Low-Cost and Area-Efficient FPGA Implementations of Lattice-Based Cryptography

Aydin Aysu, Cameron Patterson and Patrick Schaumont
Electrical and Computer Engineering Department
Virginia Tech
Blacksburg, VA, USA
e-mail: {aydinay,cdp,schaum}@vt.edu

Abstract—The interest in lattice-based cryptography is increasing due to its quantum resistance and its provable security under some worst-case hardness assumptions. As this is a relatively new topic, the search for efficient hardware architectures for lattice-based cryptographic building blocks is still an active area of research. We present area optimizations for the most critical and computationally-intensive operation in lattice-based cryptography: polynomial multiplication with the Number Theoretic Transform (NTT). The proposed methods are implemented on an FPGA for polynomial multiplication over the ideal \( \mathbb{Z}_p[x](x^n + 1) \). The proposed hardware architectures reduce slice usage, number of utilized memory blocks and total memory accesses by using a simplified address generation, improved memory organization and on-the-fly operand generations. Compared to prior work, with similar performance the proposed hardware architectures can save up to 67% of occupied slices, 80% of used memory blocks and 60% of memory accesses, and can fit into smallest Xilinx Spartan-6 FPGA.

Keywords—Lattice-based cryptography; Number Theoretic Transform; FPGA; ideal lattices

I. INTRODUCTION

Lattice-based cryptography relies on the hardness of lattice problems. Lattice-based cryptosystems are quantum resistant and are often provably secure based on worst-case hardness assumptions. In recent years, this resulted in development of several cryptographic primitives such as digital signatures [1], encryption schemes [2] and fully homomorphic encryption [3]. An excellent tutorial on lattice theory and associated hardness problems can be found in [4].

Fig. 1 shows a visual of the basic problems in lattice theory in 2D lattices, including the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). SVP is, given the lattice \( L \) generated by the basis \( B_0 = (b_0, b_1) \) and \( B_1 = (b_2, b_3) \), finding the shortest non-zero vector \( \lambda_1 \) inside the lattice with respect to the \( L_2 \)-norm. CVP is, given the basis \( B_0 \) and \( B_1 \) of lattice \( L \) and a point \( C = (c_0, c_1) \) which may or may not be on \( L \), finding the closest point to \( C \) inside \( L \) and associated vector \( \lambda_2 \) with respect to the \( L_2 \)-norm. The fundamental and the most computationally-intensive operation of lattice-based cryptography is the multiplication of two points inside the lattice. A polynomial multiplication modulo \( p \) with a reduction function \( (x^n + 1) \) corresponds to the multiplication of two points inside an \( n \)-dimensional ideal lattice [5], which can be implemented very efficiently using the Number Theoretic Transform (NTT) [6].

Figure 1. Visualization of fundamental lattice problems in 2D

Several high-throughput FPGA implementations were recently proposed for polynomial multiplication over the ideal \( \mathbb{Z}_p[x](x^n + 1) \) using NTT [7,8]. Most recently, Poppelman et al. present a low-cost hardware architecture [9]. However, these architectures are area intensive and further optimizations for low-cost targets are still possible.

In this paper, we propose low-cost and area-efficient hardware architectures that have similar performance while reducing up to 67% of occupied slices, 80% of used BRAMs and 60% of BRAM accesses compared to [9]. The proposed hardware architectures are called 2DSP and 3DSP depending on the total number of dedicated DSP blocks used to implement multiplication with NTT. The NTT of the coefficients of two input polynomials operates in an interleaved fashion to use a simple dual-port BRAM. We optimize memory organization in such a way that at each read/write access, two concatenated coefficients of the polynomials are read/written which reduces the number of BRAMs and BRAM accesses compared to prior work. Moreover, instead of storing the powers of the multiplicative generator (twiddle factors) and its square roots in a ROM, the proposed architecture generates them on-the-fly. These optimizations do not only reduce BRAM usage and access, but also decrease the slice utilization due to simplified, unified and reduced address generation logic.

