AES Encryption Accelerator Design, and an Attempt to build a Fully Homomorphic Matrix Multiply Accelerator

Prakash Kumar, Deondre Davis, Howard Zhang
University of Michigan
{prakashk,deondred,zhanghow}@umich.edu

***Abstract*—In this paper we first discuss a failed attempt to create a homomorphic encryption matrix-multiply accelerator, followed by a successful attempt to build an AES accelerator. Homomorphic encryption is a type of encryption scheme based on the rings with learnings error assumption (RWLE) that allows for bit-wise operations in the encrypted domain. This opens interesting opportunities in cloud computing on sensitive data, such as AI on health data. Due to a lack of understanding of abstract algebra, we pivoted away from homomorphic encryption accelerator for the aforementioned AES accelerator.**

The Advanced Encryption Standard (AES) has been used widely throughout industry, research, and government [10]. Because of it's wide use, there is always a need to accelerate it. In this paper, we implement and test an FPGA accelerator that speeds up AES-128 Encryption. We compare our results with a software implmentation of the AES algorithm, and with Intel's Intel® Advanced Encryption Standard (AES) New Instructions (AES-NI) as well, which the current industry standard for AES acceleration.

I. Introduction to Homomorphic Encryption

As there is an ever increasing demand for data driven solutions, there is an equally high demand for increased protection and privacy over the user information that drives these solutions. Given these focal points, homomorphic encryption serves as a hallmark strategy in allowing data analysts the flexibility to perform extensive computations on sensitive data while maintaining the privacy of user information. This is achieved by utilizing a homomorphic encryption scheme that enables external parties to perform a set of operations on the encrypted data that simulates particular operations on the original data, without ever needing to expose the original data to said external parties. This process is valid by maintaining homomorphic encryption's criteria that:

$$encr(a*b) = encr(a)*encr(b) \qquad (1)$$

Here, 'a' and 'b' are data sources, * is the desired operation on unencrypted, original data, and * is the necessary operation performed on the encrypted data source to simulate *. Unfortunately, there is significant computational overhead in utilizing homomorphic encryption because of the need to convert the desired operations to their encrypted domain equivalents, which may result in a performance slowdown up to several orders of magnitude based on the operation. By developing a hardware accelerator for certain operations in the encrypted domain, significant performance speedups that could greatly reduce the overall run-time of the process can be obtained. To do this, a homomorphic matrix multiplication accelerator was attempted before switching to AES acceleration.

A. *Matrix Multiplication Acceleration*

In order to accelerate homomorphic matrix multiplication, we will use the suggestion in CHET to vectorize the multiplication operation into vector operations using homomorphic addition, multiplication and rotation [7]. The CHET paper discusses and shows how this multiplication can be vectorized to run in parallel, as shown in Figure 1. Using this model to do the matrix multiplication in an FPGA, an overall encryption pipeline could be put together as shown in Figure 2.

B. *Profiling CKKS, a Homomorphic Scheme*

There are many homomorphic encryption schemes that support addition, multiplication, and rotation, so we will need to pick, understand and implement a scheme that supports all three in order to vectorize the operations to run in parallel in hardware. One such method is the CKKS method for Homomorphic encryption for Arithmetic of Approximate Numbers [6].

Because the CKKS scheme is a popular one, a preexisting software library implemented by Microsoft

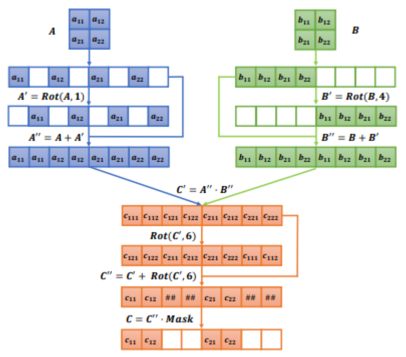


Fig. 1. Homomorphic Matrix Multiplication [7]

