NTTSuite: Number Theoretic Transform Benchmarks for Accelerating Encrypted Computation

Juran Ding Yuanzhe Liu Lingbin Sun Brandon Reagen

***Abstract*—Homomorphic encryption (HE) is a cryptographic system that enables computation to occur directly on encrypted data. In this paper we develop a benchmark suite, named NTTSuite, to enable researchers to better address these overheads by studying the primary source of HE's slowdown: the number theoretic transform (NTT). NTTSuite constitutes seven unique NTT algorithms with support for CPUs (C++), GPUs (CUDA), and custom hardware (Catapult HLS). In addition, we propose optimizations to improve the performance of NTT running on FPGAs. We find our implementation outperforms the state-of-the-art by 30%.**

I. Introduction

Homomorphic encryption secures data via lattice-based cryptography. In modern HE schemes, data are encoded into polynomials with noise. When performing computations in HE, these polynomials must frequently change representation to improve performance. This is done via the number theoretic transform (NTT). The NTT is a variation of the more familiar FFT, and it can be used to reduce polynomial multiplication runtime from $O(n^2)$ to $O(nlogn)$. Where multiplication can be significantly sped up in the NTT (or evaluation) domain, some functions can only be processed in the native, or coefficient, representation. In HE, NTT is typically the most expensive function, due to the frequency of transforming polynomial ciphertext and the complexity of NTT. For example, a recent paper profiled a machines learning algorithm running in HE and found that 55% of the total runtime could be attributed to NTT [1].

In this paper we present and develop NTTSuite[1]: a collection of reference implementations for standard NTT algorithms to enable performance and efficiency optimizations on accelerated platforms, including GPUs and custom hardware. NTTSuite constitutes seven core NTT algorithms that highlight the differences in how NTTs are typically implemented, including DIT, DIF, Flat-NTT, Pease, Pease_nc, Six-step, and Stockham. While building the benchmarks, we realized an opportunity for a more efficient implementation of the Pease algorithm that elides data copies between stages. We call this implementation Pease No Copy (Pease_nc) and include it in NTTSuite for a total of seven benchmarks. In addition to implementations, we have a testing environment derived from the CPU implementation. This way, any user can run the CPU (C++) version to get parameter settings and input/outputs to validate accelerator implementations, both for GPU and FPGA. The core of the benchmark is the accelerator implementations, which we believe are most important, as overcoming the large slowdowns of HE requires custom hardware. NTTSuite provides verified implementations of all seven benchmarks using Catapult HLS. With NTTSuite, users can download the code and immediately begin optimizing NTT accelerators with HLS pragmas and code rewriting while comparing hardware results to published results and verifying designs against our test harness.

To demonstrate the utility of NTTSuite we profile the benchmark on all three backends using a range of problem sizes, from 1024 to 16384 points. The experiments highlight the versatility of the benchmark and provide a set of baseline numbers that researchers can use to improve on. We show that HLS optimizations can be made to perform well using a combination of unrolling, partitioning, and pipelining pragmas matched with the careful selection of SRAM type. Finally, we compare our novel NTT algorithm with HLS optimizations against a recent competitive design, HEAX [2], and demonstrate a 30% performance improvement while using fewer resources.

This paper makes the following contributions:

1) We develop (and will release) NTTSuite: a collection of seven NTT algorithms and test harness for CPU, GPU, and custom hardware support.
2) We develop a novel NTT algorithm designed to perform well in custom hardware, named Pease_nc.
3) We optimize, profile, and implement the NTTSuite benchmarks and find our novel NTT algorithm with pragmas and modular reduction optimizations outperforms the current state-of-the-art.

II. The NTTSuite Benchmarks

In addition to the textbook DIT and DIF algorithm, there exist other variations of the NTT algorithm tailored to different computing platforms and microarchitectures. NTTSuite supports seven unique NTT algorithms to enable researchers to compare both the algorithmic tradeoffs on different hardwares and select the best fit for their platform. The seven were chosen to span the tradeoffs in the NTT algorithm design space. In our our evaluation of NTTSuite, we found that an optimized Pease algorithm is the best implementation to fully utilize unrolling and pipelining in FPGA design due to its unique memory access pattern.

NTTSuite: We implement each NTT algorithm in NTTSuite on different computation platforms including C++ for CPUs, CUDA for GPUs, and Catapult HLS for FPGAs. Our new algorithm, Pease_nc aims to remove memory copy in Pease algorithm while allowing us to fully pipeline and parallelize the computation using the same degree of memory partition.

[1]This work was done while Juran, Yuanzhe, and Lingbin were Master degree students at NYU. The paper documents experiments and code that being released for those interested.

