Studying OpenCL-based Number Theoretic Transform for heterogeneous platforms

Evangelos Haleplidis*†, Thanasis Tsakoulis*‡, Alexander El-Kady*‡,
Charis Dimopoulos*‡, Odysseas Koufopavlou‡§, Apostolos P. Fournaris*
*Industrial Systems Institute, R.C. ATHENA, Patras Science Park, 26504 Platani-Patras, Greece
†Department of Digital Systems, University of Piraeus, Piraeus, Greece
‡Electrical and Computer Engineering Department, University of Patras, Rion Campus
Email:*(haleplidis, tsakoulis, elkady, dimopoulos, fournaris)@isi.gr, §odysseas@ece.upatras.gr

2021 24th Euromicro Conference on Digital System Design (DSD) | 978-1-6654-2703-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/DSD53832.2021.00058

***Abstract*—Lattice based cryptography can be considered a candidate alternative for post-quantum cryptosystems offering key exchange, digital signature and encryption functionality. Number Theoretic Transform (NTT) can be utilized to achieve better performance for these functionalities, where polynomials are needed to be multiplied. NTT simplifies the multiplication overhead allowing point-wise multiplication by transforming the polynomials into the spectral domain and then inversing the result to the original domain. It is important to optimize this technique that is used in a wide range of computing systems. In this paper we study the feasibility of using OpenCL, a portable framework, to implement a parallelized version of NTT which allows deployment on heterogeneous platforms, such as Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). We measure the performance of our implementation on a GPU and evaluate when and where such a deployment is beneficial. Our results showed that the proposed parallel implementation is a viable acceleration approach for these algorithms for lattice-based cryptography solutions.**

***Index Terms*—NTT, Inverse NTT, Cryptography, OpenCL, Parallel Programming**

I. Introduction

For many years, a thoroughly researched point of interest by the scientific community has been the polynomial multiplication, as evident by the sheer variety of potential applications that it can be utilized in. Operations such as those of correlation, convolution, multi-precision and matrix multiplication, Superfast Toplitz system solvers (Px=y), CRC encoding and decoding, Prime Factor Algorithms (PFA), as well as cryptographic schemes employ the polynomial multiplication at their core.

In regards to cryptography, the recent research efforts that have been observed in the development of quantum computers have the ability to cause a paradigm shift in the way secure communications are implemented in today's computing systems. Quantum systems have the ability to completely bypass the security level that current Public Key cryptography protocols based on integer factorization (e.g. RSA) or discrete logarithm (e.g. ECDSA) problems can offer, by mainly utilizing Shor's algorithm [1]. Thus, the need to design fast, efficient and resilient schemes against quantum computers has led to the emergence of various such efforts, primarily led by the NIST Post-Quantum cryptography standardization project [2] and the official recommendations regarding post-quantum security from the NSA [3].

In this new reality, Lattice based cryptography (LBC) can be considered a very strong alternative for future post-quantum cryptosystems, as is already apparent by its presence in the NIST post-quantum cryptography contest [2], offering key exchange, digital signature and encryption functionality. However, the challenging issue of performance requirements in the forms of execution speeds, area or power constraints raises the need of further research into optimizing the most intensive operation in lattice based cryptosystems, i.e. polynomial multiplication. One very interesting technique to realize polynomial multiplication is the Number Theoretic Transform (NTT), where the polynomials to be multiplied are transformed into the spectral domain, simplifying the multiplication overhead by point-wise multiplying the two polynomials. The inverse transform (NTT^{-1}) is then responsible for reconstructing the final result vector into the normal domain, thus reducing the overall multiplication complexity from $O(n^2)$ to $O(n \cdot log(n))$. Contrary to the way Discrete Fourier Transform (DFT) operates, NTT involves integer arithmetic operations, elevating NTT to a more suitable candidate for cryptographic applications that require robust and error-free operations.

Considering the fact that polynomial multiplication is the main bottleneck of lattice-based cryptography and that many finalist candidates in the NIST PQC standardization effort [2] utilize NTT [4] [5], it is of paramount importance to optimize the aforementioned technique for a wide range of computing systems, ensuring fast, secure and flexible implementations that can be adopted by resource constrained platforms in an ever expanding and connected environment.

A. Related Work

As the NTT algorithm is present in a large number of various applications, many different implementations have been proposed before. Aiming to render the NTT operation efficient for real-world usage, there have been works in regards to hardware implementations that incorporate different optimization techniques, targeting either FPGA platforms [6] [7] [8] [9] [10] [11] or even low-power ASIC designs, such as [12], that exhibits a no-cost power dissipation of 30%. There have also been other software implementations like the Nflib

978-1-6654-2703-6/21/$31.00 ©2021 IEEE
DOI 10.1109/DSD53832.2021.00058

library [13], as well as GPU-based optimization efforts [14] [15] [16] [17] [18] [19]. However, these works have as basis the proprietary CUDA parallel computing platform, a fact that can potentially limit the widespread adoption and inclusion of the above optimizations in a plethora of computing systems.

