Acceleration of core post-quantum cryptography primitive on open-source silicon platform through hardware/software co-design*

Emma Urquhart[1] ✉[0009−0005−9235−5320]
eu233@cam.ac.uk
and Frank Stajano[1][0000−0001−9186−6798]
frank.stajano@cl.cam.ac.uk

University of Cambridge, Cambridge (United Kingdom)

Abstract. Post-Quantum Cryptography (PQC) algorithms are currently being standardised and their early implementations are not as efficient as the well-established public key cryptography (PKC) algorithms that have benefited from decades of optimisations. We report on our efforts to accelerate the Number Theoretic Transform (NTT), the most computationally expensive primitive in the Kyber (ML-KEM) and Dilithium (ML-DSA) PQC algorithms selected by NIST for standardisation. Our target platform is the OpenTitan Big Number Accelerator (OTBN), part of the first open-source silicon root-of-trust chip. We implemented the Kyber NTT in OTBN assembly, using only the existing instructions, and identified its bottlenecks. We then restructured the code to exploit parallelism and defined additional assembly instructions for the open-source co-processor that would enable execution of our vectorised implementation. Our hardware/software co-design approach yielded a significant performance improvement: NTT ran 21.1 times faster than the baseline implementation which used only OTBN's existing instructions. Our approach fully leverages the potential for parallelism and maximally exploits the existing capabilities of OTBN. Some of our optimisations are fairly general and might be successfully applied to other contexts, including accelerating other algorithms on other platforms.

Keywords: OpenTitan · Open-Source Hardware · Post-Quantum Cryptography · Number Theoretic Transform · Kyber · ML-KEM · Performance Optimisation · Hardware/Software co-design

1 Introduction

In the realm of Internet of Things (IoT) security, the ever-increasing ubiquity and connectivity of mobile devices presents challenges and opportunities. Public Key Cryptography (PKC) algorithms such as Rivest-Shamir-Adleman (RSA)

* Authors' preprint of 2024-07-11. To appear in *Proc. International Conference on Cryptology and Network Security (CANS 2024), Springer LNCS.*

[13] and Elliptic Curve Cryptography (ECC) [9] are used to ensure the confidentiality and integrity of communication channels. Execution of these algorithms can be expensive on resource-constrained devices and therefore many are equipped with specialized cryptographic co-processors. In February 2024, the first commercially-available open-source silicon root of trust was released by the OpenTitan [7] consortium stewarded by lowRISC. This chip marks a significant development in secure and transparent hardware. It incorporates the OpenTitan Big Number Accelerator (OTBN), a cryptographic co-processor specifically designed for use in IoT devices [8].

The advent of quantum computing poses an impending threat to the public-key cryptosystems that are integral to the security of communications between these devices. The mathematical problems on which algorithms such as RSA and ECC rely include factorization of large integers and the discrete logarithm problem. These tasks are computationally infeasible for classical computers, but can be solved in polynomial time by a quantum adversary, as proven by Shor[15]. To mitigate this threat, Post-Quantum Cryptography (PQC) provides solutions that can be implemented on classical hardware but are capable of withstanding quantum attacks. The US National Institute for Standards and Technology (NIST) is conducting a standardization process for post-quantum cryptosystems, currently in its fourth round.

Lattice-based cryptography is emerging as a promising approach to PQC, encompassing three of the four algorithms already selected for standardisation: CRYSTALS-Kyber (ML-KEM) [3], a key encapsulation mechanism, and two digital signature algorithms, CRYSTALS-Dilithium (ML-DSA) [4] and Falcon [5]. Implementation of these algorithms, particularly on resource-constrained devices, poses practical challenges as their core operations incur significant overheads on existing platforms. Modern cryptographic co-processors such as OTBN are optimised for execution of RSA and ECC, providing large integer arithmetic capabilities. However, lattice-based PQC algorithms do not involve such operations and therefore do not benefit from the capabilities of these co-processors. OTBN is not yet specialised for PQC. We propose realistic and low-cost hardware improvements to OTBN that can be leveraged to significantly enhance the performance of PQC on this platform.

Due to the predicted prevalence of ring-based lattice schemes, we focus on a characteristic bottleneck of such schemes: polynomial multiplication. We investigate acceleration of polynomial multiplication via the number theoretic transform (NTT) in Kyber. The target platform is OTBN. The proposed instructions and software implementations leverage OTBN's existing capabilities and incorporate viable architectural extensions. Our main contributions include:

- Baseline implementation of the NTT in Kyber using the existing OTBN instruction set, replicating the C reference implementation as closely as possible.
- Detailed analysis of the bottlenecks of the baseline implementation and current limitations of OTBN in the context of efficient execution of NTT.

- Set of 8 additional assembly instructions for acceleration of NTT on OTBN and estimations of cycle cost for each instruction.
- A novel, vectorized implementation of the NTT in Kyber for OTBN, which leverages the new instructions and achieves a reduction in cycle cost from 92,074 to 4,356 cycles compared to the baseline (21.1× improvement factor).

Our approach for identifying the bottlenecks and our ideas on how to speed them up might also serve as inspiration for researchers seeking to accelerate other algorithms on other platforms. Our code is publicly available at: https://github.com/emmau678/opentitan/tree/mphil_thesis_pqc_acceleration.

2 Background

2.1 OpenTitan Big Number Accelerator

OpenTitan is the first open-source silicon root of trust (RoT) project, aimed at creating a high-quality reference for transparent and secure silicon. The first OpenTitan chips, based on the Earl Grey discrete RoT, reached commercial availability in February 2024. OTBN, a co-processor for acceleration of asymmetric cryptography, serves as a fundamental hardware security primitive. Its large data path and specialized instruction set enhance the efficiency of classical PKC algorithms. However, OTBN is not yet optimised for PQC.