The rest of the paper is organized as follows: Section II gives an overview of NTT and presents a memory-efficient NTT algorithm. Section III proposes methods for area efficiency hardware architecture, implementation results are given in Section IV and Section V concludes the paper.
II. NUMERICAL THEORETIC TRANSFORM

A. Overview

NTT is essentially a Discrete Fourier Transform defined over a finite field or ring and does not require complex arithmetic [10]. The generic forward NTT\(_n(a)\) transforms an \(n\)-degree polynomial of the form
\[
f(x) = a_0x^0 + a_1x^1 + \cdots + a_{n-1}x^{n-1}
\]
with the modulo \(p\) coefficients \((a_0, a_1, \ldots, a_{n-1})\) into a \(n\)-degree polynomial of the form
\[
F(X) = A_0X^0 + A_1X^1 + \cdots + A_{n-1}X^{n-1}
\]
with the modulo \(p\) coefficients \((A_0, A_1, \ldots, A_{n-1})\) defined as
\[
A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \mod p, i = 0, 1, \ldots, n - 1
\]
where \(\omega\) (twiddle factor) is a given primitive \(n\)-th root over the polynomial ring. The inverse transform, NTT\(_n^{-1}\), computes:
\[
a_i = n^{-1} \sum_{j=0}^{n-1} A_j \omega^{-ij} \mod p, i = 0, 1, \ldots, n - 1
\]
The NTT exists if and only if for some integer \(k, p = kn + 1, \omega^k = 1 \mod p\), and \(\forall i < n, \omega^i \neq 1\).

The complexity of schoolbook polynomial multiplication is \(O(n^2)\) which makes it unfeasible to use for large \(n\) values (a typical \(n\) value for lattice-based cryptography is 1024). NTT reduces the cost of multiplication to a quasi-linear complexity of \(n\log n\). The multiplication of two \((n-1)\)-degree polynomials can be calculated using Equation (5):
\[
a * b = \text{NTT}_{2n}^{-1}(\text{NTT}_{2n}(a), \text{NTT}_{2n}(b))
\]
where “\(*\)” represents point-wise multiplication.

If the Equation (5) is used for multiplication, prior to NTT the coefficients of the input polynomials should be doubled \((2n)\) by zero-padding. Moreover, if the multiplication is done over a polynomial field of the form \(\mathbb{Z}_p[x]/(f(x))\) the resulting polynomial of the multiplication with \(2n\) coefficients should be reduced back into a polynomial with \(n\) coefficients \((n-1)\) degree with respect to the reduction function \(f(x)\).

When the reduction function is \((x^n + 1)\), NTT is referred to as a Fermat Theoretic Transform (FTT). In this case a special mathematical property increases the efficiency of using FTT further [10]. Equation (6) shows the case of FTT.

\[
\tilde{c} = \text{NTT}_n^{-1}\left(\text{NTT}_n\tilde{a}, \text{NTT}_n\tilde{b}\right)
\]
where \(\tilde{c} = \omega\) and
\[
\tilde{a} = (\varphi^0a_0, \varphi^1a_1, \ldots, \varphi^{n-1}a_{n-1})
\]
\[
\tilde{b} = (\beta_0, \varphi^1\beta_1, \ldots, \varphi^{n-1}\beta_{n-1})
\]
\[
\tilde{c} = (c_0, \varphi^1c_1, \varphi^2c_2, \ldots, \varphi^{n-1}c_{n-1})
\]
when the reduction function is \((x^n + 1)\) the NTT operations can be directly applied to a polynomial with \(n\) coefficients without zero-padding to a polynomial of \(2n\) coefficients. However, in this case the coefficients of the polynomial should be multiplied by \(\varphi\) prior to NTT and the resulting vector should be multiplied by the powers of \(\varphi^{-1}\) to convert \(\tilde{c}\) back to \(c\). While these operations add \(2n\) multiplications, the NTT size is halved and the reduction operation is eliminated.