B. Our Contribution

In this paper, we present the results of our OpenCL implementation of the NTT and Inverse NTT algorithms on an NVIDIA GTX 1050 Ti GPU. While the performance was evaluated based on the reference C code implementation of round 2 of Dilithium Digital Signature Scheme [4], this work can be used for other techniques that are employing these algorithms. OpenCL was selected due to the portability and flexibility that it provides to make our implementation usable in various platforms. To the best of our knowledge, this is the first OpenCL implementation and comparison of these two algorithms.

The rest of the paper is organized as follows. In Section II, a brief overview of the Number-Theoretic Transform (NTT) definition is presented, in Section III the proposed OpenCL implementation is described in detail and in Section IV the results of the comparison between the C implementation of the reference code and its equivalent OpenCL solution are presented and discussed.

II. Number Theoretic Transform (NTT)

A. Definitions

Number-theoretic Transform (NTT) is a generalization of the Discrete Fourier Transform (DFT) to finite fields. It allows the execution of computations on integer sequences without any round errors and a significant reduction in computational complexity.

Let f be a polynomial of degree n, where $f = \sum_{i=0}^{n-1} f_i X^i$ and $f_i \in \mathbb{Z}_q$, q is a prime number and ω is the primitive n-th root of unity or otherwise called twiddle factor, which must satisfy the conditions $\omega^n \equiv 1 (\text{mod } q)$ and $\omega^i \neq 1 \pmod q \ \forall i < n$, where $q \equiv 1 \pmod n$. Under these conditions, the forward NTT is then defined as:

$$\hat{f}_i = NTT(f) = \sum_{j=0}^{n-1} f_j \omega_n^{ij} \bmod q \tag{1}$$

Respectively, the inverse NTT (INTT) operation is defined as:

$$f_i = INTT(\hat{f}) = n^{-1} \sum_{j=0}^{n-1} f_j \omega_n^{-ij} \bmod q \tag{2}$$

The above Equations 1 and 2, when applied in their basic form can lead to an increase of the required computational overhead. Thus, in order to avoid multiplying two polynomials f and g over a polynomial ring $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$ in $O(n^2)$, the following NTT property can be utilized:

$$f \cdot g = INTT(NTT(f) * NTT(g))$$

Taking advantage of this property, we can achieve the computation of the same multiplication in $O(n \cdot log(n))$ time complexity. Based on that fact, the NTT and INTT algorithms that we adopted for easier development and testing can be seen on Algorithm 1 and 2 respectively.

Algorithm 1 In-place forward radix-2 Cooley-Tukey NTT

Input a polynomial $a(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$, n-th primitive root of unity $\omega_n \in \mathbb{Z}_q$, $n = 2^l$
Output NTT(a)

$\hat{a} \leftarrow$ bit-reverse(a)
for $(i = 1 : i < l : i++)$ **do**
  $m \leftarrow= 2^{l-i}$
  $\omega_m \leftarrow \omega_n^{n/m}$
  for $(j = 0 : j < n : j++)$ **do**
    $\omega \leftarrow 1$
    for $(k = 0 : k < m/2 : m++)$ **do**
      $t1 \leftarrow \omega * \hat{a}[k + j + m/2] \bmod q$
      $t2 \leftarrow \hat{a}[k + j]$
      $\hat{a}[k + j] \leftarrow t1 + t2 \bmod q$
      $\hat{a}[k + j + m/2] \leftarrow t1 - t2 \bmod q$
      $\omega \leftarrow \omega * \omega_m \bmod q$
    end for
  end for
end for
return $\hat{a}(x)$

Algorithm 2 In-place forward radix-2 Cooley-Tukey INTT

Input $\hat{a}$ polynomial $\hat{a}(x) \in \mathbb{Z}_q[X]/\langle X^n + 1\rangle$, n-th primitive root of unity $\omega_n \in \mathbb{Z}_q$, $n = 2^l$
Output INTT($\hat{a}$)

$a \leftarrow \hat{a}$
for $(i = l : i >= 1 : i--)$ **do**
  $m \leftarrow= 2^{l-i}$
  $\omega_m \leftarrow \omega_n^{n/m}$
  for $(j = 0 : j < n : j++)$ **do**
    $\omega \leftarrow 1/n$
    for $(k = 0 : k < m/2 : m++)$ **do**
      $t1 \leftarrow a[k + j]$
      $t2 \leftarrow a[k + j + m/2]$
      $a[k + j] \leftarrow t1 + t2 \bmod q$
      $a[k + j + m/2] \leftarrow \omega * [t1 - t2] \bmod q$
      $\omega \leftarrow \omega * \omega_m \bmod q$
    end for
  end for
end for
return $a(x)$

B. NTT and optimization platforms

As will be discussed in the next section, both NTT and Inverse NTT algorithms offer avenues for parallelization. Nowadays there are a number of platforms that can support

hardware acceleration for parallel computing, from FPGAs, to CPUs and General Purpose Computing on Graphics Processing Units (GPGPUs) [20].