OTBN features a 32-bit wide control path and a 256-bit wide data path, each containing 32 registers. Its security-centric design incorporates a reduced instruction set which comprises a base subset for control flow and a big-number subset for wide-integer arithmetic in data flow. The separation of paths reduces the risk of data leakage. OTBN supports data integrity protection and secure wipe of internal states. Cryptographic security is further enhanced by its internal random number generation mechanism which is connected to the Entropy Distribution Network. OTBN implements two dedicated memories of 4 kiB: instruction memory (IMEM) and data memory (DMEM), separated to bolster security.

2.2 The Number Theoretic Transform

Polynomial multiplication is one of the most performance-critical implementation elements of lattice-based PQC schemes such as Kyber and Dilithium. The NTT is commonly adopted for this purpose, enabling a reduction in complexity of polynomial multiplication from $O(n^2)$ to $O(n)$. It is a specialised form of the Discrete Fourier Transform, which operates on the finite field $\mathbb{Z}_q$ instead of complex numbers. The NTT operates by transforming polynomials into a domain in which multiplication is highly efficient. The multiplications are performed within the NTT domain before the results are transformed back into the normal domain using the inverse number theoretic transform (INTT). The NTT transforms polynomials from their coefficient representation to their point-value form. Multiplication of two polynomials in their point-value form is a straightforward

pointwise product. Given polynomials f and g, we compute their product according to Equation 1, where $\circ$ denotes multiplication within the NTT domain:

$$f \cdot g = INTT(NTT(f) \circ NTT(g)) \tag{1}$$

The operation of the NTT is given in Equation 2. This equation describes the transformation on a polynomial g of degree n, where $g = \sum_{i=0}^{n-1} g_i X^i$ and $g_i \in \mathbb{Z}_q$. In this equation, ω represents the primitive n-th root of unity, where $\omega^n \equiv 1 \mod q$ and, for any $1 \leq k < n$, $\omega^k \not\equiv 1 \mod q$. Values of ω^{ij} are known as twiddle factors. Multiplication by these values is equivalent to evaluating the polynomial at powers of the n-th root of unity. The INTT reverses this transformation.

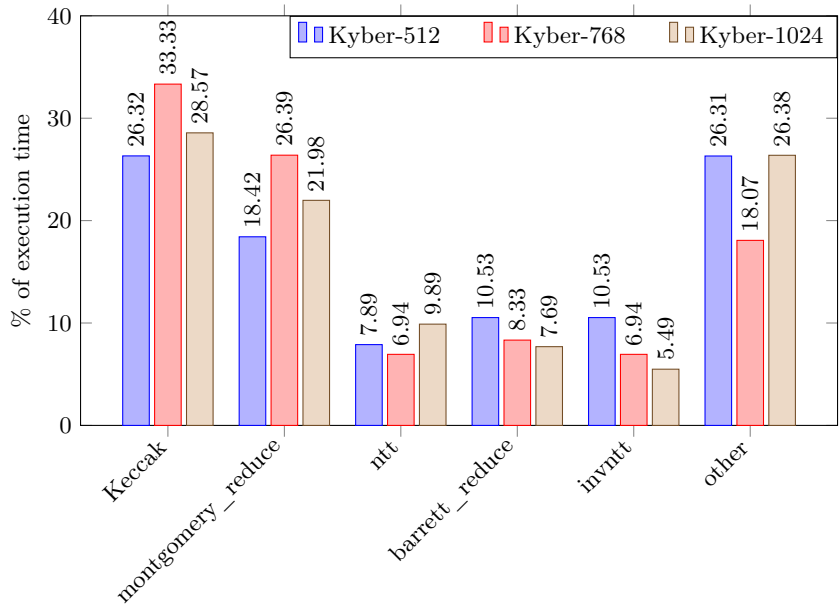
$$\hat{g} = \mathrm{NTT}(g) = \sum_{i=0}^{n-1} \hat{g}_i X^i, \quad \text{with} \quad \hat{g}_i = \sum_{j=0}^{n-1} g_j \omega_n^{ij} \pmod q. \tag{2}$$

2.3 Profiling the reference Kyber C implementation

After selecting Kyber as the focus of our optimisation efforts, we conducted an analysis of the reference C implementation to inform our choice of algorithmic components to target for acceleration. Two implementations of Kyber are available within the official repository [12]: a platform-agnostic implementation and an optimised AVX2 implementation. The optimised implementation may be run on processors that support the AVX2 instruction set. AVX2 offers capabilities for signed and unsigned processing of high and low parts of packed values within SIMD registers. These processors also offer out-of-order execution, meaning that instructions can be interleaved. Our target platform is OTBN, which has a restricted instruction set, does not support advanced extensions like AVX2 and does not support out-of-order execution. Therefore, the platform-independent implementation was the most suitable reference point for our work. Given that OTBN is not yet equipped with a compiler, the reference C code cannot be directly excuted on the platform. We obtained the profiling results by executing the reference code on a regular laptop (Core i7 processor). We assume an approximate equivalence in terms of distribution of computational effort within the Kyber algorithm between the reference C code and a full OTBN implementation.

Three executables are generated for each parameter set (512, 768 and 1024) by compiling the test program; `test_kyber$ALG`, `test_kex$ALG` and `test_vectors$ALG`, where `$ALG` identifies the parameter set. According to the repository documentation, `test_kyber$ALG` runs 1,000 tests which encompass key generation, encapsulation and decapsulation. We used `test_kyber$ALG` for profiling. The parameter sets correspond to the different security levels of Kyber. We obtained results for all three security levels. The algorithm remains the same for each security level; all that changes are parameter values. We generated a flat profile for each parameter set, i.e. a table that captured the total amount of time spent in the execution of each function. A visualisation of the percentage of execution time spent within each function is presented in Figure 1.