Twiddle Factors: NTT can be written as the form in which $x[n]$ is a counterpoint to time domain in Fourier transform and $X[n]$ is a counterpoint to frequency domain in Fourier transform:

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn} \quad (1)$$

where $W_N^i = W_N^{i+N} = g^{\frac{P-1}{N} i}$ is called the twiddle factors. All twiddle factors can be pre-computed and pre-loaded into memory instead of computing them on-the-fly to improve performance. This optimization can be apply to all seven algorithms.

Decimation in Time (DIT): NTT is a divide-and-conquer algorithm that can break down a N-point DFT transform into two smaller $\frac{N}{2}$-points transforms recursively, reducing the time complexity from $O(N^2)$ to $N \log N$. NTTSuite implements the DIT using an iterative form instead of recursive form, as this is more suitable for HLS.

Decimation in Frequency (DIF): Unlike DIT, which does decimation in time domain, DIF decimates at frequency domain. From the symmetrical expression of DIT and DIF, the major difference is butterfly operations between each stages and the order of bit-reverse.

Pease: The work in Pease [3] introduces a new factorization method, which can be represented by Algorithm 1:

$$F_{2^t} = \{\Pi_{k-1}^t (I_{2^{c-1}} \otimes F_2 \otimes T_c)\} R_2^{2^t} \quad (2)$$

Algorithm 1 Pease Alogrithm

Require: $x[n]$ is input array, $W[n]$ is twiddle factor, $n = 2^L$, P is a prime
Ensure: $x \leftarrow$ DFT of x
  bit_reverse(x)
  for $s \leftarrow L$ to 1 **do**
    $base \leftarrow\sim (0xFFFFFFFF << (c-1)))$
    for $r \leftarrow 0$ to $\frac{N}{2} - 1$ **do**
      $f_1 \leftarrow x[r << 1]$
      $f_2 \leftarrow x[(r << 1) + 1]$
      $y[r] \leftarrow (f_1 + f_2)\%P$
      $y[r + \frac{N}{2}] \leftarrow (f_1 - f_2)\%P$
    end for
    Swap(x,y)
  end for

Stockham: Both DIT and DIF algorithms need Bit-Reverse operation, resulting in additional memory accesses. Stockham modifies the natural order NTT algorithm and uses two arrays to avoid bit-reverse operation.

Flat-NTT: HLS perform deficiently in loop unrolling and pipelining if the iteration count is not fixed. Therefore, in the DIT and DIF algorithm, there is a variation that flattens two inner loops to a single loop [4].

Six-step: The Six-step algorithm splits a large NTT into several smaller ones. Although time complexity remain same, more computational overhead is introduced. When number of data points is large (e.g., 16384 data points), smaller blocks are made to fit into a cache, which is beneficial to CPU performance.

Pease_nc: We also optimize the Pease algorithm to save time doing overhead copy work named Pease_nc. The Pease_nc swaps input and output array after each computation loop instead of copy. To enable more parallelism, it uses two types of butterfly operations in different stages. This way, the algoirthm can both save time doing extra work copy work and support unrolling scalability to utilize more hardware effectively.

III. Optimizations

Catapult HLS tool offers two major optimization method: Pipelining and Loop unrolling. In addition, the HLS pragma inline is also an optimization method. Ozcan and Aysu introduce and optimize with the above method. [5] In our paper, we focus on prioritizing improvement in speeding up by analyzing the memory access pattern to breakdown data dependency, paralyzing the algorithm using above optimization method, and utilizing the on-chip block ram (BRAM). We categorize them into three major optimization to establish baseline designs competitive with the state-of-the-art. i) *Pipelining.* This strategy divides a loop iteration into multiple stage so that the next iteration of loop can start before previous iteration completes once all data needed for earlier stage is ready, therefore improving throughput and therefore improving performance. Decreasing the time between executing loop iterations is an effective way to increase performance.to fully utilize the hardware resource and increase throughput and performance of the hardware. However, this is challenging in NTT due to data dependency and memory resource contention. Base on analyze of memory access pattern, we optimize accesses to reduce the iteration intervals (II). In the Access Pattern Analysis part, we comprehensively demonstrates the effect of each algorithm's possibilities to do pipeline. ii) *Parallelism.* The basic structure of each NTTSuite benchmark can be described as nested loop: the outer loop iterates over stages and the inner loop computes the $\frac{N}{2}$ butterfly operations. Since each butterfly operation is independent, we can parallel the computations using more hardware. The challenge is again the memory access pattern, as some algorithms require memory index remapping after each stage. We show that the Pease algorithm is highly amenable to paralleled with using our memory access pattern analysis. iii) *Operation optimization.* We identify the modular reduction operation is the bottleneck of the NTT. To optimize this, NTTSuite includes a recent optimization to perform reduction on FPGAs [2].