A GPU can contain a large number of cores, ranging from hundred to thousands, allowing tasks to be executed in parallel. GPUs are used for a multiple different applications, other than graphic computations, such as machine learning and cryptography. GPUs also have on-chip memory, which is small but fast, and off-chip memory, which is large but slow. It is up to the developer to allocate the memory to be used as well as provide the functional computations to be performed on the input. There are two major frameworks for heterogeneous parallel computing, NVIDIA's Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL).

CUDA is a parallel computing platform and programming model, developed by NVIDIA, for performing computations mainly on NVIDIA's GPUs. CUDA allows developers to write C++ code and execute functions in parallel in thousand of lightweight thread, grouped in thread blocks. The function that runs in parallel is also called a kernel.

OpenCL on the other hand, is an open standard maintained by the Khronos Group consortium. Unlike CUDA, OpenCL is a more generic and portable framework, agnostic to the parallel processing architecture at hand and is usable on multiple heterogeneous platforms. The same concepts of CUDA are applied in OpenCL but with different nomeclature. For example a CUDA thread is a Work-item in OpenCL, representing one kernel, and a CUDA thread block is an OpenCL Work-group.

There are implementations such as [17] for NTT and Inverse NTT using CUDA which have achieved an increase in processing speed. While studies such as [21] show that CUDA can perform faster parallel computations for example for Basic Linear Algebra (BLAS) functions, CUDA is limiting the potential exploitation of our proposed solution in hardware that do not have an NVIDIA GPU, for example in FPGAs.

We opted to use OpenCL as the framework upon to implement NTT and Inverse NTT. Studies such as [22] show that the benefits of using OpenCL, whilst a little less optimal than CUDA, far outweigh the performance issues. Having our solution based on OpenCL provides a more realistic approach for deploying on a broader range of platforms, such as FPGAs on embedded devices.

III. Proposed NTT/Inverse NTT OpenCL Solution

This section describes our proposed OpenCL implementation. We will outline the potential avenues for parallelizing both NTT and Inverse NTT and discuss the implementation details. Our approach closely follows the NTT/Inverse NTT implementation in [4] that uses the Cooley-Tukey Fast Fourier Transform (CT-FFT) [23] and implements the algorithms defined in Section II-A.

A. Parallelization optimizations

Both NTT and Inverse NTT algorithms using CT-FFT are implemented using three for-loops taking as input a polynomial in the form of a vector v with n values, each n_i value being the coefficient of the polynomial. The outer for-loop counts the rounds of the algorithm. The number of rounds is equal to $log_2(n)$. In each round the vector v is split into equal size blocks. While the number of rounds is equal for NTT and Inverse NTT, the computations and the length of each block is slightly different as presented in Section II-A.

B. NTT

In NTT, each round the input vector is spilt into $2^{round-1}$ blocks as can be seen in Figure 1. Each block has a length equal to $n/(2^{round-1})$ and is logically split into two halves. The two internal for-loops, as described in Algorithm 1 - lines 5-14, execute the butterfly operations for each round in each block by performing computations using one coefficient from the first half of the block with the corresponding coefficient in the second half of the block, $n[i]$, $n[i + blockLength/2]$. The corresponding coefficients required for calculating the results for the next round for the second half of the block follows a similar pattern, $n[i]$, $n[i - blockLength/2]$.

It should be noted that Figure 1 does not detail the computation performed with the coefficients as defined by the algorithm, rather it shows which coefficients are required to be processed with each other in order to output the required input for the next round of computations.

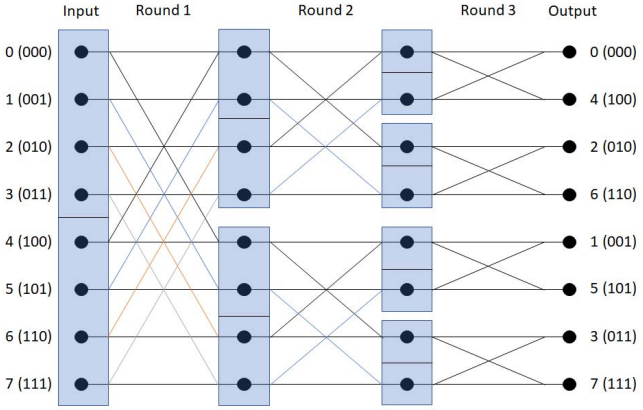

Fig. 1. NTT computations without initial bit reversal

It must also be noted that Algorithm 1 first does a bit reversal on line 1 on input vector and changes the order of the coefficients of the polynomial in order to perform the operations and calculate the output coefficients. While the example showed in Figure 1 does not perform the bit reversal, however the same operations are performed and the only difference is that the output is bit reversed. Our implementation of NTT implements Figure 1.