B. Memory-Efficient NTT Algorithm

Pseudo-code for the NTT is given in Listing 1 [11]. There are \(\log n\) stages, and at each stage a total of \(n\) values are evaluated. At each iteration of the inner loop two values are generated as in a Cooley-Tukey (CT) radix-2 butterfly [12], hence it takes \((n/2)\log n\) iterations to complete the NTT operation of a polynomial with \(n\) coefficients. The algorithm in Listing 1 is memory-efficient compared to the algorithm used in [9] since the value \(\omega\) is generated inside the inner-loop (line 13).

\textbf{Input} : modulus \((p),\) size of the transformation \((n = 2^k, k \in \mathbb{N})\), primitive root of unity degree \(n\) \(\omega \in \mathbb{Z}_p\), coefficients of polynomial degree \(n - 1\) \((a \in \mathbb{Z}_p[x])\)
\textbf{Output} : NTT of \(a\) \((A \in \mathbb{Z}_p[x])\)

1. \(A = \text{bitreverse}(a)\)
2. for \(i = 0\) to \(\log_2(n) - 1\)
3. \(\text{temp}_\omega = 1\)
4. \(\text{final}_\omega = 1\)
5. for \(j = 0\) to \(2^i - 1\)
6. for \(t = 0\) to \((n/2^{i+1}) - 1\)
7. \(\text{index}_1 = ((t) * 2^{i+1}) + j\) \{index\}
8. \(\text{index}_2 = \text{index}_1 + 2^i\) \{gen\}
9. \(c = A(\text{index}_1);\) \{data\}
10. \(d = A(\text{index}_2);\) \{load\}
11. \(\text{CT}_\text{Out}_1 = (c + \text{final}_\omega * d) \% p\) \{CT Butterfly\}
12. \(\text{CT}_\text{Out}_2 = (c - \text{final}_\omega * d) \% p\) \{store\}
13. \(A(\text{index}_1) = \text{CT}_\text{Out}_1\) \{data\}
14. \(A(\text{index}_2) = \text{CT}_\text{Out}_2\) \{store\}
15. \(\text{temp}_\omega = (\text{temp}_\omega * \omega) \% p\)
16. \{IMOG\}
17. \(\text{final}_\omega = \text{temp}_\omega\)
18. \{end\}
19. \{end\}

Listing 1: Pseudo-code of memory-efficient NTT algorithm

Fig. 2 shows the high-level block diagram of the memory-efficient NTT algorithm and the algorithm used in [9]. Index generation (Index Gen) calculates the index values for each iteration. Data Load uses the calculated index values to generate required data. Iterative Multiplicative Operand Generator (IMOG) produces the \(\text{final}_\omega\) values. CT Butterfly executes the CT radix-2 butterfly and Data Store stores the results of each iteration. The algorithm in Listing 1 uses IMOG to produce \(\text{final}_\omega\) values whereas an additional Index Gen and Data Load is used in [9].

After the NTT of two vectors are calculated, point-wise multiplication and the inverse NTT should also be computed. In the case of FTT, additional operations such as multiplication by the powers of \(\varphi\) and \(\varphi^{-1}\) as defined in Equation (7) should also be computed. 
III. PROPOSED HARDWARE ARCHITECTURES

The proposed hardware architectures perform multiplication over the ideal \( \mathbb{Z}_p[x]/(x^n + 1) \) with the modulo \( p = 65537 \), and can support input polynomials for various sizes of \( n \) by using a fully sequentialized architecture. There are 6 consecutive system tasks executed by the hardware:

1. Multiplication of the coefficients of input polynomial \( a \) and \( b \) by the powers of \( \varphi \) as illustrated in Equation (7)
2. NTT using the memory-efficient NTT of Listing 1 with the powers of \( \omega \)
3. Point-wise multiplication of the coefficients of two transformed polynomials
4. NTT using the memory-efficient NTT of Listing 1 with the powers of \( \omega^{-1} \)
5. Multiplication of the coefficients of the resulting polynomial by the powers of \( \varphi^{-1} \) as illustrated in Equation (7)
6. Final multiplication of all the coefficients with the constant \( n^{-1} \mod p \)

The key ideas are to calculate the powers of \( \varphi, \omega, \omega^{-1} \) and \( \varphi^{-1} \) on-the-fly during the 1st, 2nd, 4th, and 5th system tasks respectively, and to reduce BRAM access rate and simplify address generation during the 2nd, 4th system tasks by improving memory organization.