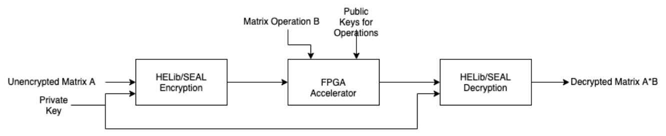


Fig. 2. Simple pipeline for how we will use the software to do the encryption/decryption and use an accelerator to do the matrix multiplication on encrypted data. [7]

called Microsoft SEAL [2] was used to help the homomorphic encryption pipeline. The three operations (rotation, addition, multiplication), as well as encryption and decryption itself were first profiled using Intel Vtune [5] to see where the slowdown occurs. We found that rotr64, multiply, and leftshift were the ”hotspots” of the program, according to Vtune. This was not great for us because left shift is already a very simply operation, and did not offer much in terms of potential. Additionally, Vtune also identified a parallelism level of 23.5 percent, which turned out be fairly low. This offered less opportunity to accelerate, which provided additional reason to eventually adjust our objective.

C. *Homomorphic Acceleration: Moving Forward*

While learning more about homomorphic encryption, we found that our group’s general lack of knowledge on encryption made it difficult to understand the nitty gritty details of homomorphic encryption. Furthermore, since homomorphic encryption is an active area of research, there are too many papers that introduce new schemes that are not implemented or designed in open source software libraries, making them hard to profile and then speed up with an accelerator. The CKKS paper we initially found was already outdated, and our profiling efforts showed results that were difficult for us to understand due to our lack of abstract algebra and encryption knowledge. Ultimately, we spent too much time reading new papers for new encryption schemes, accelerators, and optimizations without really being able to define a proper problem statement, so we moved away form homomorphic encryption for the sake of the course project.

II. AES Encryption

A. *Motivation*

AES Encryption is a cryptography algorithm used to encrypt and decrypt information. Approved by the federal government in 2001, the standard has been adopted by public and private institutions and is widely in use today, making it a prime target for acceleration [10]. Because of this, Intel has created a specific AES New instruction set that optimizes every step of the encryption process for enhanced security and performance. The goal of this project is to build an FPGA accelerator and attempt to achieve the level of enhancement that would other not be achievable on a generic CPU.

B. *Introduction*

AES Encryption is a symmetric encryption scheme, meaning it uses one private key to encrypt and decrypt the data. The encryption follows a specific set of steps that obscure the data until it is unintelligible. Encryption is done on blocks of 16 bytes each, and each block can be encrypted or decrypted independently using a regular set of steps. Because of this, AES encryption has large potential for parallelization by simply stamping out more hardware components to do work on each block.

C. *Steps to AES Encryption*

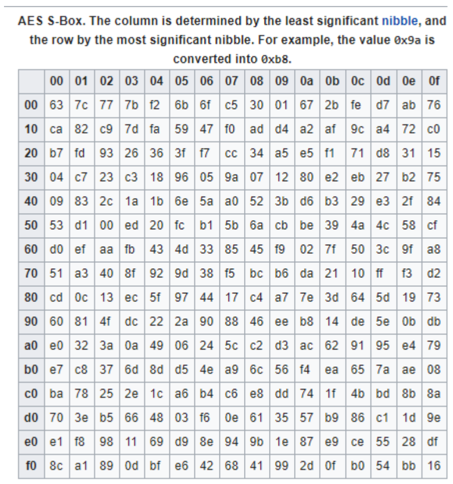
AES encryption can be broken down into a series of rounds:

1) Key Expansion: For each round, AES uses a different private key derived from the input private key. Key Expansion generates these private keys in preparation of each round.

