Flexible and Efficient Number Theoretic Transform on FPGA for Zero-Knowledge Proving

Okenna Onkejiaka
Cornell Tech, Cornell University
Electrical and Computer Engineering
New York, NY, USA
noo8@cornell.edu

Sanjay Thallam
Cornell Tech, Cornell University
Electrical and Computer Engineering
New York, NY, USA
sct86@cornell.edu

Young He
Cornell Tech, Cornell University
Computer Science
New York, NY, USA
jh2795@cornell.edu

Abstract—**With the recent boom in popularity of cryptocurrencies, demand for efficient and high performing cryptographic acceleration is higher than ever. Most notably, Zero-Knowledge Proofs (ZKPs) are gaining quite a bit of attention due to their prevalence in blockchain technologies. At the heart of ZKPs is an operation known as the Number Theoretic Transform (NTT): where a finite-field coefficient polynomial is transformed similar to that of the Discrete Fourier Transform (DFT). There have been a variety of works proposed to accelerate NTTs, ranging from CUDA-based GPU acceleration [1] to fully custom ASICs [2]. Taking the middleground, we explore designing an efficient NTT accelerator on Field-Programmable Grid Arrays (FPGAs). In this work we present TheNTT, a novel architecture that combines the benefits of several existing techniques. We then show that it achieves more efficient use of FPGA hardware, while retaining comparable levels of performance in throughput compared to existing accelerators designs.**

I. Introduction

A zero-knowledge proof is a means by which one can prove the validity of a statement without revealing the statement itself. Typically, there are two parties involved. The “prover” is the party trying to prove a claim, while the “verifier” is responsible for validating the claim. As it was First introduced in a paper written in 1985 by Shafi Goldwasser, Silvio Micali, and Charles Rackoff [3], zero-knowledge proofs are not a new concept. Like much of today’s cryptography, there are many different encryption schemes that a zero-knowledge proof can be based on. For the purpose of this project, we will ignore most of the higher-level algorithms and focus on implementing a fundamental arithmetic process that appears across many algorithms: the Number Theoretic Transform (NTT).

The Number Theoretic Transform is an alternate take on the Discrete Fourier Transform that leverages the power of Finite Fields to optimize and simplify the calculation process that occurs when converting between domains. One of the primary optimizations of the NTT is that it works completely in the integer domain compared to the DFT which uses a mix of imaginary and real numbers. This heavily optimizes the math which helps reduce the mathematical complexity from $O(n^2)$ to $O(nlog(n))$.

II. Background

A. Mathematical Motivation

In zero-knowledge proving, polynomials are used to encode knowledge that are passed between the prover and the verifier. As a result, multiplication between (very complex) polynomials are commonplace throughout the encryption process. However, consider the naive grade-school multiplication algorithm, where f, g are two polynomials of finite degree N:

$$f(x) \cdot g(x) = \sum_{i=0}^{N} f_i x^i \cdot \sum_{j=0}^{N} g_j x^j = \sum_{i=0}^{N} \sum_{j=0}^{N} (f_i g_j) x^{i+j}$$

It is clear that this calculation is $O(N^2)$ with respect to the number of computations, which is very slow in practice. One method of making the multiplication faster involves the use of the Discrete Fourier Transforms (DFT), which leverages the fact that point-value representation of polynomials can be multiplied efficiently. We define the DFT of $f = \sum_{i=0}^{N} f_i x^i$ via its coefficient vector $[f_0, \ldots, f_N]$:

$$\tilde{f}_k = f(\omega^k) = \sum_{j=0}^{N-1} f_j \omega^{jk} \text{ where } \omega = e^{2\pi i/N} \text{ and } \omega^N = 1$$