Fig. 1. Profiling results for reference Kyber implementations



From the data in Figure 1, we firstly note that the distributions across the parameter sets are similar, which aligns with expectations given that the code is the same. The Keccak function is the most computationally intensive component of the algorithm. However, the Keccak core is most conducive to acceleration in pure hardware, as it was designed as a hardware-oriented implementation of the SHA-3 hashing algorithm [2]. The purpose of our research is to investigate acceleration of Kyber on OTBN through hardware/software co-design. Therefore, while a custom Keccak accelerator could likely be integrated as a hardware extension, it is not the ideal candidate for optimisation via instruction set extensions which do not aim to drastically alter the hardware architecture.

The four next most expensive functions are `montgomery_reduce`, `barrett_reduce`, `ntt`, and `invntt`. It is important to note that the `montgomery_reduce` function is called from both `ntt` and `invntt`, while the `barrett_reduce` function is called from the `invntt` function. Algorithmically, the `ntt` and `invntt` functions are closely (inversely) related and hence share a number of similarities. As a result, it appeared likely that it would be possible to design certain optimisations to target both functions. Because `montgomery_reduce` is a component of the `ntt` function, it was also targeted in our optimisation strategies. We implemented and accelerated the NTT using acceleration techniques and new instructions which we believe would contribute similar performance improvements to an implementation of the INTT.

3 Methodology

3.1 Development environment and testing infrastructure

During our development work, OTBN was still being taped out as an engineering sample. A Python simulator for OTBN is available as part of the OpenTitan repository and this was used for development. The simulator is cycle-accurate for all existing OTBN instructions, which execute in either one or two CPU cycles. In the case of designing instruction set extensions, it was necessary to consider the required hardware modifications to estimate cycle counts. The instructions we propose are designed to maximally leverage OTBN's existing hardware components and require only minor modifications. This not only facilitates an easily implemented and low-cost solution that enhances performance, but ensures that cycle count estimates are aligned with the ground-truth performance of existing instructions. A C compiler is not yet available for OTBN, so all code was written in OTBN assembly.

Correctness of implementations was validated by extending the testing infrastructure within the simulator. The repository contains a simple testing framework for sample instructions. An input assembly file and a corresponding file containing expected output values is provided for each test case. We integrated a Python script that dynamically creates input and output files for customisable input ranges and the corresponding outputs generated by function prototypes. We created template files containing placeholders indicating the values to be dynamically overwritten for each input/output pair and used them to automatically create the files for processing. For simple subroutines and instruction tests, we wrote the function prototypes in Python. For increasingly complex implementations, we used `ctypes` to ensure absolute alignment with the reference C implementation of Kyber, as Python's type handling would lead to divergence of results. Once the full implementation of `ntt()` was complete, we implemented a black-box test to run NIST vector tests. We captured the input/output pairs to the `ntt` function by separately running the Kyber reference implementation and storing the state of the input/return array before and after invocation of `ntt()`.

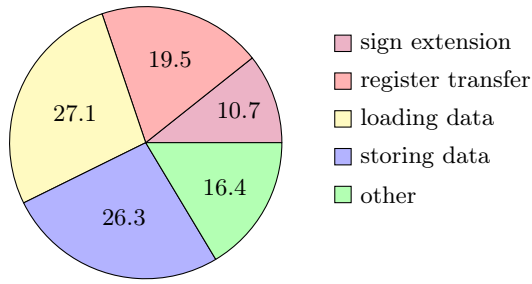
3.2 Identifying the Bottlenecks in the Baseline Implementation

We first developed a baseline implementation of the NTT by porting the code in the Kyber reference implementation as directly as possible to OTBN assembly, using only the existing instructions. This process established the current performance of OTBN in execution of these functions and provided baseline performance benchmarks against which to compare optimisation efforts.

It should be noted that the Kyber reference implementation has not been optimised for any platform. However, the reduced instruction set of OTBN constrains the potential for optimisation without implementing instruction set extensions, so the baseline performance provides a reasonable estimate of its capabilities. Analysis of the baseline implementation on OTBN granted insights into performance bottlenecks and particularly inefficient operations, hence serving

to motivate optimisations and inform the design of instruction set extensions. In Figure 2, we present our analysis of the baseline implementation of NTT on OTBN. Performance bottlenecks are analysed in terms of percentage of overall cycle count.

Fig. 2. % execution time spent on functionality types in NTT baseline



The baseline implementation of NTT required 91,939 CPU cycles to execute. During the implementation process, the most significant performance impediment we noticed was the restriction to scalar computations on the wide data registers (WDRs) of 256 bits. These registers were designed to perform arithmetic on large integers; however, in the case of `ntt`, the integers involved in computation, including intermediate results, do not exceed 32 bits and hence the large register capacity is not utilised. The computational effort spent on these operations is not being exploited to its full potential and the same effects could be achieved by operating on much smaller units. We noted that vectorisation of operations could be maximised to enable full use of WDRs.

Although the majority of instructions in both subsets execute in a single clock cycle, in the context of the Big Number subset, additional operations are required to perform certain computations. OTBN only supports unsigned arithmetic and the NTT function operates on signed values. Two's complement is used to represent negative numbers. Although the values involved in multiplication operations are 16 bits in width, it was necessary to sign-extend these numbers to 64 bits before using `BN.MULQACC`, as it operates on 64-bit operands. This had to be done manually if this data had been transferred from GPRs. In the interest of maintaining a constant-time implementation, we adopted the following approach. The sign bit is isolated through a right shift of 15 bits and multiplied by a 64-bit mask with the upper 48 bits set. The result is then XORed with the original 16-bit value, resulting in sign-extension to 64 bits. This process costs 4 cycles for each of two operands, before a multiplication can be performed. Sign extension required 10.7% of the total cycle count of the `ntt` baseline (2). In order to enhance the efficiency of sign extension, we reduced the operand size in our vectorised implementation. For example, if lanes were 16 bits in length,

this would align with the operand width and therefore sign extension beyond this width would not be necessary.