We propose two hardware architectures, 2DSP and 3DSP, that can significantly reduce the number of occupied slices, BRAMs as well as BRAM accesses. Both of these architectures use one dedicated multiplier to perform multiplication operations during system tasks. The main difference between the 2DSP and 3DSP architecture is that 2DSP uses one dedicated multiplier to implement on-the-fly multiplicative operands, whereas 3DSP improves the maximum operating frequency by using two dedicated multipliers. The optimizations used for low-cost and area efficiency are essentially the same for both architectures.

A. On-the-fly Iterative Multiplicative Operand Generation

The key observation is that during system tasks there are sequential multiplications of the coefficients by the powers of \( \varphi, \varphi^{-1}, \omega \) and \( \omega^{-1} \). During the execution of the 1st system task, the first coefficients \( a_0, b_0 \) will be multiplied by \( \varphi^0 \), the second coefficients \( a_1, b_1 \) by \( \varphi^1 \), and so on, up to the multiplication of the last coefficients by \( \varphi^{n-1} \). Instead of storing these values inside a ROM as in [9], the hardware architectures proposed in this paper calculate these values on-the-fly as given by the schedule in Table I.

The input polynomial coefficients are supplied to the hardware in an interleaved fashion, and \( \varphi \) operands are generated by the multipliers on-the-fly. As scheduled in Table I, the next \( \varphi \) operand should be generated in two clock cycles. If only one multiplier is used, the execution of a multiplication and a modulo-\( p \) reduction should be completed in two clock cycles. This becomes the critical path of the hardware during synthesis and place-and-route operations, and an additional multiplier can be used to increase the maximum operating frequency. If two multipliers are used for generating the operands, one of them is used for generating even powers of \( \varphi \) while the other one is used for generating the odd powers of \( \varphi \). Using this scheme, each multiplier can generate the next multiplicative operand in 4 clock cycles which shortens the critical path.

The scheduling of the 2nd, 4th and 5th system tasks uses the same schedule in Table I. Powers of \( \omega, \omega^{-1} \) and \( \varphi^{-1} \) are generated instead of \( \varphi \). However in the 4th and 5th system task there is only one polynomial to be evaluated. This introduces a bubble in every odd clock cycle at the multiplication step of this schedule. To improve schedule utilization, the bubble in the 5th task is filled in by multiplying the results of the \( \varphi^{-1} \) multiplications by the constant \( n^{-1} \mod p \) through interleaving the execution of the 5th and 6th system tasks. Section IV elaborates on the cost of scheduling bubbles.