This effectively calculates the value of f at ω^k for each positive $k \leq N$, giving us a valid point-value representation of f. Then, each $\tilde{f}_k \cdot \tilde{g}_k$ represents the transformed output of $f \cdot g$, by the convolution theorem of Fourier transforms. From this, one can then undo the DFT by applying the inverse Discrete Fourier Transform (iDFT):

$$f_k = \frac{1}{N} \sum_{j=0}^{N-1} \tilde{f}_j \omega^{-jk} \text{ where } \omega = e^{2\pi i/N} \text{ and } \omega^N = 1$$

From this, a full end-to-end fast polynomial multiplication pipeline is built; but in its current state, the DFT/iDFT algorithm still bottlenecks the computation to $O(n^2)$. To remedy this, we utilize Fast Fourier Transform (FFT), which utilizes recursion to more efficiently calculate the transforms:

$$\sum_{j=0}^{N-1} f_j \omega^{jk} = f_0 + f_1\omega + \cdots + f_N \omega^N$$

$$= (f_0 + f_2 + \cdots + f_{N-1}^{N/2-1}) + \omega(f_1 + f_3 + \cdots + f_N^{N/2-1})$$

$$= \sum_{j=0}^{N/2-1} f_{2j}\omega^{jk} + \omega \sum_{j=0}^{N/2-1} f_{2j+1}\omega^{jk}$$

(This can be done in a similar way for iDFTs.) From this, we can see that any DFT can be decomposed recursively into two smaller DFTs. With the recursion master theorem, it is clear that we have reduced the complexity to $O(n \log n)$ for the transform. This improves the overall complexity to $O(n \log n)$ which is asymptotically better than the naive $O(n^2)$.

It is important to note that by restricting the coefficients to a finite field, this multiplication process can become easier to achieve in hardware. In fact, this is not an outlandish restriction to make: integers can be efficiently processed, as well as being finite in terms of possible expressions. To do so we restrict the coefficients to be in $\mathbb{F}_p$, which is the Galois group with p elements. In such fields, it can be shown that if N is a divisor of $q-1$, then all N-th roots of unity exist; and by the cyclic nature of $\mathbb{F}_p$, there exists a unique ω such that $\omega^N = 1$. This indicates that even though our coefficient field is no longer algebraically complete (like $\mathbb{C}$), we can still perform transforms akin to DFTs.

In this case, the (inverse) Number Theoretic Transforms (NTTs/iNTTs) are used in placed of DFTs/iDFTs in the aforementioned process. We similarly define the NTT and iNTT for f as:

$$\tilde{f}_k = \sum_{i=0}^{N-1} f_i r^{i \cdot j}, f_k = \frac{1}{N} \sum_{j=0}^{N-1} \tilde{f}_j \omega^{-jk} \text{ where } \omega^N = 1 \text{ in } \mathbb{F}_p$$

Notably, the same recursive idea of FFTs can still be used for NTTs, making the multiplcation pipeline still $O(n \log n)$ for $\mathbb{F}_p$-coefficient polynomials. As a result, NTTs are often heavily used in zero-knowledge proving situations, and a hardware acceleration of this calculation could yield considerable performance advantages.

B. Primary Schemes of Zero-Knowledge Proofs

When implementing the actual cryptography element for zero-knowledge proofs, there are two main methods: zk-STARK and zk-SNARK. At a high level, the prover generates a proof for a certain completed task and the verifier is able to use the proof result to determine if a request is valid or not. This approach is used as it allows for cryptography to be done without the verifier ever learning any details about the data itself, leaving it obscured and secure. This approach uses hashing to encrypt the data which is a process that is highly optimized, however, it may be vulnerable to decryption with the advent of quantum computing.

In contrast to the SNARK approach, STARK is more resource heavy; however, the data encrypted by this method may be more secure. This comes down to the fundamental encryption methodology as STARK uses collision-resistant hashing instead of the ecliptic curve model used by SNARK. Another major pro for STARK is its scalability. When expanding the system size, the complexity for STARK grows linearly; whereas it grows exponentially for the SNARK approach, making it harder to verify. However, at this point in time, the STARK approach is less widely used in industry as the SNARK approach has been present for much longer, and many networks are already optimized. Therefore, a lot of the existing research has been focused on the SNARK approach.