$$\begin{bmatrix} 54 & 73 & 20 & 67 \\ 68 & 20 & 4B & 20 \\ 61 & 6D & 75 & 46 \\ 74 & 79 & 6E & 75 \end{bmatrix} \Longrightarrow \begin{bmatrix} E2 & 91 & B1 & D6 \\ 32 & 12 & 59 & 79 \\ FC & 91 & E4 & A2 \\ F1 & 88 & E6 & 93 \end{bmatrix} \Longrightarrow \begin{bmatrix} E2 & 91 & B1 & D6 \\ 32 & 12 & 59 & 79 \\ FC & 91 & E4 & A2 \\ F1 & 88 & E6 & 93 \end{bmatrix}$$

Example Key Expansion step given left input key

2) Substitute Bytes/Shift Rows: Each block is represented as a 4x4 byte matrix, in order by columns from left column to right. During each round, the first step is to use the Rijndael Substitution Box(S-box) [3] to substitute all values with a different set of predetermined values. Then, the rows of the 4x4 matrix is rotated to the left


AES S-Box. The column is determined by the least significant nibble, and the row by the most significant nibble. For example, the value `0x9a` is converted into `0xb8`.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 3. Rijndael S-box [3]

by each row index. While the S-box could be computed each iteration, it is instead just simply substituted. The S-box is shown in figure 3.

3) MixColumns: The next step of a round is called MixColumns, which computes a matrix multiplication in the Galois-2 field [1] domain on each input word of a data block with the predefined shuffling matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

This multiplication effectively mixes the columns of a data block for obscurity, and the reason for using Galois-2 fields is that operations will never increase the size of the blocks, removing the issue of data overflow. This is achieved due to operations in the Galois-2 field domain being defined as a series of bit-shifts and XORs which allow all of the computation to be contained within the bit-range of the input size. The multiplication operations are defined as follows where 'b' is an input byte:

$$b * 01 = b \tag{2}$$

$$b * 02 = (b << 1) \oplus (b[7]\, ?\, \texttt{0x1B} : \texttt{0x00}) \tag{3}$$

$$b * 03 = (b * 02) \oplus b \tag{4}$$

And addition in Galois-2 field domain is simply an XOR of the operands.

4) AddRound: The last step of a round is called AddRound, which simply computes a Galois-2 field addition (XOR) with the remaining data and the appropriate extended private key.

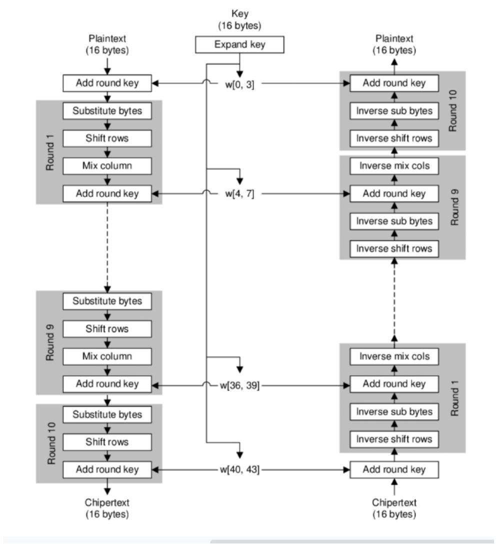


Fig. 4. AES encyrption pipline image [1]

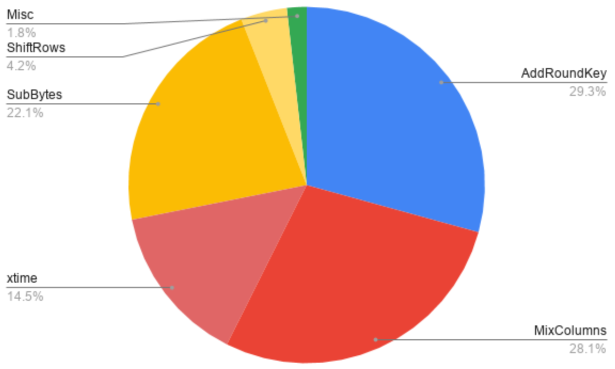
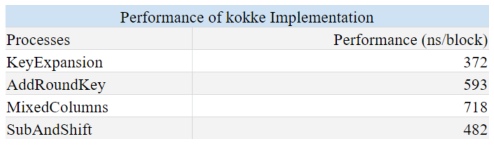


Fig. 5. AES Tiny-AES gprof Profile. Colors chosen to better correspond with the stages used in our accelerator. Program tested with 1GB of data

Common software implementations of AES use the above steps together as shown in figure 4.