Another notable bottleneck was the requirement to transfer data between the register types to perform different operations. Certain instructions, such as multiplication, are only available in the Big Number subset. Conversely, other operations, such as left shift, are only available in the base subset. Given that the Big Number subset is designated for data flow, transferring data back and forth between WDRs and GPRs during the main computation is both inefficient and not compliant with standard practice. However, at certain points in the implementation this is necessary, for example, loading array values at fine granularity from data addresses. A resulting objective of our design of new instructions was to minimise the requirement for data transfer between register types. Transferring of data between register types cost 19.5% of the cycle count of the `ntt` baseline (2).

The array `r`, which is processed by the NTT function, consists of 256 16-bit elements which are stored contiguously in memory. Each element is processed individually and reads from memory to GPRs can only be performed on 32-bit-aligned boundaries. Therefore, data can only be loaded in fixed 32-bit blocks. This complicated the element loading and storage procedures. In order to load and operate on `r[j]`, we floor-divide the index `j` by 2 by performing a right shift by 1 bit, in order to identify the index of the 32-bit block containing `r[j]`. We then shift the result right by 2 to compute the byte offset from the base address of `r` from which to load the block. We determine whether index `j` is even or odd by computing `j AND 1`. In the case of an even index, we isolate `r[j]` via an `AND` of the loaded block with a 16-bit mask. In the case of an odd index, we shift the loaded block right by 16 bits. However, this approach contains a conditional statement, which may lead to violation of constant-time properties. In the case of development of the baseline and optimised implementations, we avoided the use of conditional statements in order to retain constant-time properties.

Therefore, we require a single execution path for loads and stores of odd- and even-indexed values. To load and isolate an individual array value (of either odd or even index) in a GPR, we follow the process outlined in Figure 3. We begin by loading a data block containing 2 contiguous array elements following the previous procedure, one of which is at the required index `j`. We compute `j AND 1` and its inverse. We shift both values left by 4 so the non-zero remainder represents 16. We then shift the loaded block right by the former value (`(j AND 1) << 16`), shift left by the same value and finally shift right by the latter value (`(NOT (j AND 1)) << 16`). This method isolates the required element in the least significant position in the case of both odd and even indices, enabling subsequent computations. For storing the result, we only overwrite one element of `r`, leaving the other 16 bits of the 32-bit block unchanged. The opposite 16-bit value in the block is isolated in a similar way to `r[j]`. Before the final block is stored to memory, the two 16-bit components are shifted back to their original position and combined with an XOR. We noted the evidently large computational overhead introduced by replicating the elementwise loading procedure of

the reference implementation. In the `ntt` baseline (2), loading of values into GPRs (including subsequent manipulation of loaded values) cost 27.1% of the total cycles. Meanwhile, constructing the resulting data blocks and storing them to memory cost 26.3% of the total cycles. We aimed to reduce the number of load and store operations, eliminate manipulation of loaded values and facilitate parallel computation on loaded values directly.

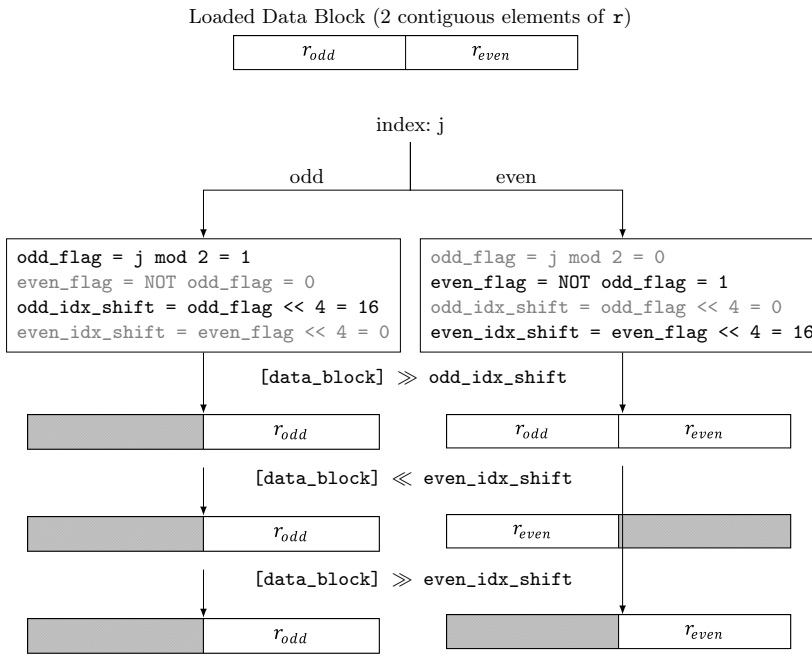


Fig. 3. Loading and isolating data elements in GPRs

4 Vectorised Implementation Design

We designed the vectorised implementation of the `ntt()` function, outlined in pseudocode in Algorithm 1, with the aim of maximising parallelism, minimising load/store overhead, minimising data transfer between register types and more efficiently handling signed multiplication. We designed an implementation that maximises the vectorisation potential of OTBN and enables full usage of the capacity of the WDRs. To implement it, we designed new instruction set extensions that complement the existing capabilites of OTBN and incorporate some moderate hardware modifications, which could realistically be introduced to the

platform. The optimised implementation loads and stores 16 polynomial coefficients at a time, significantly reducing the load/store overhead of the baseline. Array elements are operated on in-place using vectorised instructions, which enables us to avoid manipulation of loaded values. Explicit sign extension is no longer required due to the narrower lane widths.

The values of `zeta` are broadcasted at 32-bit intervals across WDRs before entering the `fqmul` computation. Because the broadcast instruction replicates the value in a GPR across all lanes of a WDR, we must load `zeta` from memory directly into a GPR. We aimed to minimise the overhead of loading and isolating elements. We achieved this by unrolling two iterations of each loop which required a new value of `zeta` to be assigned. This enabled us to load two values at once, isolate them and retain the second one in a separate GPR instead of performing a second load. In the baseline implementation, loading and isolating two values of `zeta` cost 26 cycles, whereas this approach costs only 6 cycles.