III. Related Work

Different architectures have been proposed to implement the NTT algorithm in hardware [4], [5]. Many of them are tailor-made to suit the specific application being performed, and as such, they make certain assumptions about the size of the prime number being used for the finite field. For example, [5] postulates 3 different variations of NTT each with a field of different sizes. In this case, the NTT is being employed in Spectral Modular Arithmetic (SMA) cryptographic algorithm. Algorithms for the fundamental modular arithmetic operations of addition, subtraction, reduction, and division are given, and these vary depending on the field size we choose to operate over. In the paper [5] the authors estimate some theoretical values for resource usage and compare their finally synthesized design to their estimates.

[6] Discussed an implementation that chose to optimize both the long I/O time and the NTT operation itself. This was to achieve a balanced design on a hardware-software co-designed model. This attention to I/O time was also seen in the design by [4].Other works such as [4] shows another NTT implementation called the "Four-Step NTT". This method assumed the use of special Goldilocks Primes, which are of the form: $q = \varphi^2 - \varphi + 1$. Choosing this value as our modulus allows for unique optimizations due to its unique properties [7]. Furthermore, by implementing certain steps in the Four-Step-NTT recursively, the authors devise a new algorithm called the "N-step NTT" [4]. In this approach, a large NTT is split into a series of sub-NTTs of a base size 2^l, which is suitable for pipelined hardware implementations that will be able to sustain an output of l elements per cycle.

Works such as [8], [9] described more general classifications of NTT hardware. Broadly, there are two categories of approaches: Run Time Configurable (RTC), in which the parameters of the cryptographic scheme (modulus and degree of the polynomial) can be changed at run-time; as well as Compile Time Configurable (CTC), where the entire design must be recompiled in order to change these features.

As we continued to build, it was important for us to obtain a solid understanding of the different methods [10] of calculating NTTs. Their work describes different variations of NTTs as well as the number of additions and multiplications associated with each one. It provided a solid basis for us to compare the tradeoffs and complexities of each.

IV. Technical Approach

A. High Level Overview

Our design was an implementation of the Cooley-Tukey NTT algorithm. In this method, inputs are taken in bit-reversed order (the index of the bit is reversed to find its new index) for the operation. In our case, the odd input is multiplied by the corresponding twiddle factor before it is added to, or subtracted from the even input. The architecture is shown in 1 below. The design is optimized to be modular and flexible. Our design will use 2 processing elements (PEs) and each will have a block RAM. This is one less block RAM per PE than the original design from [11]. BRAMs are reused at every stage and the address generation module controls the write address for the read address for data going in and out of the RAMs. At the final stage, the outputs of the PE are streamed directly into a modified PE that stores the final value in a third block RAM that a user can access easily. Each of the key components are described in the sections below.

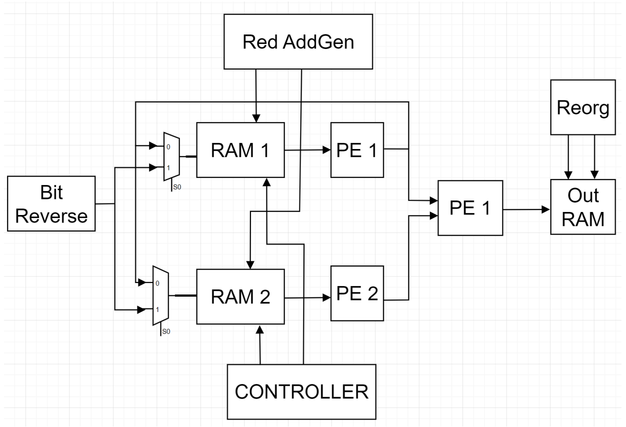

Fig. 1. Top level block diagram of the TheNTT module

B. Main Processing Element (Butterfly Unit)

The main engine in the NTT operation is the butterfly operation. For our implementation, We will be using the Cooley-Tukey algorithm which takes inputs in bit-reversed order. With this implementation, the outputs end up being stored in the correct order. It is also commonly referred to as the decimation in time algorithm. The PE takes in data through a single port and passes it through a multiply unit and a shift register. The delay on the shit register is one more cycle than it takes for the multiply to produce its output. The diagram is shown below.