In the example showed in Figure 1, for an input vector v with $n = 8$, for the first round, there is only one block, with block length equal to 8 and $n[0]$ is computed corresponding with $n[0 + 8/2]$, which translates to $n[4]$. In the second round there are two blocks, with block length equal to 4 and $n[0]$ is computed corresponding with $n[0 + 4/2]$ which translates to $n[2]$.

We can safely parallelize the computations of each round, which correspond to the two internal for-loops, as the computations that are occurring with the corresponding coefficients $n[i]$, $n[i + blockLength/2]$ in each block are independent of the other computations in all the other blocks of the vector v in the same round and are done in parallel and not sequential and don't depend on results from other rounds.

Theoretically, we're taking an $O(n \cdot log(n))$ algorithm and since we run the two internal for loops in parallel, we only have the same actions repeated $log(n)$ times, thus transforming the algorithm to $O(log(n))$. Realistically this applies only while there are still available cores in the GPU and the GPU is not saturated. Once the GPU is saturated, $O(log(n))$ does not apply as will be shown in Section IV.

C. Inverse NTT

In Inverse NTT, the logic is the same, but the splitting of the blocks is backwards. In each round the vector is split into $2^{(log_2(n)-round-1)}$ blocks, which is the reverse of NTT. Each block has a length equal to $n/(2^{(log_2(n)-round-1)})$. Identical to NTT, each block is logically split into two halves. The two internal for-loops execute the butterfly operations for each round in each block by performing computations using one coefficient of the first half of the block with the corresponding coefficient in the second half of the block, $n[i]$, $n[i + blockLength/2]$. E.g. For a vector v with $n = 256$, for the first round, there are 128 blocks and the $n[0]$ coefficient is computed with the $n[1]$ coefficient.

As with NTT, Inverse NTT can also have the two internal for-loops safely parallelized as, again, the computations that are occurring with the corresponding coefficients $n[i]$, $n[i + blockLength/2]$ in each block are independent of the other computations in all the other blocks of the vector v in the same round and are done in parallel.

D. Batching NTT/Inverse NTT

There is also another avenue to increase the efficiency of parallel platforms by batching computations as much as possible. Algorithms, such as discussed in I, may require the transformation of multiple polynomials in sequence. If there are enough cores available in the platform we can batch the input with a number of equal sized polynomials as one input vector.

In order to implement batching, it is crucial to distinguish each index of the different polynomial coefficients so that we can identify the block, as well as which half it belongs to. This is simply implemented by discovering the local index of each coefficient of each polynomial based on the index location of the input vector, the global index, by performing a modulo operation of the global index with the size of the polynomials. Once the local index is computed, it is easy to calculate the local block by performing the same operations as discussed in the previous two subsections.

This same process is applied both for the NTT and the Inverse NTT algorithm. The output of the batching process is one vector containing the results of each polynomials in the same sequence as the input.

IV. Results

We implemented our work on an Intel i7-7700 CPU with 3.6 Ghz frequency hosting an NVIDIA GTX 1050 Ti GPU with 786 cores running Windows 10. The selection criteria of this GPU was the utilization of an average performance commercial device. We compared the processing speed of NTT and Inverse NTT, with various sized polynomials and different batching sizes, in C, on the CPU, and in OpenCL, on the GPU. To perform a more relevant comparison we used the C reference implementation of NTT and Inverse NTT of phase two of Dilithium with an OpenCL equivalent as discussed in Section III.

A. NTT experimental results

We tested various permutations of the number of coefficients of polynomials as well as the number of batched input polynomials. For the number of coefficients we tested polynomials with 64, 128, 256, 512, 1024 and 2048 and for varying batching sizes we used 1, 2, 4, 8, 16, 32 and 64 input polynomials. The total number of coefficients that are being processed by the NTT and Inverse NTT algorithm is the number of coefficients per polynomial multiplied by the batch size. As a result, the input size of the algorithm varies from 64 to 131,072 coefficients with each coefficient being an unsigned integer.