A. Access Pattern Analysis for Pipelining and Parallelism

The major computation and memory access patterns can be categorized into three types: DIT, DIF, and Pease. All other variations are designed to satisfy different situations like solving the extra bit reverse, loop flattening, or using swap to replace traditional copy work. Regardless, their computation paradigms are all in three mentioned types.

Pipelining. In NTTSuite, Algorithms DIT, DIF and Peace are implemented in the same way in which the outer loop iterates stages and inner loop comprises $\frac{N}{2}$ non-overlapping butterfly

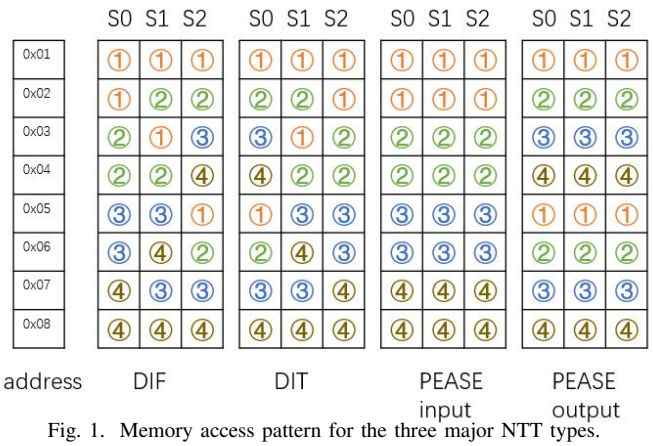

Fig. 1. Memory access pattern for the three major NTT types.

operations. Operations in stage $n+1$ cannot start until all operations in stage n complete and all inner loop butterflies have no data dependencies between each other. Thus, it is possible to pipeline and unroll onto inner loop. In the access pattern graph, see Figure 1, ① denotes the two inputs needed to compute the i-th butterfly operation in the inner loop and Sj indicates the stage j. To fully pipeline DIT, ① and ② should be in different *blocks*, i.e., memory partitions. The challenge with DIT is that the butterfly input pattern changes for every stage, inevitably resulting in block/partition conflicts, i.e., structural hazards, that limit the opportunity for pipelining. E.g., if we partition memory for perfect pipelining in S0, Figure 1 shown that stages S1 and S2 will incur conflicts due to the pattern the data was written back. Since the DIF's memory access is symmetrical to DIT's, which also has the strided pattern, we found the HLS tool could not produce well-pipelined designs with these algorithms.

As seen in Figure 1, the Pease input follows a simple memory access pattern that we can easily separate ①②, ②③,③④ by setting 0x01, 0x02, 0x05, 0x06 to a block and other address to the other block in all stages. (The same works for the output array.) Furthermore, because our Pease implementation is out of place, the resource contention is also much lower than DIT and DIF. Leveraging the FPGA's of Dual port memories, Pease can have $\frac{N}{2}$ read ports resource and $\frac{N}{2}$ write ports resource in $\frac{N}{2}$. This enables us to achieve a pipeline initiation interval of 1 (II=1), even without memory partitioning.

Parallelism. We further improve performance using parallel hardware to execute multiple butterflies from the same stage simultaneously. In DIF algorithm, if parallelizing ①②③④ in stage 1, then {0x01, 0x02} and {0x03, 0x04} and {0x05, 0x06} and {0x07, 0x08} must be separated. But in stage 2 we have: {0x01, 0x03}, {0x02, 0x04}, {0x05, 0x07}, {0x06, 0x08}, which means all eight addresses must be separated. (The same holds for DIT algorithm.) For the Pease algorithm, we can simply divide the input to four parts by setting blocksize=2 and divide the output into four parts by setting interleave=4 in HLS. (Blocksize=B divides a memory into multiple B-word memory partitions. Interleave=M places adjacent memory locations into a different memory partitions. E.g., interleave=4 would partition memory 0,4 into the first memory block, 1,5 into second, 2,6 into third and 3,7 into fourth.) Computing four parallel butterflies on the 8-point problem, there is no difference among these algorithms because DIT, DIF, and Pease all use 8 BRAMS. However, with same analysis on arbitrary 2^t points, DIF and DIT must have 2^t BRAMS to do 4-butterflies in parallel, while Pease still needs only 4 blocks on its input and output array. Thus, we find Pease is also the most amenable to parallelism and can scale up or down depending on constraints. It is also possible to do memory remapping after each stage to support parallelism and pipelining for DIT and DIF algorithms, but it also introduces resource overhead and design complexity. In general, Pease has the best quality to do pipeline and parallel execution.