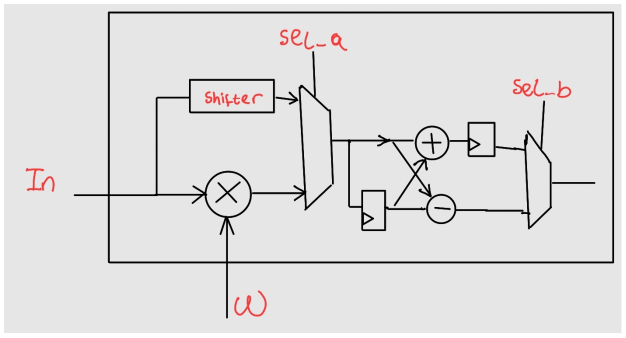

Fig. 2. Architecture of our processing element (PE)

There is a control unit that manages the toggling of the sel_a and sel_b. With this design, each output is produced one at a time with the bottom half coming out first.

C. Control Logic for Main PEs

The PEs are designed with the sole purpose of taking in input and performing the butterfly operation. The module has no notion of valid or invalid values. All control signals are outsourced to different modules to ensure the correct flow of data through the system. The Controller Module plays a key role in this management and as the system scales, it will ideally manage all PE's, This is driven by the fact that all PE's and other associated block RAMS are homogenous. Essentially, they move in lockstep performing the exact same operation at the same time at each stage.

D. BRAM Address Generation - Inputs

The BRAM address generator is responsible for selecting which addresses in the BRAM are read and written to at any one point in time. This module is a critical piece of the overall implementation of our circuit as it is responsible for managing the bulk of the data that is used. The main challenge of this component was trying to minimize the amount of logic needed to perform everything whilst still making the code fully parameterized. The bulk of this logic was associated with the read operations. In this part of the code, the goal was to determine at any cycle, which addresses needed to read to. One of the issues with this is that it the address locations being read are constantly changing and the offset between the addresses is constantly changing as well. To handle this, bit manipulation was leveraged as it offered an efficient and simple solution to address selection. The algorithm below outlines how this was done.

At a high level, this algorithm is a hybrid between a state machine and a counter that can increment and maintain bit states as needed. This algorithm was intentionally designed using only shifters, increments, and registers to minimize the amount of LUTs and power needed to run it. Prior to taking this approach, the designs became extremely large and inefficient which is in sharp contrast to this approach. Another major optimization in the design is the fact that a single BRAM controller can be used to control all of the input storage BRAM blocks across the entire system. As the system scales with more copies of the NTT engine on this chip, this is a major advantage as it greatly reduces the amount of ancillary circuitry needed allowing for more efficiency and throughput.

Algorithm 1 BRAM Read Algorithm

```
 1: procedure BRAM ADDRESS GENERATION
 2:     Initialize local control signals
 3:     while done == 1'b0 do
 4:         pause (wait for data loading)
 5:     while NTT not done do
 6:         if counter == 1'b0 then
 7:             counter ← 1'b1
 8:             if memAddr2[pivot] == 1'b1 then
 9:                 if pivot == 1'b0 then
10:                     pivotLocation ← pivotLocation - 1'b1
11:                     reset upperAddress, lowerAddress
12:                 else
13:                     lowerAddress ← 0
14:                     upperAddress ← start of upper bound
15:             else
16:                 lowerAddress ← next needed address
17:                 upperAddress ← next needed address
18:         else
19:             lowerAddress ← lowerAddress + 1
20:             upperAddress ← upperAddress + 1
```

The other key functionality of this module is the write component. Compared to the read addressing scheme, the write addressing is relatively simple as it consists of just a shift register. When a read address is generated, it is added to the shift register. The length of this shift register is directly determined by the width of the input data and adjusts accordingly. By using a shift register instead of manually recalculating the read addresses, the complexity of this circuit was effectively cut by 50%.