D. Profiling AES

To achieve a baseline for comparison with our accelerator, we profiled Tiny-AES, an online C/C++ implementation of AES Encryption and Decryption developed by GitHub user Kokke. Using gprof, we found that the program spent most of it time in AddRoundKey, MixColumns, xtime, SubBytes, and ShiftRows (see figure 6). Further analysis of the program showed xtime was a part of MixColumns. In our FPGA accelerator, SubBytes and ShiftRows were combined into one process. For these two reasons, we colored MixColumns and xtime a similar color (and SubBytes and ShiftRows as well).



Performance of kokke Implementation	
Processes	Performance (ns/block)
KeyExpansion	372
AddRoundKey	593
MixedColumns	718
SubAndShift	482

Fig. 6. Performance of software implementation of each stage of program algorithm. Units of seconds/block to convenient comparison to accelerator. Result of feeding each process 1 gigabyte of blocks

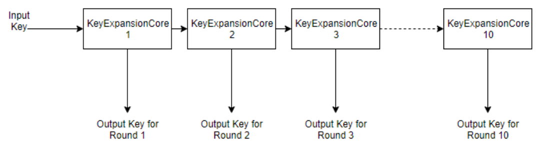


Fig. 8. Key Expansion High Level Block Diagram

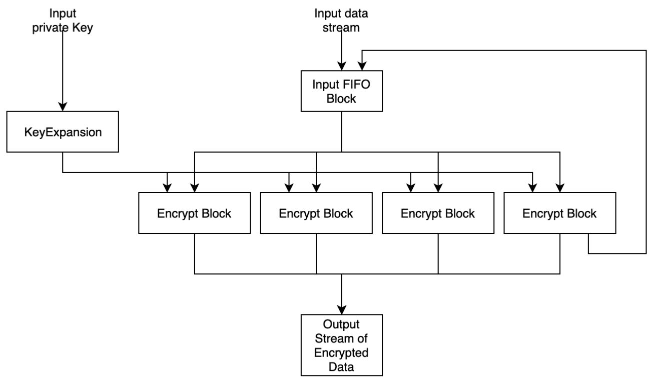
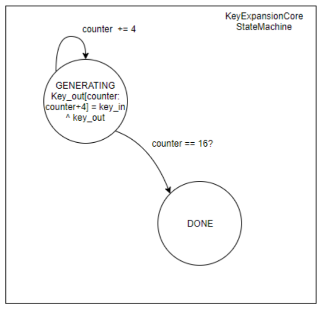


Fig. 7. AES encryption hardware block diagram with 4 parallel processing units



KeyExpansionCore StateMachine
counter += 4
GENERATING Key_out[counter:counter+4] = key_in ^ key_out
counter == 16?
DONE

Fig. 9. Key Expansion State Machine

It can be seen in figure 6 that misc comprises 1.8 percent of the program. All other components can in theory occur in parallel. By Amdahl's Law, we can expect at most 1/0.018 = 55.6x speedup. We'll be coming back to this number in the results.

III. AES-128 Accelerator Design

Under these considerations, we designed an AES-128 accelerator to first operate on one block of data, and then extended it to operate in a parallel manner on multiple blocks. An example of this parallel processing can be shown in figure 7. Based on the space available in the hardware, this can be scaled indefinetely to have as many parallel processing units as we want, linearly increasing the speedup of the acceleration. Of course, this is mitigated by the available space available for the processing.

