba Tree Evaluation processor. Each step was performed in a 'for' loop which could be unrolled or pipelined via synthesis directives. The best balance between performance and resource use was achieved by pipelining all loops, parallelizing the memory structures with ARRAY_PARTITION directives, and minimizing the use of loop unrolling.

As shown in Figure 5, this implementation of Karatsuba included modulo q reduction of intermediate results leading to c-bit coefficients, $q < 2^c$. Barrett reduction [30][31] was used for the leaf products. For the rest of the circuit, reduction was performed using circuit which would check if a value was above the modulus and, if it was, subtract the modulus from the value. This logic proved to be more resource efficient than implementing a full Barrett reduction circuit for each step. The number of checks and subtractions performed was varying, dependent upon the amount by which the value of interest could have increased beyond the modulus.

E. Results

Execution times of the Karatsuba modular polynomial multiplication function from a highly-optimized software library FLINT were captured on a computer with quad-core AMD A10-7850K running at a 3.7 GHz clock frequency. The computer was running Red Hat Enterprise Linux operating system with 14.5 GB of RAM. The FPGA execution times were taken from the Vivado HLS 2017.2 for a Xilinx Virtex UltraScale xcvu190-flgb2104-3-e device running at the maximum clock frequency of 200MHz. Figure 6 shows the speedup of the HLS Karatsuba implementation running on an FPGA over the FLINT library's Karatsuba polynomial multiplication function running on a CPU. The FPGA hardware results don't include potential overheads associated with data transfers between FPGA fabric and top-level system. On the other hand, those could be minimized in a complete system performing streamlined calculations. For 64-bit coefficients, the achieved speedup is in the range from 4 to 9, for 128-bit coefficients from 48 to 74, and for 192-bit coefficients from 81 to 137.

TABLE II
RESOURCE AVAILABLE ON TARGET FPGA DEVICE.

Xilinx Virtex UltraScale xcvu190-flgb2104-3-e device resources			
BRAM (36Kb)	DSP48E	FF	LUT
3780	1800	2148480	1074240

Since the HLS Karatsuba implementation stores the intermediate products on the FPGA fabric, its ability to scale is strictly reliant upon the amount of available memory resources i.e. BRAMs. Table II presents a summary of major hardware resources available on our target device. The full FPGA resource requirements for each operand size are given in Table III.

The bit width c of the coefficients had no effect on the overall throughput of the system. As the coefficient bit width was increased, wider cascades of DSP48's were synthesized on the FPGA. As shown in Figure 7, these cascades consisted of several groups of parallel DSP48's placed in consecutive layers. The intermediate products calculated by each DSP48 were combined into a final product. The increase in depth of these cascades as the width c was increased was minimal. For example, the latency of the module responsible for the calculation of the leaf products increased by only 2 clock cycles (from 19 to 21 cycles) when the coefficient bit width was changed from 32 bits to 64 bits. The interval of the module stayed constant at 18 clock cycles. This constant throughput despite

TABLE III
RESOURCE USAGE FOR KARATSUBA FPGA IMPLEMENTATIONS. *MORE FPGA BRAMS REQUIRED THAN SUPPORTED BY TARGET DEVICE.

Coeff. per Operand	Coeff. Width (bits)	Resource Usage [%]			
		BRAM	DSP48E	FF	LUT
32	64	9	24	5	7
	128	23	72	8	13
	192	32	72	13	19
64	64	19	24	7	10
	128	37	72	18	25
	192	52	72	18	25
128	64	24	24	9	13
	128	49	72	15	21
	192	67	72	23	31
256	64	30	24	11	15
	128	60	72	19	26
	192	83	72	28	37
512	64	36	24	13	18
	128	72	72	22	30
	192	99	72	33	43
1024	64	122*	24	15	21
	128	223*	72	26	35
	192	344*	72	38	49
2048	64	299*	24	17	24
	128	576*	72	30	39
	192	875*	72	43	55

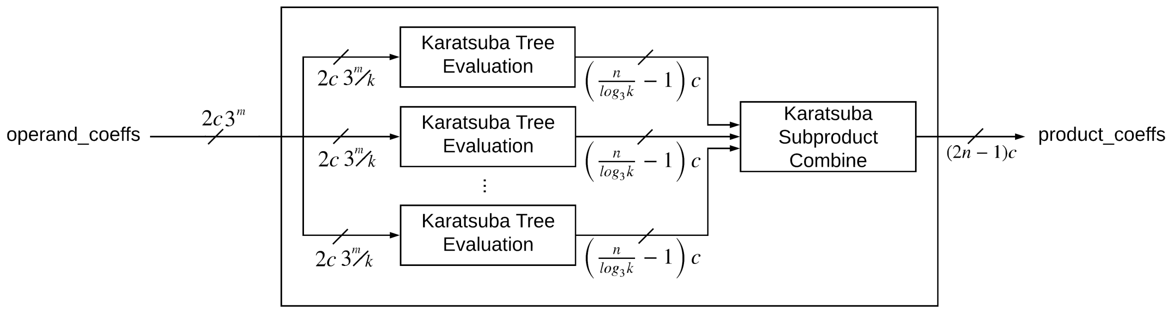
operand_coeffs
$2c3^m$
$2c\,3^m\!/k$
Karatsuba Tree Evaluation
$\left(\frac{n}{log_3k}-1\right)c$
Karatsuba Subproduct Combine
$(2n-1)c$
product_coeffs

Fig. 4. Top-level block diagram of the Karatsuba circuit.