E. BRAM Address Generation - Twiddle Factors

In the design, the twiddle factors are preloaded onto their own BRAM unit akin to how the inputs are preloaded into their own BRAM. Compared to the size of each of the input BRAMs, the twiddle factor BRAMs are half the size and need a completely different algorithm for accessing. Unlike the inputs which are eventually updated and rewritten to the BRAM, the twiddle factor BRAM effectively acts as a read-only memory once the factors are loaded on during initialization. This allowed us to marginally reduce the amount of signaling needed to complete this circuit. The algorithm below outlines the high-level outline for how this module works.

As with the previous BRAM addressing scheme, it was heavily optimized and only uses adders and shifters

Algorithm 2 BRAM Twiddle Factor Algorithm

```
1: procedure TWIDDLE FACTOR ADDRESS GENERATION
2:     Initialize local control signals
3:     while NTT not done do
4:         counter ← counter + 1
5:         update twiddleAddress using counter
6:         if counter == maxCount then
7:             update state using counter and twiddleIndex
```

which are highly efficient from a hardware perspective. This was an intentional choice to help reduce the amount of resources needed for this circuit.

F. Modified Processing Element for Last Stage Calculations

In our implementation, the butterfly operation from the first stage to the second-to-last stage is all calculated in the previously described main PEs. However, the last stage of calculation is yet to be addressed. The Cooley-Tukey butterfly enjoys the benefit of having local memory access for every stage except the last: for example, with 2 main PEs and a degree 8 polynomial, the top PE would only require access to input 0 to 3, and the bottom PE only input 4 to 7. Even though this is more convenient than the Gentleman-Sande butterfly, it requires the usage of an additional PE to handle the case for the last stage.

Furthermore, it is interesting to note that due to the mathematical derivation of the butterfly calculation, the twiddle factor will always be 1 for the last stage. Therefore, we further optimize this additional PE for area and use a streamlined modular adder-subtractor pair. The overall architecture can be seen in 3 below.

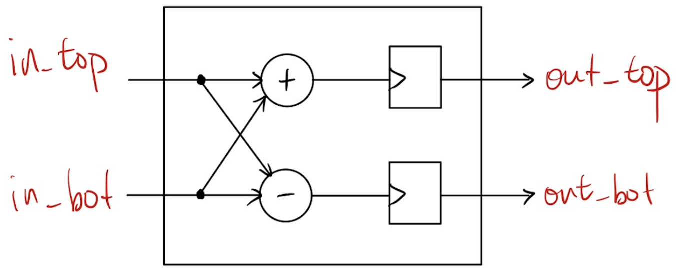

Fig. 3. Modified PE Architecture

G. Reordering Controller and Output Cache

The streaming nature of this modified PE necessitates the use of an reordering cache so that the ordering in the output coefficients are correct. Thus, we implement a control block specifically for the modified PE. This block inputs an acknowledgement from the main controller as to when it is outputting valid second-to-last stage results, and outputs three signals: wr_addr_top, wr_addr_bot, and rd_addr. The controller maintains state with a counter that keeps track of how many sets of second-to-last stage results it have received from the upstream main PEs, and uses that to deduce corresponding read and write addresses that is used for the reodering RAM block. The write address logic is as follows:

$$wr\_addr = \{\neg is\_top, \neg counter[0], counter[MSB:1]\}$$

And the read address simply counts up from 0 after all outputs have been cached from the modified PE. This provides the top level module with correctly ordered final outputs, which marks the completion of the NTT calculation.