The implementation is split into two parts: the first deals with values of `len` which are multiples of 16. This means that the number of elements between `r[j]` and `r[j+len]` can be stored in a distinct number of WDRs. Therefore the new values of `r[j]` and `r[j+len]` can be separately computed and written to memory in batches of 16 elements. Within the `fqmul` function, intermediate values can occupy up to 32 bits. At this point, the lane widths are effectively expanded from 16 to 32 bits. This process is illustrated in Figure 4, where shaded sections of mask registers represent all 0s and non-shaded sections all 1s. Note that shifts applied are vectorised on 32-bit lanes. The lower and upper 16-bit elements in each 32-bit lane of the wide data register are extracted into two separate registers. We isolate the elements of even index by pre-loading a 256-bit mask with every even-indexed 16-bit element set and performing an `AND` operation between this register and the loaded values. Then, to isolate the values at odd indices, a vectorised right shift by 16 bits of each 32-bit lane is used to place them in the lower positions. The `fqmul` operations can then proceed in the same way for both vectors. The results of the two `fqmul` computations are then combined by reversing the shift and performing an XOR between the two registers.

The second part of the implementation deals with values of `len` which are factors of 16. The number of elements between `r[j]` and `r[j+len]` is less than the capacity of a WDR. As data elements are loaded contiguously, computations of the new values of `r[j]` and `r[j+len]` are combined within registers. Iteration levels are merged to maximise computational capacity. This part of the implementation has one less nested loop than the first, merging the loading of zetas into the innermost loop. Then `8/len` zetas are loaded into a single register, occupying equal proportions. Each iteration reads 16 consecutive elements as a vector of `r[j]`. Since loads to wide data registers are only permitted at 256-bit boundaries, OTBN's 512-bit barrel shifter, which produces a 256-bit output, is used to load `r[j+len]` at the required level of finer granularity. The subsequent block of 16 elements is then loaded, concatenated with the previous and shifted right by `len` elements, returning the low 256 bits as the corresponding vector of `r[j+len]`. This construction of the vectors of zetas, `r[j]` and `r[j+len]` allows

the rest of the computation to proceed in the same way as the first part of the implementation. Once `fqmul` has been computed, however, the computations of the new values of `r[j]` and `r[j+len]` must be combined within the same resulting register. This is achieved using bitmasks and shifting to interleave the calculated values at offsets of length `len` within the register. Due to the direct operations on data in WDRs throughout the computation, the overhead of transferring data between register types in NTT is reduced to zero from a cycle count of 17,918 in the baseline (1).

Throughout the vectorised computation, elements are fully packed into the WDRs. The combination of the computation of new values of `r[j]` and `r[j+len]` within the same registers ensures that this potential remains maximised even for values of `len` that are less than the element capacity of a WDR. The expansion of lane widths from 16 to 32 bits is implemented for the smallest possible number of instructions. Once 32-bit precision is no longer required for intermediate computations, the implementation transitions back to the initial mode of operation on 16 elements in parallel. Minimisation of load/store overhead is achieved, as the optimised implementation loads and stores 16 elements at once. Additional pre-processing of loaded array values is eliminated as all loaded values are operated on directly in the positions within the register at which they were loaded. As shown in Table 1, the number of cycles spent on the loading of values and manipulation of loaded values was reduced from 24,947 to 1,196 cycles for NTT (1). Similarly, the cycles required for constructing and storing data blocks to memory was reduced from 24,192 to 592 for NTT. Loading of `zeta` values into GPRs before broadcasting is optimised by isolating and storing the two values that are loaded at once from memory. The multiplication process has been streamlined to eliminate the requirement for explicit sign extension in software. In the baseline implementation of NTT, 9,854 cycles were spent on sign extension, however this costs no additional cycles in the optimised implementations.

Table 1. Comparison of Cycle Count Distribution Between Implementations

Implementation	Sign Extension	Register transfer	Load	Store	Other	Total
`ntt_baseline`	9854	17918	24947	24192	15028	91939
`ntt_optimised`	0	0	1196	592	2568	4356