Our measurements are shown in tables I, II, IV and V. The first column of each table contains the batch size of the input polynomials. The remaining columns represent a different set of experiments based on the size of the polynomial, in other words the number of coefficients. The values indicate the amount of time, in nanoseconds, required for the algorithm to execute either in the CPU or in the GPU.

We collected the GPU measurements using OpenCL's profiling events that NVIDIA GPU supports and for the C code, we used the timespec_get function. It must be noted that we are interested only in the execution time of both the C and OpenCL kernel code and not on any other overheads, such as memory transfer which may vary among platforms.

During our tests, we found that by running single experiments the test results varied substantially. We estimate that this difference in our results had to do with the timescale of the running time of the algorithm in respect to the input size, being in the nanosecond scale, as well as to errors in the measurement process. To collect accurate measurement as possible, we performed for each test an arbitrary 100 times iterations of the algorithm and calculated the average running time. Then we run our test an arbitrary 100 times and collected the minimum time reported. This process almost eliminated the variance of the testing results to a number of a couple nanoseconds.

Tables I and II show the testing results for the NTT algorithm for the reference C code implementation and the GPU OpenCL implementation respectively. To better illustrate the comparison between these two implementations we divided these tables into three segments based on the number of coefficients of the input polynomials, 64 and 128, 256 and

512 and finally 1024 and 2048. Each segment depicts two respective columns of table I and II. These comparisons are illustrated in Figures 2, 3 and 4 accordingly.

TABLE I
NTT C CODE METRICS (EXECUTION TIME IN NS)

Batch Size	Number of coefficients					
	64	128	256	512	1024	2048
1	314	693	1818	3636	8576	18900
2	631	1393	3249	7106	17600	37600
4	1245	2825	6451	13892	35000	82270
8	2514	5549	12823	28265	69900	173656
16	5064	11088	26239	64400	139400	371841
32	10036	22660	54826	132100	284000	733024
64	20138	47313	108742	263600	569000	1543784

TABLE II
NTT OPENCL CODE METRICS (EXECUTION TIME IN NS)

Batch Size	Number of coefficients					
	64	128	256	512	1024	2048
1	9171	9049	10331	11133	14126	15043
2	9127	9274	10483	11358	17376	17408
4	8354	9300	10575	11570	18144	20362
8	8387	9391	10696	12128	21504	31506
16	8465	9547	11381	15776	31744	53654
32	8543	10143	14032	26816	63456	97278
64	9274	13075	23753	46016	111456	228783

Figure 2 contains four characteristic curves. The solid-line blue curve corresponds to the 64-element column of the C code implementation and solid-line red curve corresponds to 64-element column of the OpenCL implementation. In a similar way, blue and red dashed-line curves depict the 128-element columns of table I and table II respectively. We opted for a logarithmic scale on the $x - axis$, as we were exponentially increasing the number of batched polynomials. The rest of the Figures follow the same approach, with the dashed-lines corresponding to the greater vector size implementation.

As expected, based on the analysis in Section III, the timing behaviour of the OpenCL implementation, shown by red lines, remains almost constant for a large number of input polynomials. However, it is crucial to notice that when the number of polynomial surpasses 32, for both 64 and 128 vector size, then the OpenCL performance degrades. At this point, the processing capability of the device has approached to its saturation point and no more data can be treated in parallel. However even in that case, the OpenCL implementation continues to be better than the C code.

As it is important to ascertain when it is beneficial to use our approach we have mapped the intersection points between the corresponding implementations in Figure 2 by a magenta-colored circle. Ideally, the x-value of each intersection point multiplied by the corresponding vector size would be equal. Alternatively, the x-value of p_{64} should be two times larger than x-value of p_{128}.

From the experimental results, the intersection points of the 64 and 128 vector size implementations lie around $p_{64} = [25.64, 8500]$ and $p_{128} = [13.08, 9500]$ respectively. This outcome could be also interpreted by comparing the y-values of intersection point which differ only for $1us$.

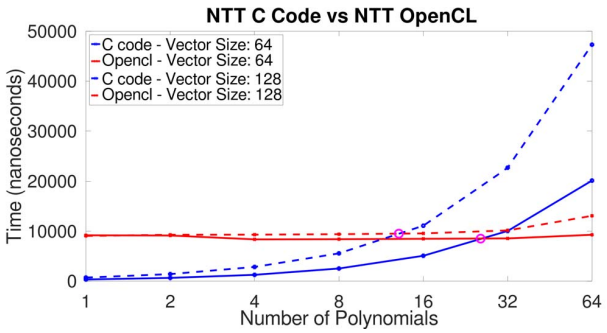

Fig. 2. Vector size: 64 and 128.