The reordering logic necessitates a 2-read 1-write cache; this is not supported with a simple BRAM. Thus, we implement a custom cache consisting of 2 separate BRAM modules, with the final output selected using a multiplier based on the most significant bit of rd_addr. The architecture for this block can be seen in 4, where blue inputs are controlled by the aforementioned controller block.

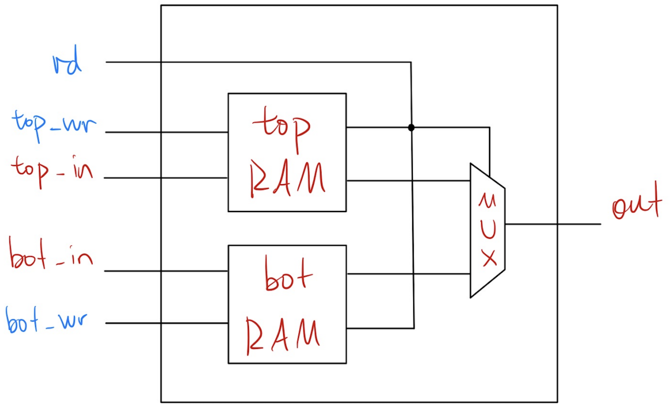

Fig. 4. Out RAM architecture, with custom 2-read 1-write

V. RESULTS

A. Hardware Utilization

To determine our hardware utilization, we utilized Quartus Prime and compiled it to an Arria II FPGA.

The Arria II is a smaller FPGA compared to some more modern FPGAs such as an Agilex 7, however, we ran into issues compiling with Modelsim-Prime-Pro thus we stuck to Quartus Prime Standard. The table below includes the hardware utilization of our design on the Arria II chip. Our design was synthesized assuming it would be performing a 1024-point NTT with a 13-bit wide coefficient. We were surprised to see that no block RAMs were inferred and it is likely because the synthesis tool determined that they would fit nicely into the LUTs and that's what occurred. No DSPs were used likely because much of the large multiplication was divided into smaller chunks in the multiplication modules. These new operations likely did not map well to the DSPs.

Clk Period	5.93 ns
Frequency	188 MHz
ALUTs	366
ALMs	421
DSP	0
Combinational ALUT/register pair	235

TABLE I
HARDWARE UTILIZATION AND FREQUENCY ON ARRIA II

B. *Throughput Estimations*

Although we did not explicitly test for throughput in a practical application, one can make a reasonable estimation as to the throughput of this design. We are basing these numbers on a single NTT unit with N number of butterfly stages. The table below contains the results of these calculations.

In general, the number of clock cycles it takes to compute an NTT for our design is $2N * Log_2(N)$. For the throughput calculations, we used the clock frequency from the synthesis of 187 MHz which resulted in a clock period of 5.3 ns.

VI. DISCUSSION

A. *Comparison to Prior Works*

When compared to previous work, this design is theoretically able to cut the number of instantiated BRAM blocks in half. Prior papers have used 2 BRAM blocks per butterfly unit with each BRAM block holding the entire input dataset. In comparison, the new design only uses 1 BRAM block per butterfly unit. This effectively cuts the number of BRAM blocks in half. On an FPGA, these are amongst the largest blocks and also consume a large amount of power, therefore reducing the number of blocks theoretically leads to improved space and power efficiency. The trade-off with this design is the amount of time needed to actually perform the calculations. When 2 BRAM blocks are present per butterfly unit, the upper and lower line data can be read at the same time, however, with this design an extra clock cycle is needed to process the 2 data elements along with additional circuitry to ensure data quality. Another area of comparison is in the algorithm used for this implementation. When implementing the NTT, there are 2 primary algorithms: Cooley-Tukey and Gentleman-Sande. The primary difference is the order of operations and the ordering in which the twiddle factors are applied. Although at a high level this does not make a huge difference, we did find that our circuitry for certain circuit controls was slightly smaller when we tried for both.