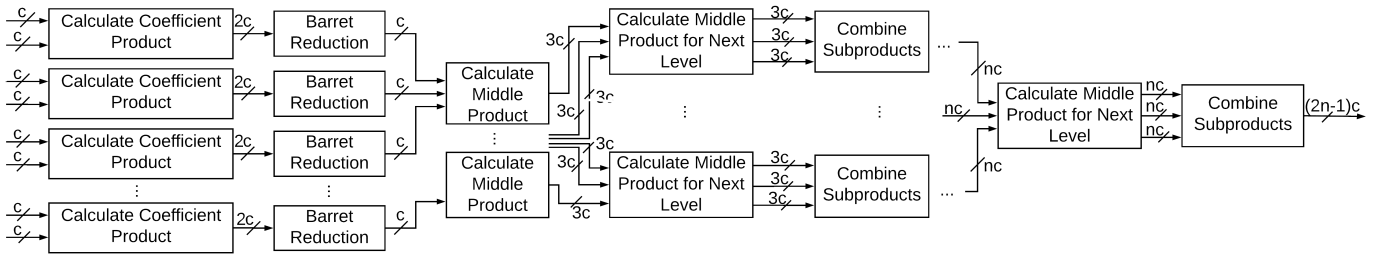

Fig. 5. Block diagram for the Karatsuba tree evaluation circuit.

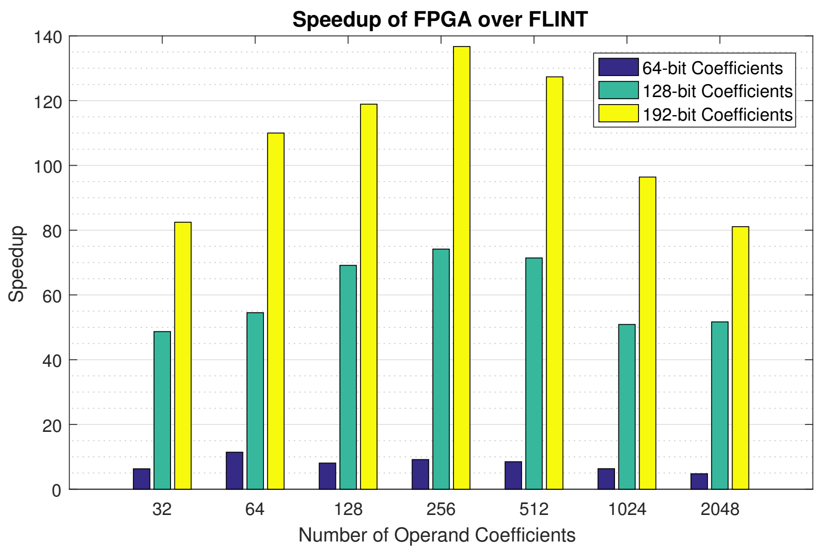

Fig. 6. Speedup results for HLS Karatsuba accelerator.

an increase in coefficient bit width contributed significantly to speedups shown for the larger coefficient sizes in Figure 6.

V. Conclusion and Future Work

HLS is becoming a powerful tool in the world of hardware acceleration. Our results show that, if used with care, current HLS tools can provide significant speedups for computationally demanding operations. HLS tools have come a long way in recent years and will likely be an important part of hardware acceleration in the future.

The potential of our HLS Karatsuba implementation to scale is strictly reliant upon the amount of on-chip memory resources i.e. BRAMs. Our future work will explore modifications to current solution to incorporate off-chip memory instead of BRAMs. Another next step after this work is to use the insights gained from the development of the Karatsuba accelerator for the development of a library of functions that will support a full set of homomorphic operations.

References

[1] Nimbix Inc., “Nimbix: High Performance Computing (HPC) Cloud.”