<table>
<thead>
<tr>
<th>Clock Cycle(cc)</th>
<th>cc 0</th>
<th>cc 1</th>
<th>cc 2</th>
<th>cc 3</th>
<th>cc 4</th>
<th>cc 5</th>
<th>cc 6</th>
<th>cc 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input -2DSP</td>
<td>( a_0 )</td>
<td>( b_0 )</td>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( a_3 )</td>
<td>( b_3 )</td>
</tr>
<tr>
<td>Input -3DSP</td>
<td>( a_0 )</td>
<td>( b_0 )</td>
<td>( a_1 )</td>
<td>( b_1 )</td>
<td>( a_2 )</td>
<td>( b_2 )</td>
<td>( a_3 )</td>
<td>( b_3 )</td>
</tr>
<tr>
<td>Multiplicative Operand</td>
<td>( \varphi^0 )</td>
<td>( \varphi^0 )</td>
<td>( \varphi^1 )</td>
<td>( \varphi^1 )</td>
<td>( \varphi^2 )</td>
<td>( \varphi^2 )</td>
<td>( \varphi^3 )</td>
<td>( \varphi^3 )</td>
</tr>
<tr>
<td>Multiplication</td>
<td>( a_0 / \varphi^0 )</td>
<td>( b_0 / \varphi^0 )</td>
<td>( a_1 / \varphi^1 )</td>
<td>( b_2 / \varphi^2 )</td>
<td>( a_2 / \varphi^2 )</td>
<td>( b_3 / \varphi^3 )</td>
<td>( a_3 / \varphi^3 )</td>
<td>( b_3 / \varphi^3 )</td>
</tr>
<tr>
<td>Next Operand Generation -2DSP</td>
<td>( \varphi^1 * \varphi^1 )</td>
<td>( \varphi^1 * \varphi^1 \mod p )</td>
<td>( \varphi^2 * \varphi^2 \mod p )</td>
<td>( \varphi^3 * \varphi^3 \mod p )</td>
<td>( \varphi^4 * \varphi^4 \mod p )</td>
<td>( \varphi^5 * \varphi^5 \mod p )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next Operand Generation -3DSP</td>
<td>( \varphi^2 * \varphi^2 )</td>
<td>( \varphi^2 * \varphi^2 \mod p )</td>
<td>( \varphi^3 * \varphi^3 \mod p )</td>
<td>( \varphi^4 * \varphi^4 \mod p )</td>
<td>( \varphi^5 * \varphi^5 \mod p )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B. Improving Memory Organization via Unified BRAM

The classic CT Butterfly architecture is used in the NTT of polynomial coefficients. At each execution of the CT Butterfly, 2 coefficients should be read out from the BRAM and also 2 output coefficients should be written back to the BRAM. This corresponds to 4 BRAM accesses at each clock cycle. Moreover, this also requires the generation of 2 read and 2 write addresses. Even with the BRAMs configured as dual-port, the 4 accesses require duplication of each BRAM. Performance of the CT Butterfly is limited by BRAM bandwidth.

The key observation is that most of the system tasks will be performed on the same coefficients of two polynomials. A coefficient pair of two polynomials \( \{a_i, b_i\} \) will be multiplied with the same power \( \omega^i \), then that pair will be transformed using the same powers of \( \omega \) and the pair will also be pointwise multiplied. Instead of storing the coefficients of two polynomials in separate BRAMs and executing these system tasks on the first polynomial and then to the second polynomial as in \[9\], we can store two coefficient pairs of the polynomials in the same address and apply execution of the system task in a concurrently-interleaved fashion. Xilinx BRAM width can be configured up to 36-bits by treating parity bits as data bits. Since coefficients are 17-bits (modulo \( 2^{17} + 1 \)), 34-bits of data can be stored inside one address of the BRAM. This allows reading or writing two coefficients using only one BRAM access, and also improves coalescing.

As a result, system tasks can be executed in a concurrently-interleaved fashion for the polynomial \( a \) and polynomial \( b \) as scheduled in Table I for the 1st system task or in Table II for the 4th system task.

A detailed overview of the proposed hardware architecture is given in Fig. 3. Datapath implements the CT Butterfly operation during the 2nd and 4th system task and multiplications during other system tasks. IMOG generates the powers of \( \omega^k, \omega^l, \omega^{-1}, \varphi^{-1} \). The Coefficient BRAM stores the coefficients of the polynomials. There are \( n \) addresses and at each address 34-bits (\( 2 \log_2 p \)) of data is stored in the Coefficient BRAM. At any time a read or write access is made to the address \( i \), the concatenated \( \{a_i, b_i\} \) is read or written. Read Data Scheduler (RDS) unpacks the data read from the Coefficient BRAM and supplies data to Datapath, whereas Write Data Scheduler transfers the output generated by the Datapath to Coefficient BRAM. The System Controller (SC) orchestrates the whole process depending on the currently executed system task. 2DSP architecture uses only one multiplication-reduction block whereas the gray shaded region in Fig. 3 is additionally used by the 3DSP architecture.