A. Accelerating KeyExpansion

The first step was to identify how to accelerate each of the individual steps. When implementing KeyExpansion, we found that the entire process was sequential and had a dependency on the immediately previous cycle. Because of this, KeyExpansion was simply implemented without too much acceleration. Ultimately, we found KeyExpansion to take 50 clock cycles, and this only has to be run once for any encryption/decryption, since inputting new blocks of code relies on the same keys generated from KeyExpansion. We built the "KeyExpansionCore", a module used to generate a new key for each round while relying on the inputs from the previous keys. A simple overview diagram is in figure 8 and a statemachine for the generation of the KeyExpansion is in figure 9. One component of the KeyExpansion module is an XOR with an Rcon Iteration byte, which could either be computed or found using a look-up table in the module. To reduce computation and space, we hold a small look up table to just populate the values when in the GENERATING step, as shown in figure 8.

B. Accelerating SubBytes/ShiftRows

It was identified that the Sub Bytes/Shift Rows part of each round could be grouped together. Since it is a simple look up table, the hardware is fairly simple. Figure 10 shows a diagram of the how the components are wired in SubAndShift. Wiring is done using a simple assign block in SystemVerilog.

C. Accelerating MixColumns

In accelerating the MixColumns module, we first observe that due to each Galois-2 field domain operation being composed of only XOR and bit-shift operations, we can implement the entire multiplication in our accelerator as a combinational logic block, thus giving this component the ability to execute the multiplication

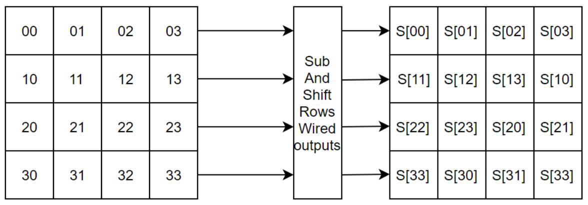


Fig. 10. SubAndShift Diagram to show how the wiring is done with the S-box

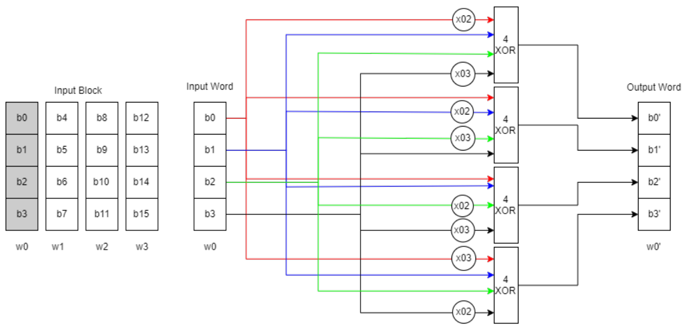


Fig. 11. MixColumns module component diagram for the wiring of the Galois-2 field domain matrix multiplication

within a single cycle. Utilizing logic equations (2), (3), and (4), we constructed the combinational logic block shown in figure 11 to perform the multiplication. Furthermore, we observe that the four words within a given data block are independent of one another when carrying out the multiplications, so we can employ the use of four multiplier blocks in parallel, which enables the module to carry out all necessary operations on an input block within a single clock cycle.

D. *Accelerating AddRound*

Since AddRound is a simple XOR operation in each round, there was no need for any acceleration, XOR's are just stamped out in hardware to compute the results.

IV. Results

A. *Speedup by Step/Stage*

After correctly implementing each module, we tested our speeds against those of Tiny-AES, the results shown in figure 12. The performance values are per block,

Speedup by Stage			
Module	Clock Cycles	Performance (ns/block)	Tiny-AES (ns/block)
KeyExpansion	50	89.35	372
Rounds	12	104.55	1793

Fig. 12. The rounds module represents the combination of 3 module, SubAndShift, AddRoundKey, and MixedColumns.