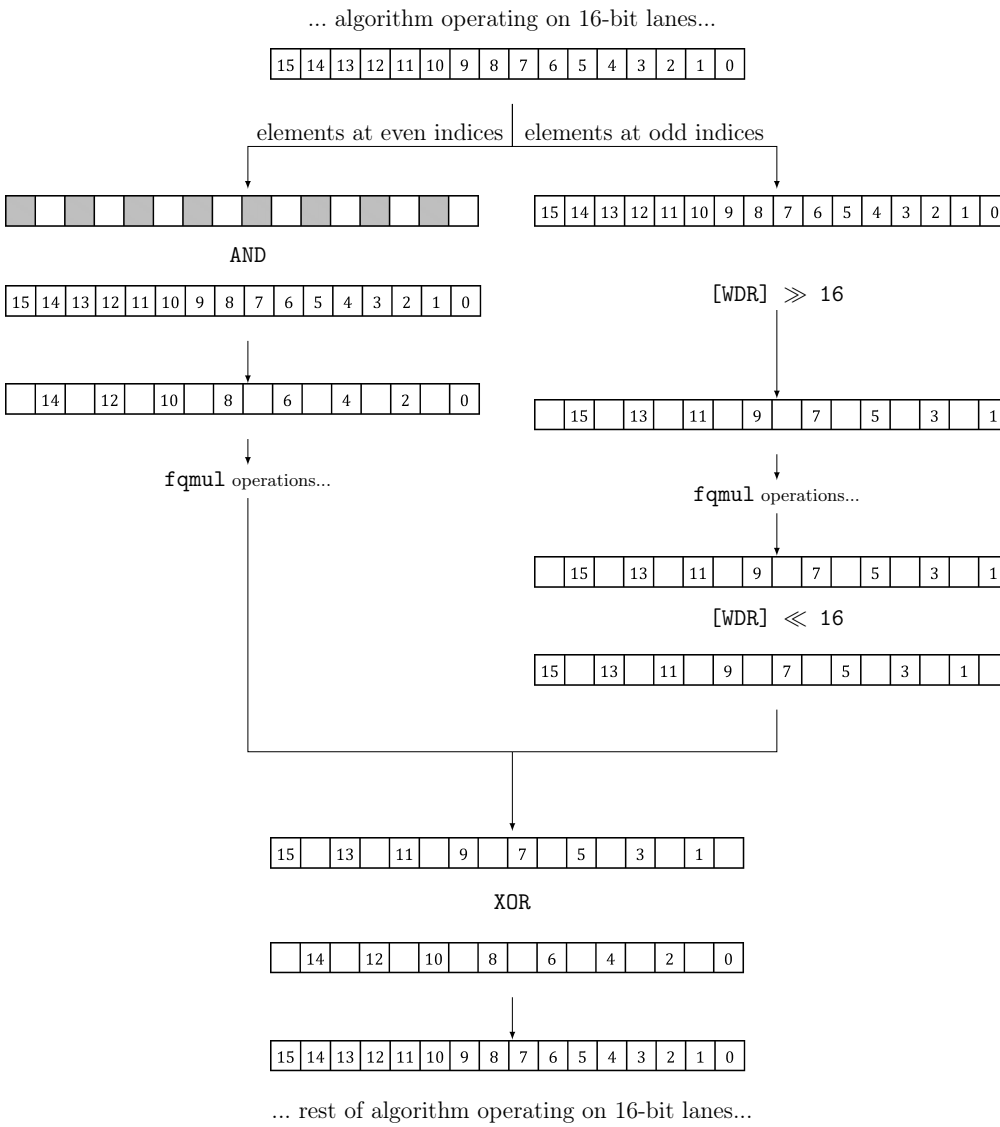


Fig. 4. Expansion of lane width from 16 to 32 bits

Table 2. Instruction Set Extensions

Instruction	Description	Latency (Cycles)
BN.LSHI	Concatenates contents of 2 WDRs and shifts right by an immediate. Truncates to 256 bits.	1
BN.MULVEC	Vectorized signed multiplication on the low 16 bits of each 32-bit lane.	1
BN.MULVEC32	Vectorized multiplication on each 32-bit lane. Truncates to 32 bits.	1
BN.ADDVEC	Vectorized addition on each 16-bit lane.	1
BN.SUBVEC	Vectorized subtraction on each 16-bit lane.	1
BN.RSHIFTVEC	Vectorized right shift on each 32-bit lane.	1
BN.LSHIFTVEC	Vectorized left shift on each 32-bit lane.	1
BN.BROADCAST	Replicate value in selected GPR in each 32-bit lane of WDR.	1

5 Instruction Set Extensions for OTBN

The new instruction set extensions are outlined in Table 1. This section describes the reasoning used to estimate cycle counts, including details of how existing hardware components are leveraged and hardware modifications required.

BN.LSHI: operates in the same way as the existing BN.RSHI instruction for OTBN which executes in one cycle, but performs a left shift instead of right.

BN.MULVEC: Existing 64-bit multiplication units would need to be reconfigured to operate independently on 32-bit segments and truncate results to 32 bits. Due to the shorter combinatorial path for 32-bit lanes, the critical path should execute within the single cycle required for current 64-bit operations.

BN.MULVEC32: The sign bit of each 16-bit input should be isolated (using a simple shift) and replicated across bits 17-31 of each lane using a series of multiplexers. The same logic as in BN.MULVEC can then be followed. Given that the existing BN.MULQACC can multiply and add to an accumulator within a single cycle, it is reasonable to assume single-cycle latency in this case.

BN.ADDVEC and BN.SUBVEC: OTBN features a 256-bit adder, which add/subtract in one cycle. Therefore, it should be possible to perform vectorized addition/subtraction on a 256-bit register within a single cycle.

BN.RSHIFTVEC and BN.LSHIFTVEC: OTBN features a 512-bit barrel shifter that produces a 256-bit output within one cycle. This is more complex than a 256-bit barrel shifter, which would be sufficient for this instruction. Vectorized shifts would be performed across 256-bit registers, in parallel lanes of 32 bits. The combinatorial path of each of these shifts would be much shorter than that of a full 256-bit shift as computations within each lane can be computed independently. Therefore it would execute in one cycle.

BN.BROADCAST: OTBN features a GPR selector multiplexer. The GPR value should be replicated 8 times across the destination WDR, which would

require a simple fanout of wires. The existing WDR input multiplexer would need to be extended by one input to accept data from GPRs. The critical path requirements are relatively simple in the context of the existing instruction set, so this instruction should execute in one cycle.

6 Evaluation and Results

We extensively validated the correctness of our implementations using the gold-standard input/output value pairs generated by NIST test vectors. We captured the expected input and output value pairs upon entry to and exit from the NTT function during execution of the official reference implementation on a regular laptop. We used the `test_vectors$ALG` executable upon compiling the reference C code on Linux. This generates 10,000 sets of test vectors which contain keys, ciphertexts and shared secrets. The `$ALG` variable is used to identify the parameter set, which represents the security level in Kyber. We gathered values for all 3 parameter sets and for each set, tested our implementations in a black-box test setting, comparing expected and generated output values.

Our optimised implementation of NTT demonstrates a 21.1× speed-up over our OTBN baseline. It should be noted that the reference implementation that we translated to OTBN assembly was not optimised for any platform. We did not invest efforts into manual optimisation of the baseline OTBN implementation using the existing instruction set. This may be an avenue for further research, which may more precisely quantify the performance benefits added solely by our new instructions and the optimisations they enable. However, it is worth noting that the limited nature of the existing OTBN instruction set constrains optimisation potential without introducing new instructions.