Comparing the x-values of the intersection points, it is obvious that there is a minor deviation of 2% from the theoretical values. This above-mentioned difference is reasonable and could be explained due to statistical errors as we don't record values at those points. However, for batch sizes greater than 26 and 13, OpenCL shows better performance than the C equivalent for 64 and 128 coefficient respectively. In other words, the OpenCL implementation outperforms the C in cases where the input data size is larger than 1,640 total coefficients. In the 128 vector size case, the tipping point comes around to 1,674 coefficients.

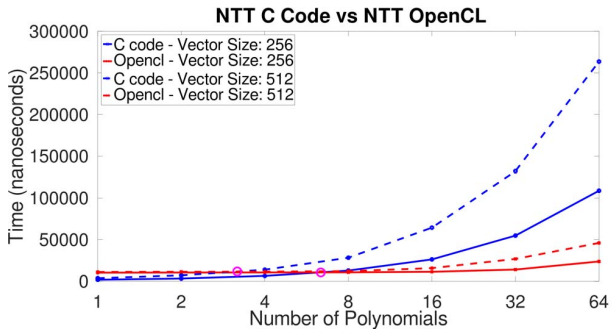

Fig. 3. Vector Size: 256 and 512.

Figure3 illustrates the timing measurements for 256 and 512 polynomial size related to the number of batched input polynomials. The C code (blue curves) as expected still follows the same growth, an almost linear behaviour as the number of polynomials increase. On the other hand, OpenCL implementations (red curves) exhibit minor variations until the input is greater than 16 polynomials. At this point, as with the previous Figure, the OpenCL performance degrades but still outperforms by far the C code.

As depicted, the intersection point in the 256 vector size is located at $p_{256} = 6.64$. In other words, if the number of batched polynomials is greater than 6, the OpenCL implementation, for 256 vector size, outperforms the C program. In case of 512 vector size, the corresponding intersection point

lies at $p_{512} = 3.3$. Similarly, a migration to OpenCL would be beneficial for implementations that require greater than 3 batched polynomials as input data size.

Lastly for the NTT experimental results, the 1024 and 2048 vector size implementations are shown in Figure 4. At this size of batched input, OpenCL outperforms the C implementation from the start. In the first case, any input data size of polynomials larger than 2 should be implemented in OpenCL. In case of 2048 vector size, the OpenCL approach should be preferred.

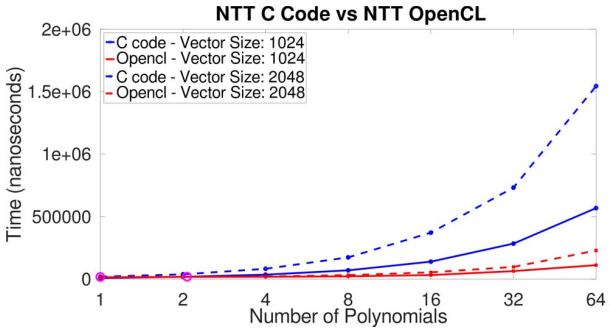

Fig. 4. Vector Size: 1024 and 2048.

Table III better illustrates the timing performance gain of OpenCL versus the C code. As has already been made clear, the OpenCL kernel outperforms the C implementation, values in the table larger than one, only after an adequate number of input values. Correlating the number of coefficients with the number of polynomials, it is clear that the OpenCL implementation resulting over 2048 total elements outperforms the C code.

TABLE III
NTT C/OPENCL SPEEDUP

Batch Size	Number of coefficients					
	64	128	256	512	1024	2048
1	x0.03	x0.08	x0.18	x0.33	x0.61	x1.26
2	x0.07	x0.15	x0.31	x0.63	x1.01	x2.16
4	x0.15	x0.3	x0.61	x1.2	x1.93	x4.04
8	x0.3	x0.59	x1.2	x2.33	x3.25	x5.51
16	x0.6	x1.16	x2.31	x4.08	x4.39	x6.93
32	x1.17	x2.23	x3.91	x4.93	x4.48	x7.54
64	x2.17	x3.62	x4.58	x5.73	x5.11	x6.75

B. Inverse NTT experimental results

A similar approach has been followed for the Inverse NTT experimental results. At first, C code and OpenCL metrics are shown in tables IV and V respectively. Again batch sizes range from 1 to 64 and the number of coefficients per polynomial varies from 64 to 2048. All values in the tables are expressed in nanoseconds.