Overall AES Encryption Comparison	
Implementation	Performance (GB/s)
Kokke's Tiny-AES	0.083
Intel's AES-NI	1.5-58*
Our FPGA accelerator	1.650

Fig. 13. End performance, calculated with 1 gigabyte input. From Tiny-AES to our accelerator, we achieved 19.88x speed up. We found Intel's AES-NI varies in performance (see Results, Overall Speedup)

so where a single round is 12 clock cycles, a full AES encryption would require 12 * 10 = 120 clock cycles. Similarly, the time to perform all SubAndShift, AddRoundKey, and MixedColumns for a single full encryption would be 1793 * 10 = 17930 ns. Although these tests were done while isolating the rounds from key expansion, these metrics told us we were on the right track at this point. We then moved on to combine the individual modules with Key Expansion for our full encryption accelerator.

B. *Overall Speedup*

We managed to achieve almost 20x speed up compared to our baseline. There were many contributing factors. Surprisingly, running the Key Expansion module and the RoundsCore module in parallel helped, more significantly for input data of smaller size. Exploiting computational redundancies within MixedColumns also contributed to a big portion of the speed up, since this module alone account for 42.6 percent of the original CPU time.

Intel's AES-NI technology is considered the industry standard and is supported and actually automatically enabled in many of their latest processors [8]. It allows for hardware level encryption using their specially designed instructions. We found various performance ratings with this technology. On the lower end, we found a test conducted by a consulting firm claiming it could achieve a 1.5 gigabyte per second AES Encryption [9]. On the other hand, a test conducted by Intel itself claims 16051 milliseconds to encrypt 1 gigabyte of data using their AES-NI technology [4]. This translates to 58 gigabytes per second.

Either way, we've shown that our FPGA implementation of the AES accelerator produces significantly higher throughput compared to software implementations not using Intel's instruction set. It is comparable to existing accelerators.

C. *Space and Power Usage*

The current implementation of the AES accelerator uses 9859 LUTs and 2857 Flip Flops, and is using almost

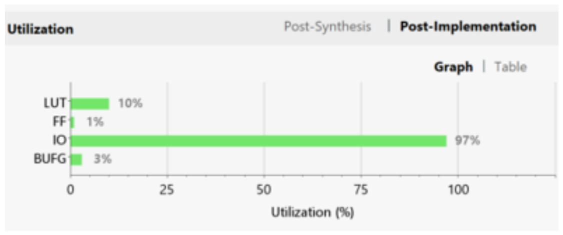


Fig. 14. Accelerator IO Usage in Vivado using device xc7k160tifbg676

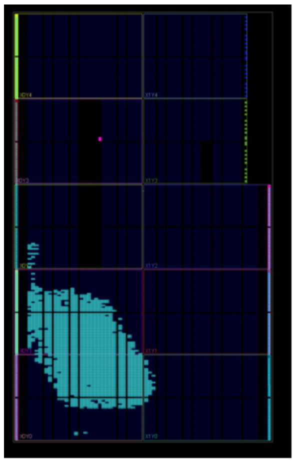


Fig. 15. Accelerator Space Usage in Vivado using device xc7k160tifbg676

all of the IO ports of the xc7k160tifbg676-2L device selected as the output bytes of the encryption. Total Power usage is 393.327 W, according to the implementation in Vivado. It is difficult for us to quantify these numbers to determine how big or small/power efficient the accelerator is, but is certainly something to keep note of. If the space usage is too big, the accelerator becomes impractical to actually use and interface with a CPU, for example.

We can see the large io_usage is shown in Figure 14. This is because we are pipeing all of our output as IO from the accelerator, instead of using a structure to buffer the data as it goes out. In order to see how much space is being used, we looked at the implementation using Vivado and created Figure 15. From this, it seems we may have more space (more FF's and LUTS) to create more parallel structures without cost, either by implementing a "streaming" FIFO for our output ciphertext, or by using hardware with even more IO ports.