There is a feedback loop in IMOG and in order to calculate the next exponent, at first, the current exponent should be generated. During the last stage of the NTT and during the 1st and 5th system tasks, a new multiplicative operand should be generated in a maximum of two clock cycles. This causes the feedback loop consisting of a multiplication and a reduction modulo \( p \) to complete its execution in a maximum of two clock cycles, which becomes the critical path of the hardware. In order to achieve higher operating frequencies, an additional multiplication-reduction block could be used in IMOG to increase the execution time to 4 clock cycles. If used, one of the blocks generates odd exponents while the other generates even exponents as given in Table I.

Table I: Simplified Pipeline of NTT Operations

<table>
<thead>
<tr>
<th>( Cc )</th>
<th>( Cc )</th>
<th>( Cc )</th>
<th>( Cc )</th>
<th>( Cc )</th>
<th>( Cc )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read ( {a_i, b_i} )</td>
<td>Read ( {a_i, b_i} )</td>
<td>Read ( {a_i, b_i} )</td>
<td>Read ( {a_i, b_i} )</td>
<td>Compute ( {a_i, A_i} )</td>
<td>Compute ( {b_i, B_i} )</td>
</tr>
<tr>
<td>Compute ( {a_i, A_i} )</td>
<td>Compute ( {b_i, B_i} )</td>
<td>Compute ( {a_i, A_i} )</td>
<td>Compute ( {b_i, B_i} )</td>
<td>Compute ( {a_i, A_i} )</td>
<td>Compute ( {b_i, B_i} )</td>
</tr>
<tr>
<td>Write ( {a_i, b_i} )</td>
<td>Write ( {a_i, b_i} )</td>
<td>Write ( {a_i, b_i} )</td>
<td>Write ( {a_i, b_i} )</td>
<td>Write ( {a_i, b_i} )</td>
<td>Write ( {a_i, b_i} )</td>
</tr>
</tbody>
</table>
The Datapath is shared for all system tasks and consists of one adder, one subtractor, one multiplication and one modulo $p$ operator. Adder and subtractor is only used for the calculation of the inner loop of NTT in Listing 1 during 2nd and 4th system tasks, in other system tasks RDS feeds one input of the adder and subtractor as constant zero so that it won’t have any effect.

SC issues the selection of the MUXs depending on the current executed system task as given in Table III. In 2DSP architecture there are 3 MUXs whereas in 3DSP architecture 4 MUXs are used. SC selects which multiplicative operand to be iterated and which values to be multiplied by the Datapath. During the 1st, 2nd, 4th and 5th system tasks the powers of $\psi^1, \omega^1, \omega^{-1}, \psi^{-1}$ are iterated by IMOG and fed to the Datapath. During the 3rd system task, the coefficients stored in the BRAM are point wise multiplied thus the values of RDS are forwarded to Datapath. During the 6th system task MUX3 selects the constant $n^{-1} \mod p$. While executing the 2nd and 4th system tasks, the output of IMOG should not always be directly used as specified in Listing 1. Generated IMOG values are fed to Datapath only after the loop count of the most inner loop is reached.

Fig. 4 shows the memory allocation and data transfer for read/write operations of all system tasks. During the 1st, 2nd and 3rd system tasks, two coefficients can be read or written using one access and the BRAM is 100% utilized. After the 3rd task BRAM utilization is 50% since coefficients of only one polynomial left for processing.

C. Simplified Address Generation

The proposed hardware architectures uses a simple dual port unified BRAM to store all the coefficients of polynomials. Since there is only one address generated for read signal and this is delayed and/or bitreversed and sent to write address. Moreover, the proposed hardware architecture does not require an address generation to read the powers of $\psi$ and $\omega$ since they are generated on-the-fly. Therefore, the resulting implementation significantly reduces the control and address generation logic compared to [9]. This will be quantified further in Section IV.