As with NTT, the experimental results of tables IV and V are being presented in pairs. Larger vector size implementations are shown by dashed-lines in contrast with the smaller vector size implementations which are illustrated by solid-lines. As

TABLE IV
INVERSE NTT C CODE METRICS (EXECUTION TIME IN NS)

Batch Size	Number of coefficients					
	64	128	256	512	1024	2048
1	338	795	1731	3809	8400	18000
2	668	1617	3736	7922	17000	36100
4	1341	3325	7335	16000	33700	76345
8	2737	6571	14974	31700	67300	163600
16	5565	13202	30119	63300	129700	331268
32	11308	25911	65064	126000	272900	648613
64	21642	56983	133653	255900	542000	1399069

TABLE V
INVERSE NTT OPENCL CODE METRICS (EXECUTION TIME IN NS)

Batch Size	Number of coefficients					
	64	128	256	512	1024	2048
1	8720	8474	9232	9994	17216	17968
2	8696	8690	9329	10175	17888	20032
4	7954	8886	9416	13984	18240	18651
8	8149	8937	9728	13888	19552	28202
16	8032	9121	10369	15328	27648	47912
32	8379	9606	13036	24576	55968	86580
64	8939	12101	21787	40960	98624	180832

before, blue and red curves represent the conventional and OpenCL implementations respectively.

Figure 5, depicts the first two columns of table IV and V for 64 and 128 polynomial size. As NTT and Inverse NTT are the same order in complexity we observed a similar pattern as expected. In the case of 64 polynomial size, the C code initially has a better performance, but when the batch size becomes greater than 23, OpenCL yields better results.

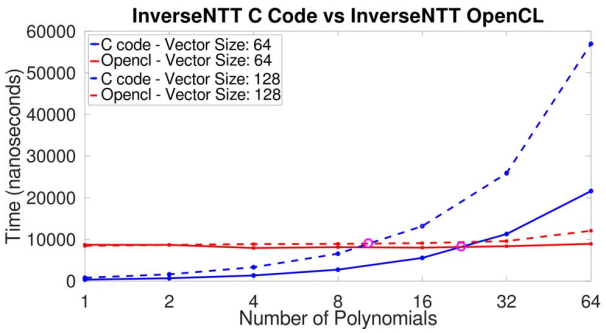

Fig. 5. Vector Size: 64 and 128.

For both vector sizes, there are small differences as the the number of input polynomials increase. For batch size greater than 32, the OpenCL performance becomes linear to the polynomial size. In case of 128 vector size, the intersection point where OpenCL is better is located just over 11 polynomials as expected.

Subsequently, the 256 and 512 polynomial size implementations are depicted in Figure 6. Again the same pattern ensues. On the other hand, OpenCL show small variations until the polynomials become greater than 32. After that point, OpenCL complexity becomes also linear. In case of 256-

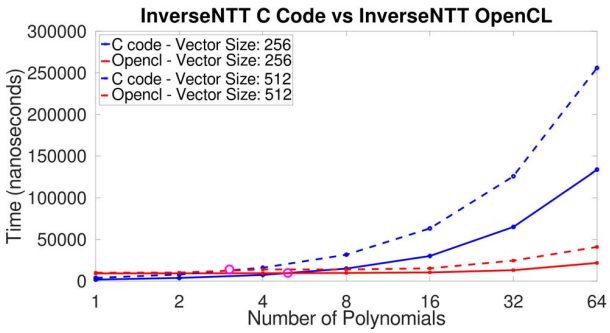

Fig. 6. Vector Size: 256 and 512.

coefficients, OpenCL outperforms the C code for any number of polynomials larger than 7. The corresponding limit in case of 512-implementation steps down to 3 as expected.

Finally, Figure 7 shows the 1024 and 2048 polynomial sizes respectively. The key point in this Figure is that OpenCL time complexity change from almost constant to linear after 8 polynomials. In case of 2048 polynomial sizes, the experimental results show that OpenCL is better for any number of batched input polynomials. As expected for 1024 coefficients, the polynomial input number must be greater than two.

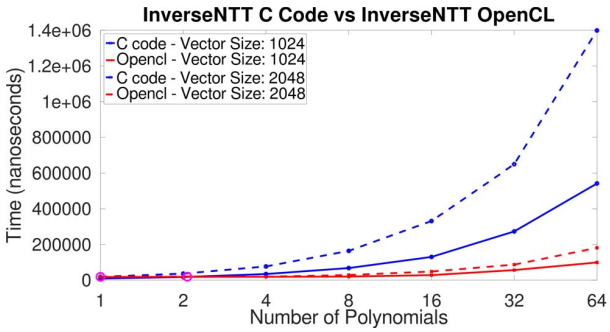

Fig. 7. Vector Size: 1024 and 2048.

We created a similar table for the speed up gain for the Inverse NTT, as shown in table VI. We observed that there is not much of a difference between this table and table VI. Since both algorithms share the same complexity and have almost the same number of operations, we expected that they would have the same results. As such the OpenCL kernel for the Inverse NTT outperforms the C implementation when the overall number of coefficients exceed 2048.