V. Next Steps

As this is an initial implementation of AES-128, there are many future considerations to be made:

- What is the tradeoff from number of parallel processing components and space? Since we built an accelerator without space constraints, we could scale up as far as the FPGA we selected allowed. In ther real world, this accelerator would compete for space with other components in a computer, such as the cache or other accelerators, and this must be explored.
- Implementing a FIFO structure to stream out the ciphertext data when polled by a separate device would free up IO ports from the Accelerator, allowing us to linearly scale more parallel processors.
- Support for more advanced versions of AES, including AES-256 and Cipher Block Chaining, Propogating Cipher Block Chaining, Counter mode, and more.
- Support decryption as well as encryption, which should be as simple as reversing the order of many of the modules used in encryption.

VI. Individual Contributions

Below we break down some of these individual contributions. Some tasks we all spent together include writing the report and proposal, and creating the poster.

- Prakash worked on learning and reading about both homomorphic encryption and AES encryption studying the Rings with Learning Errors assumption, as well as about symmetric/asymmetric encryption schemes, how the steps of AES works, etc. With AES, his focus was on the implementation of KeyExpansion module and RoundCore modules in SystemVerilog as well as made much of the integration efforts in putting the modules together and testing them.
- Howard's work was focused on software profiling and code analysis on both the Homomorphic Encryption and AES spaces. Tools used include Intel VTune for Homomorphic Encryption and gprof for AES Encryption. His profiling of Tiny-AES and its subcomponents helped identitfy bottlenecks. Additionally, he conducted resesearch and calulated results from test results to better understand our progress in comparison to the industry standard.
- Deondre worked on much of the reading on homomorphic and AES encryption, and helped implement the mixcolumns SystemVerilog module, as well as a test bench to ensure that it works. He also helped create the overall integration pipeline in SystemVerilog to put modules together into a Round.

Acknowledgements

We would like to thank Professor Satish Narayanasamy, and Jack Wadden for their guidance and advice throughout the project.

References

[1] "Kavilaro aes." [Online]. Available: https://kavaliro.com/wp-content/uploads/2014/03/AES.pdf

[2] "Microsoft seal: Fast and easy-to-use homomorphic encryption library." [Online]. Available: https://www.microsoft.com/en-us/research/project/microsoft-seal/

[3] "Rijndael s-box," Nov 2019. [Online]. Available: https://en.wikipedia.org/wiki/Rijndael_S-box

[4] Admin, "Intel® aes-ni performance testing on linux*/java* stack," Dec 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-aes-ni-performance-testing-on-linuxjava-stack#jdk-performance-improvement-operations-per-minute-evaluation

[5] Admin, "Intel® vtune™ amplifier," Oct 2019. [Online]. Available: https://software.intel.com/en-us/vtune

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," *Advances in Cryptology – ASIACRYPT 2017 Lecture Notes in Computer Science*, p. 409–437, 2017.

[7] R. Dathathri and O. H. K. K. S. M. Todd, "Chet: Compiler and runtime for homomorphic evaluation of tensor programs," in *ASPLOS2019*, 2018.

[8] V. Gite, V. Gite, V. Gite, Zta, S. Pimenta, and H.-J. Petrich, "How to find out aes-ni (advanced encryption) enabled on linux system," Aug 2018. [Online]. Available: https://www.cyberciti.biz/faq/how-to-find-out-aes-ni-advanced-encryption-enabled-on-linux-system/

[9] G. Harari, "A look at the performance impact of hardware-accelerated aes," Aug 2018. [Online]. Available: https://www.scottbrownconsulting.com/2011/10/a-look-at-the-performance-impact-of-hardware-accelerated-aes/

[10] Swenson, "Nist's encryption standard has minimum 250 billion economic benefit, according to new study," Dec 2018. [Online]. Available: https://www.nist.gov/news-events/news/2018/09/nists-encryption-standard-has-minimum-250-billion-economic-benefit