### Table III. MUX Selections issued by the System Controller

<table>
<thead>
<tr>
<th>Mux</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mux1</td>
<td>$\psi^1$ &amp; $\omega^1$ &amp; X &amp; $\omega^{-1}$ &amp; $\psi^{-1}$ &amp; X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mux2</td>
<td>IMOG &amp; IMOG,R &amp; RDS &amp; IMOG,R &amp; IMOG &amp; RDS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mux3</td>
<td>$a,b$ &amp; RDS &amp; RDS &amp; RDS &amp; RDS &amp; $n^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mux4</td>
<td>$\omega^1$ &amp; $\omega^{-2}$ &amp; X &amp; $\omega^{-1}$ &amp; X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* $X$ is don't care

Fig. 4. Memory allocation and data transfer

Figure 5. Area comparison of hardware architectures

### IV. IMPLEMENTATIONS RESULTS

The proposed hardware architectures are implemented in VHDL. The VHDL RTL codes are synthesized to the Spartan-6 LX100 FPGA with a speed grade -3 using XST. The resulting netlists are placed and routed to the same FPGA using SmartExplorer in ISE 13.3. To make a fair evaluation and comparison to [9] we target the same FPGA and use the same version of XST and ISE.

The area comparison of the proposed hardware architecture and [9] is given in Fig. 5. The proposed hardware architectures reduce the number of occupied BRAMs by 66% to 80%, and the number of occupied slices by 52% to 67% depending on the value of $n$. As $n$ increases the increase in used slices is negligible but BRAM usage increases linearly and becomes the limiting-factor, for low-cost implementations. The proposed methods reduces the BRAM usage significantly and makes the resource utilization more balanced compared to [9] at the cost of using more DSPs. The proposed architectures fits into the smallest Spartan-6 FPGA even for the largest $n$ value.

The total clock cycles required to execute one multiplication is $2n(\log n) + 7n$. It is important to observe that the proposed architectures complete one multiplication
in more clock cycles than [9] due to bubbles introduced at the NTT schedule during 4th system task. As shown in Fig.6, this could be compensated since smaller architectures also yields to an increased maximum operating frequency because routing delays are lower than a bigger design.

Fig.6 shows that the proposed hardware architectures are much more efficient than prior low-cost implementations. The 3DSP version requires an additional DSP resource and has a negligible slice increase as shown in Fig. 5, while enabling much higher maximum operating frequencies by shortening the critical path. The proposed 3DSP architecture even outperforms [9] for some n values.

The proposed hardware does one read and one write operation for each iteration of the inner loop of NTT which is the most BRAM access heavy operation, while 5 memory accesses (2 read, 2 write access to Coefficient BRAM and one read access to ROM) is required in [9]. This corresponds to approximately 60% BRAM access reduction.

One limitation of the proposed memory organization is the 18-bit limit on the modulo prime p. However, Xilinx 7 Series FPGAs allow BRAM widths of up to 72 bits, allowing up to 36-bit p values.

High-throughput hardware architectures are presented in [7,8]. Györfi et al. [7] use a fully parallelized hardware and the results are generated in O(1) time with O(log n) latency. However a fair comparison is not feasible because the hardware architecture is implemented on a big Virtex-5 using 3,639 slices and 68 BRAMs while the selected n and p values \((n = 64, p = 257)\) are much smaller than what we target. Göttert et al. present an architecture that implements NTT in O(log n) time [8]. The proposed hardware is implemented on a Virtex-7 which costs orders of magnitude higher than our target FPGA. Moreover, the architecture LUT and FF usage is at the order of 100Ks while selected n and p values \((n = 256, p = 7681)\) are much smaller than what is used in the proposed architectures.

V. CONCLUSION

The interest in lattice-based cryptography is increasing due to its quantum-resistance and its provable security for worst-case hardness assumptions. We present methods for area optimizations to the most critical and computationally-intensive operation in lattice-based cryptography: polynomial multiplication with the Number Theoretic Transform (NTT). The proposed methods are implemented on an FPGA for polynomial multiplication over the ideal \(\mathbb{Z}_p[x]/(x^n + 1)\). Compared to prior work with similar performance, the proposed architectures significantly reduce slice utilization, number of BRAMs and BRAM accesses. The proposed hardware architectures are implemented in VHDL and can even fit into Xilinx’s smallest Spartan-6 FPGA.

REFERENCES