V. Conclusions and Future Work

In this paper we presented an experimental study of a parallelized implementation of NTT and Inverse NTT using OpenCL. We tested our implementation on system with an Intel i7-7700 CPU with 3.6 GHz frequency hosting an NVIDIA GTX 1050 Ti GPU with 786 cores running Windows 10. We compared the execution time of the NTT and Inverse NTT reference C code of phase two of Dilithium and the kernel execution time of the respective OpenCL code on the GPU.

TABLE VI
Inverse NTT C/OpenCL speedup

Batch Size	Number of coefficients					
	64	128	256	512	1024	2048
1	x0.04	x0.09	x0.19	x0.38	x0.49	x1
2	x0.08	x0.19	x0.4	x0.78	x0.95	x1.8
4	x0.17	x0.37	x0.78	x1.14	x1.85	x4.09
8	x0.34	x0.74	x1.54	x2.28	x3.44	x5.8
16	x0.69	x1.45	x2.9	x4.13	x4.69	x6.91
32	x1.35	x2.7	x4.99	x5.13	x4.88	x7.49
64	x2.42	x4.71	x6.13	x6.25	x5.5	x7.74

Our results showed that there is a potential speed up against the CPU, with a maximum of speed up around 7.5x times depending on the number of input coefficients. We found that for an average number of 1664 coefficients the GPU kernel execution time is on par with the CPU and after that there is significant gain on using the GPU. As many algorithms for the post-quantum cryptography of the NIST competition have high degree polynomials and also a large number of polynomials our approach can be applied. Our results are lower than in [18], where there is a speed up of 17x, which was expected, as our implementation is based on OpenCL whereas [18] is based on CUDA.

As future work, we plan to take advantage of the portability the OpenCL platform offers in order to synthesize and implement our NTT and Inverse NTT solution on systems that do not have an NVIDIA GPU, such as FPGA platforms and evaluate the real-time performance of our parallelized OpenCL kernel.

Acknowledgment

This paper's work has received funding from the European Union's Horizon 2020 research and innovation programme CPSoSaware under grant agreement No 871738 and also from the European Union's Horizon 2020 research and innovation programme CONCORDIA under grant agreement No 830927.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, p. 1484–1509, Oct. 1997. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[2] "Nist post-quantum cryptography project." https://csrc.nist.gov/projects/post-quantum-cryptography.

[3] "Nsa/iad. cnsa suite and quantum computing faq," https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm, 2016.

[4] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium: A lattice-based digital signature scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–268, 2018.

[5] "Falcon: Fast-fourier lattice-based compact signatures over ntru," https://falcon-sign.info/falcon.pdf.

[6] C. P. Rentería-Mejía and J. Velasco-Medina, "Hardware design of an ntt-based polynomial multiplier," in *2014 IX Southern Conference on Programmable Logic (SPL)*, 2014, pp. 1–5.

[7] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2020.

[8] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[9] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, L. Batina and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391.

[10] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology – LATINCRYPT 2012*, A. Hevia and G. Neven, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 139–158.

[11] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 81–86.

[12] T. Fritzmann and J. Sepúlveda, "Efficient and flexible low-power ntt for lattice-based cryptography," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019, pp. 141–150.

[13] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "Nfllib: Ntt-based fast lattice library," 02 2016, pp. 341–356.

[14] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "Pqc acceleration using gpus: Frodokem, newhope, and kyber," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, 2021.

[15] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, pp. 70–95, May 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/875

[16] A. A. Badawi, B. Veeravalli, and K. M. Mi Aung, "Faster number theoretic transform on graphics processors for ring learning with errors based cryptography," in *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, 2018, pp. 26–31.

[17] W.-K. Lee, S. Akleylek, D. C.-K. Wong, W.-S. Yap, B.-M. Goi, and S.-O. Hwang, "Parallel implementation of nussbaumer algorithm and number theoretic transform on a gpu platform: application to qtesla," *The Journal of Supercomputing*, vol. 77, no. 4, pp. 3289–3314, 2021.

[18] W.-K. Lee, S. Akleylek, W.-S. Yap, and B.-M. Goi, "Accelerating number theoretic transform in gpu platform for qtesla scheme," in *Information Security Practice and Experience*, S.-H. Heng and J. Lopez, Eds. Cham: Springer International Publishing, 2019, pp. 41–55.

[19] S. Akleylek, Ö. Dağdelen, and Z. Yüce Tok, "On the efficiency of polynomial multiplication for lattice-based cryptography on gpus using cuda," in *Cryptography and Information Security in the Balkans*, E. Pasalic and L. R. Knudsen, Eds. Cham: Springer International Publishing, 2016, pp. 155–168.

[20] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for gpgpu reliability," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 94–104.

[21] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 2015, pp. 195–202.

[22] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 216–225.

[23] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.