Given that we implemented our baseline ourselves[1], we sought an additional benchmark to ensure an objective comparison. The most similar platform to OTBN on which we could execute the reference implementation directly was the RISC-V Ibex core. Our optimised OTBN implementation uses 10.7× fewer cycles.

To estimate the performance improvement for the entire Kyber algorithm, we profiled the reference implementation on a regular (Core i7) laptop. As a full implementation of Kyber was not available in OTBN assembly, we made the assumption that the distribution of computational effort within Kyber would be approximately equal across both platforms. Approximately 32% of execution time was spent in the `ntt()` and `montgomery_reduce()` functions. If our optimisations speed up these functions by 21.1×, their contribution will reduce to $32\%/21.1 = 1.5\%$ and the overall runtime will be dominated by the remaining non-accelerated 68%. Thus the overall speed-up for Kyber would be 1.43×, obtained as $T/T' = 100\%/(1.5 + 68)\% = 1.43$.

The cycle counts spent in the execution of each implementation are presented in Table 3. The performance improvement factors achieved by the optimised im-

[1] We had to manually translate the Kyber reference from C to OTBN assembly because a C compiler was not yet available for this new platform.

plementations over the baseline implementations are shown in Table 4. Because the new vectorised implementation required a complete redesign of the algorithm, incremental performance measurements would not have led to meaningful results. Therefore, only the final results are reported for implementations that had been fully functionally verified.

Table 3. Comparison of Implementation Cycle Counts

Implementation	Cycle Count
`otbn_ntt_baseline`	91939
`ibex_ntt_baseline`	46497
`otbn_ntt_optimised`	4356

Table 4. Performance Improvements over Baseline Implementation

Implementation	OTBN Baseline	Ibex Baseline
OTBN Optimised NTT	21.1x	10.7x

The `montgomery_reduce()` function is also invoked by the INTT, so our approach would need to be extended to encompass this function to fully exploit this acceleration. This would also lead to greater performance improvements in the overall algorithmic context.

7 Related Work

Previous research has explored acceleration methods for lattice-based PQC in three categories: pure software, custom hardware and hardware/software co-design.

Software approaches leverage features of modern instruction sets such as fixed-point arithmetic in Armv8-A Neon (Becker et al. [1]) and blend/shuffle instructions in Intel-AVX2 (Seiler [14]). For more limited resource-constrained settings, there is increasing interest in hardware and co-design approaches to acceleration.

A custom polynomial multiplier using bitwise modular reduction is presented by Yaman et al. [16] as an embedded hardware accelerator. In other work, Yaman et al. [11] incorporate a combination of inter- and intra-module pipelining optimizations into a custom hardware module.

Hardware-software co-designs have proposed instruction set extensions for the acceleration of the NTT, with a strong emphasis on extensions to the RISC-V target architecture. Among these, Fritzmann et al. [6] embedded tightly-coupled hardware accelerators into a RISC-V processing pipeline. Nannipieri et al. [10]

presented instruction set extensions based on a post-quantum arithmetic logical unit for acceleration of Kyber and Dilithium on RISC-V.

In contrast, our work tightly aligns with the architecture of OTBN and is designed to propose feasible extensions that could be incorporated without the requirement for custom hardware components.

8 Limitations

As mentioned, the OTBN chip was still being taped out during our development and therefore we could only measure performance through a cycle-accurate simulator. Other potentially useful measurements such as memory and power consumption were not reported by the simulator but it will be interesting to measure them on the physical chip once samples are available. It is possible that performance differences between the simulation and physical deployment may be observed. In addition, certain hardware-specific implementation challenges may arise during physical hardware development, which may not have been fully reflected in the simulated implementation. However, the simulator provided various advantages such as access to an accurate model of the entire OTBN block while the physical chip did not exist yet. It facilitated low-cost and efficient experimentation for early development. The flexibility afforded by the simulator facilitated relatively fast testing of different implementation strategies and enabled us to validate the functionality of our instructions and implementations without access to the physical chip.

9 Conclusions and Future Work

We documented and demonstrated how we achieved a substantial speed-up for the Number Theoretic Transform (an important primitive for lattice-based PQC algorithms including ML-KEM Kyber and ML-DSA Dilithium) by recasting the implementation in vectorized form and incorporating minor architectural modifications to our target processor to enable vectorized processing. Our instruction set extensions allow the programmer to make full use of OTBN's wide data path.

We have open-sourced our own contributions and made them publicly available at https://github.com/emmau678/opentitan/tree/mphil_thesis_pqc_acceleration.

Future work might investigate the general applicability of our techniques to other cryptographic contexts. Minor modifications to our implementation would enable straightforward translation to the INTT in Kyber and the (I)NTT in Dilithium, with the potential to serve as a foundation for optimisation of other PQC functions on modern cryptographic co-processors.

Acknowledgements

We are grateful to Robert Mullins and particularly Andreas Kurth at lowRISC C.I.C. for helping to define the project, offering access to relevant resources and supporting the implementation efforts of the first author.