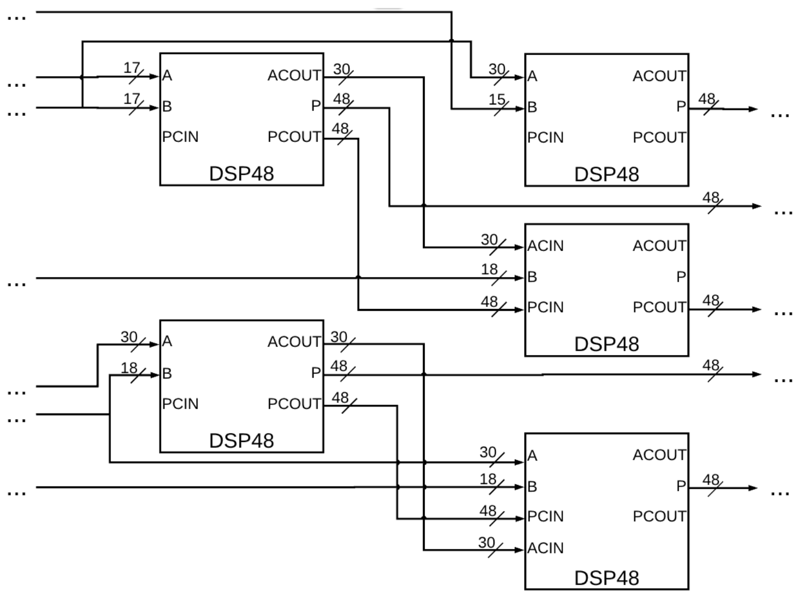

Fig. 7. DSP48 cascade used to calculate intermediate products on polynomial coefficients.

[Online]. Available: https://www.nimbix.net/

[2] S. E. Gianelli, "Xilinx and IBM to Enable FPGA-Based Acceleration within SuperVessel OpenPOWER Development Cloud," Apr 2016.

[3] Amazon Web Services, "Amazon Web Services (AWS) - Cloud Computing Services." [Online]. Available: https://aws.amazon.com/

[4] A. Henry, "The Best Cloud Storage Services that Protect Your Privacy." [Online]. Available: https://lifehacker.com/the-best-cloud-storage-services-that-protect-your-priva-729639300

[5] W. Dai, Y. Doröz, and B. Sunar, "Accelerating NTRU based homomorphic encryption using GPUs," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[6] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1509–1521, June 2015.

[7] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA co-processor implementation of homomorphic encryption," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2014, pp. 1–6.

[8] D. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[9] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," Cryptology ePrint Archive, Report 2015/631, 2015, http://eprint.iacr.org/2015/631.

[10] S. S. Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," Cryptology ePrint Archive, Report 2015/337, 2015, http://eprint.iacr.org/2015/337.

[11] Amazon Web Services Inc., "Amazon EC2 F1 Instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[12] F. Hannig, *A Quick Tour of High-Level Synthesis Solutions for FPGAs*. Cham: Springer International Publishing, 2016, pp. 49–59. [Online]. Available: https://doi.org/10.1007/978-3-319-26408-0_3

[13] "The GNU Multiple Precision Arithmetic Library." [Online]. Available: https://gmplib.org/

[14] "Multiple Precision Integers and Rationals." [Online]. Available: http://mpir.org/

[15] V. Shoup, "Ntl: A library for doing number theory." [Online]. Available: http://www.shoup.net/ntl/

[16] W. Hart, F. Johansson, and S. Pancratz, "FLINT: Fast Library for Number Theory," 2013, version 2.4.0, http://flintlib.org.

[17] "An implementation of homomorphic encryption." [Online]. Available: https://github.com/shaih/HElib

[18] "Homomorphic encryption implementation – Scarab library." [Online]. Available: http://shapecpu.de/scarab-library/

[19] T. Lepoint and M. Naehrig, *A Comparison of the Homomorphic Encryption Schemes FV and YASHE*. Cham: Springer International Publishing, 2014, pp. 318–335. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06734-6_20

[20] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[21] V. Migliore, C. Seguin, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, G. Gogniat, and R. Tessier, "A high-speed accelerator for homomorphic encryption using the Karatsuba algorithm," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 138:1–138:17, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3126558

[22] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki, "HLS design of a hardware accelerator for homomorphic encryption," in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, April 2017, pp. 178–183.

[23] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? a case study," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, Dec 2014, pp. 1–8.

[24] S. Skalicky, C. Wood, M. Łukowiak, and M. Ryan, "High level synthesis: Where are we? a case study on matrix multiplication," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013, pp. 1–7.

[25] M. Fingeroff, *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[26] D. Koch, F. Hannig, and D. Ziener, *FPGAs for Software Programmers*. Springer, 2016.

[27] W. Hart, MPIR Team, T. Granlund, and the GMP development team, *MPIR: The Multiple Precision Integers and Rationals Library*, 2nd ed., 2015. [Online]. Available: {http://mpir.org/}

[28] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Exploration of polynomial multiplication algorithms for homomorphic encryption schemes," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–6.

[29] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akad. Nauk SSSR*, v. 145, pp. 293-294, 1962, in Russian.

[30] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[31] ——, "Communications authentication and security using public key encryption : a design for implementation," Master's thesis, University of Oxford, 1984.