B. *Future Work*

In terms of future work, there is a vast amount of potential that can be derived from this baseline design. From a design perspective, one of the major areas for potential efficiency improvements is the modular multiplier. When performing operations from a hardware perspective, multiplication is particularly costly. Although an efficient modular multiplier was used, some areas for improvement potentially exist including parallelism and increased pipelining. Another future step for this design is to implement the inverse NTT. At this point, the circuit performs the traditional NTT operation only whereas many of the benchmark papers also developed inverse NTT circuitry. Another area of work would be to design a specific FPGA in mind to make use of some more hardened IP. At this point, the hardware is designed platform agnostically and this could potentially improve the performance of the circuitry. We also are a little bit skeptical of the values generated by Quartus for utilization so we would like to go back and recheck those along with compiling to more FPGA models to get a larger data sample. Finally, more work will be performed to compare the new design with prior work on parametric NTTs to cross-compare performance and power specifications.

VII. CONCLUSION

When creating novel hardware designs, many considerations have to be made when optimizing designs based on the needs of the circuit. Having the opportunity to build upon and optimize a design for the NTT has been a rewarding experience and has given us an immense amount of insights into the work that goes into such a design. Going forward, there is still significant work to

Number of Layers	Memory Loading CC	Static Setup CC	CC/layer	Number of Layers	Total CC	NTT/s
8	8	11	4	3	31	6086427
16	16	11	8	4	59	3197953
1024	1024	11	512	10	6155	30654

TABLE II
THROUGHPUT CALCULATIONS

be done, and as a team, we seek to further refine and optimize our designs. As we move into a more digital world with greater influence from the Blockchain and other technologies dependent on the NTT, further optimizations will have to be made to improve performance and power efficiency and this marks an important first step down this path.

VIII. FOOTNOTE

Link to the project GitHub Repository: https://github.com/Vrownie/NTT

REFERENCES

[1] O. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *J. Supercomput.*, vol. 78, no. 2, p. 2840–2872, feb 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-03980-5

[2] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 711–725. [Online]. Available: https://doi.org/10.1145/3470496.3527415

[3] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: Association for Computing Machinery, 1985, p. 291–304. [Online]. Available: https://doi.org/10.1145/22145.22178

[4] Uveltanna, "A throughput-optimized fpga architecture for goldilocks ntt," https://www.ulvetanna.io/news/fpga-architecture-for-goldilocks-ntt, 2023.

[5] G. X. Yao, R. C. Cheung, C. K. Koc, and K. F. Man, "Reconfigurable number theoretic transform architectures for cryptographic applications," in *2010 International Conference on Field-Programmable Technology*, 2010, pp. 308–311.

[6] A. C. Mert, E. Ozturk, and E. Savas, "Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, Aug. 2019.

[7] R. Bloemen, "The goldilocks prime," in *Math & Engineering*, 2023.

[8] K. Derya, A. C. Mert, E. Öztürk, and E. Savaş, "CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication," *Microprocess. Microsyst.*, vol. 89, no. 104451, p. 104451, Mar. 2022.

[9] J. Mu, Y. Ren, W. Wang, Y. Hu, S. Chen, C.-H. Chang, J. Fan, J. Ye, Y. Cao, H. Li, and X. Li, "Scalable and conflict-free NTT hardware accelerator design: Methodology, proof, and implementation," *IEEE Trans. Comput.-aided Des. Integr. Circuits Syst.*, vol. 42, no. 5, pp. 1504–1517, May 2023.

[10] L. Listyalina, E. L. Utari, and M. W. Wizando, "Implementation of fast fourier transform (fft) for infant crying detection," *Indonesian Applied Physics Letters*, vol. 4, no. 1, pp. 1–8, Aug. 2023.

[11] A. C. Mert, E. Karabulut, E. Ozturk, E. Savas, M. Becchi, and A. Aysu, "A flexible and scalable NTT hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Mar. 2020.