References

1. Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang and Shang-Yi Yang. "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1". *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2022**(1):221–244, Nov. 2021. https://doi.org/10.46586/tches.v2022.i1.221-244. URL https://eprint.iacr.org/2021/986.
2. Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. "Keccak". In Thomas Johansson and Phong Q. Nguyen (Editors), "Advances in Cryptology – EUROCRYPT 2013", pages 313–314. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-38348-9. https://doi.org/10.1007/978-3-642-38348-9_19. URL https://eprint.iacr.org/2015/389.
3. Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler and Damien Stehle. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM". In "2018 IEEE European Symposium on Security and Privacy (EuroS&P)", pages 353–367. IEEE, 2018. https://doi.org/10.1109/EuroSP.2018.00032. URL https://eprint.iacr.org/2017/634.pdf.
4. Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler and Damien Stehlé. "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme". *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2018**(1):238–268, Feb. 2018. https://doi.org/10.13154/tches.v2018.i1.238-268. URL https://eprint.iacr.org/2017/633.pdf.
5. Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte and Zhenfei Zhang. "Falcon: Fast-Fourier lattice-based compact signatures over NTRU", 2018. URL https://www.di.ens.fr/~prest/Publications/falcon.pdf. Submission to the NIST's post-quantum cryptography standardization process.
6. Tim Fritzmann, Georg Sigl and Johanna Sepúlveda. "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography". *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2020**(4):239–280, Aug. 2020. https://doi.org/10.13154/tches.v2020.i4.239-280.
7. lowRISC. "OpenTitan", 2024. URL https://opentitan.org/.
8. lowRISC. "OTBN - OpenTitan Documentation", 2024. URL https://opentitan.org/book/hw/ip/otbn/.
9. Victor S. Miller. "Use of Elliptic Curves in Cryptography". In Hugh C. Williams (Editor), "Advances in Cryptology — CRYPTO '85 Proceedings", pages 417–426. Springer, Berlin, Heidelberg, 1986. ISBN 978-3-540-39799-1. URL https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf.
10. Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara and Luca Fanucci. "A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms". *IEEE Access*, **9**:150798–150808, 2021. https://doi.org/10.1109/ACCESS.2021.3126208.

11. Ziying Ni, Ayesha Khalid, Dur e Shahwar Kundi, Máire O'Neill and Weiqiang Liu. "HPKA: A High-Performance CRYSTALS-Kyber Accelerator Exploring Efficient Pipelining". *IEEE Transactions on Computers*, **72**(12):3340–3353, Dec 2023. ISSN 1557-9956. https://doi.org/10.1109/TC.2023.3296899. URL https://eprint.iacr.org/2022/1093.
12. PQ-CRYSTALS. "Kyber". GitHub, Aug 2022. URL https://github.com/pq-crystals/kyber.
13. Ronald L Rivest, Adi Shamir and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". *Communications of the ACM*, **21**(2):120–126, Feb 1978. ISSN 0001-0782. https://doi.org/10.1145/359340.359342.
14. Gregor Seiler. "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography". *IACR Cryptology ePrint Archive*, 2018. URL http://eprint.iacr.org/2018/039.
15. Peter W Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". *SIAM J. Comput.*, **26**(5):1484–1509, Oct 1997. ISSN 0097-5397. https://doi.org/10.1137/S0097539795293172.
16. Ferhat Yaman, Ahmet Can Mert, Erdinç Öztürk and Erkay Savaş. "A Hardware Accelerator for Polynomial Multiplication Operation of CRYSTALS-KYBER PQC Scheme". In "2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)", pages 1020–1025. IEEE, 2021. https://doi.org/10.23919/DATE51398.2021.9474139. URL https://eprint.iacr.org/2021/485.

Algorithm 1: Vectorized NTT Implementation

```
Masks[mask_len8] ← [0x00000000] * 4 + [0xFFFFFFFF] * 4
Masks[mask_len4] ← ([0x00000000] * 2 + [0xFFFFFFFF] * 2) * 2
Masks[mask_len2] ← [0x00000000FFFFFFFF] * 4
Function mont_reduce_vec(a):
    for i ← 0 to 7 do
        t[i] ← a[i] × QINV
        t[i] ← t[i] × KYBER_Q
        t[i] ← a[i] - t[i]
        t[i] ← t[i] ≫ 16
    return t
Function fqmulvec(vec_a, vec_b):
    return mont_reduce_vec(a[i]&0xFFFF × b[i]&0xFFFF for i ← 0 to 7)
Function ntt_vec(r[256]):
    for len in {128, 64, 32, 16} do
        for start ← 0 to 255 by 2×len do
            zetavec ← broadcast zeta[k++]
            for i ← 0 to 15 do
                idx ← i×16 + start
                Vec[rj_vec] ← r[idx ... idx+15]
                Vec[rjlen_vec] ← r[idx+len ... idx+len+15]
                Vec[rjlen_vec_low] ← rjlen_vec AND [0x0000FFFF] * 8
                Vec[rjlen_vec_upp] ← rjlen_vec ≫ 16
                t_low ← fqmulvec(zetavec, rjlen_vec_low)
                t_upp ← fqmulvec(zetavec, rjlen_vec_upp)
                t ← t_low XOR t_upp
                rjlen_vec_new ← rj_vec - t
                rj_vec_new ← rj_vec + t
                store rjlen_vec_new, rj_vec_new to r[idx+len], r[idx]
    for len in {8, 4, 2} do
        for i ← 0 to 15 do
            num_zetas ← 8 / len
            zetavec ← 0
            zeta_mask ← (1 ≪ (len ≪ 4)) - 1
            for z ← 0 to num_zetas - 1 do
                tmp ← broadcast zeta[k++]
                tmp ← tmp AND zeta_mask
                zetavec ← zetavec XOR tmp
                zeta_mask ← zeta_mask ≪ (len ≪ 5)
            idx ← i × 16
            Vec[rjvec] ← r[idx ... idx+15]
            Vec[nextvec] ← r[idx+16 ... idx+31]
            Vec[rjlen_vec] ← (nextvec ⊕ rjvec) ≫ (len ≪ 4)
            Vec[rjlen_vec_low] ← rjlen_vec AND [0x0000FFFF] * 8
            Vec[rjlen_vec_upp] ← rjlen_vec ≫ 16
            t_low ← fqmulvec(zetavec, rjlen_vec_low)
            t_upp ← fqmulvec(zetavec, rjlen_vec_upp)
            t ← t_low XOR t_upp
            Vec[rjlen_vec_new] ← rjvec - t
            rjlen_vec_new ← rjlen_vec_new AND Masks[mask_len{len}]
            rjlen_vec_new ← rjlen_vec_new ≪ (len ≪ 4)
            Vec[rjvec_new] ← rjvec + t
            rjvec_new ← rjvec_new AND Masks[mask_len{len}]
            res ← rjvec_new XOR rjlen_vec_new
```