B. No Copy Optimization

Intuitively, we cannot swap input and output arrays. Since in the following stage, the current input array would now be output array, which needs to be written in a different memory access pattern not same as before to be read. We find that the memory swap costs can be eliminated for the Pease algorithm. As said before, when using dual port memory, both input and output array can provide sufficient resources to realize pipelining. So the ability to pipeline effectively is not changed after a swap. To utilize parallelism, we can set interleave=4 for both input and output, unlike what we did in Pease. Then for arbitrary 2^t data points, in each iteration, input accesses to memory addresses $\{8r{+}0, 8r{+}1, 8r{+}2, 8r{+}3, 8r{+}4, 8r{+}5, 8r{+}6, 8r{+}7,\}$ while output accesses to memory addresses $\{4r+0,\ 4r+1,\ 4r+2,\ 4r+3, 4r+2^{t-1},\ 4r+2^{t-1},\ 4r+2^{t-1},\ 4r+2^{t-1}\}$. Also, from the dual port memory all memory addresses can be accessed simultaneously. That means we can simply set Interleave=N, where N is the number of cores used to parallel. Thus, the input and output arrays can be set up with the exact same memory partition making it safe and effective to swap them while still maintaining pipeline and parallelism optimizations.

To reproduce our optimized Pease_nc (16 butterfly units) result, follow the steps below. First, set frequency to 196MHz, which can still satisfy the slack requirements of slack. Next, configure the memory resource type to dual-port RAM, which best supports pipelining and parallelism. After that, set memory partitions of both input and output arrays to interleave=16, allowing at most 16 butterfly cores working together. Finally, set pipeline interleave to one (II=1), which can significantly increase throughput.

In the end, we can run our Pease_nc on 4096 inputs with a latency of 8.6us shown in the result. HEAX, a state-of-art FPGA implementation for HE, reports a 4096 point NTT latency of 11us, using more hardware resources [2]. Thus, the Pease_nc optimization provides a 30% speedup.

IV. Performance and Analysis

In this section, we show how our optimization methods improve resource usage and performance. In addition, we show how different algorithms perform on CPU (Intel E5), GPU (RTX 8000), and FPGA (Xilinx v7690t1761-2). For a thorough evaluation, we evaluate all NTTSuite algorithms on three input sizes: 1024, 4096, and 16834.

TABLE I
NTTSUITE FPGA RESULTS USING VARIOUS VECTOR SIZES. L, F, D, AND B STAND FOR LUTS, FFS, DSPS, AND BRAMS, RESPECTIVELY.

Name	Size	Time(us)	Freq	L	F	D	B
DIF	1K	851.57	100	973	536	11	0
	4K	4009.41	100	1241	694	12	0
	16K	18601.63	100	1388	680	20	0
DIT	1K	1057.97	100	2248	998	5	0
	4K	5048.13	100	1076	671	11	0
	16K	23198.11	100	1916	776	20	0
Flat-NTT	1K	98.73	167	1142	446	11	0
	4K	466.88	167	1254	470	11	0
	16K	2159.92	167	1306	494	11	0
Pease	1K	5.27	200	20445	22313	32	320
	4K	21.83	200	20350	22349	32	320
	16K	97.19	200	19203	22364	32	320
Pease_nc (4cores)	1K	7.18	196	6005	8592	4	40
	4K	31.93	196	6079	6198	4	40
	16K	146.96	196	6073	5334	4	40
	64K	669.30	196	6145	5409	64	40
Pease_nc (16cores)	1K	2.27	196	23474	32866	16	160
	4K	8.60	196	23696	32976	16	160
	16K	37.50	196	23737	27902	16	160
Six-step	1K	13.00	150	28816	8930	82	32
	4K	38.76	150	47038	19453	163	64
	16K	144.80	150	77558	26359	163	128
Stockham	1K	114.31	135	1602	1329	10	8
	4K	546.92	135	1659	1351	10	8
	16K	2550.22	135	1741	1373	10	16
Heax [2]	4K	11	275	N/A	N/A	1185	1731

A. Methodology

To fairly compare the acceleration performance between GPU and FPGA, we calculate the time after we copy the input vectors to the device and before we copy the vectors from the device back to the host memory. For the GPU, to get a more precise and stable result, we profile the benchmarks 100 times each and report the mean as the measured GPU computation time.

For FPGA experiments, we use Catapult HLS version 10.5c to generate RTL and reports including the throughput (cycles and time), latency (cycles and time), total area, and slack, which we used as our final time result. Waveform are generated using Modelsim for RTL simulation and verification. The generated RTL is then imported to Vivado Design Suite version 2019.1 and synthesized on the FPGA chip. Vivado Design Suite reports the hardware resource used by each algorithm, including LUTs, FFs, BRAMs, and DSPs. Since the input and output vector arrays map to memory ports, those BRAMs are not included. We report each algorithm's best-optimized results to make comparisons using the best performing designs.

B. Observation and Analysis

Figure 2 compares NTTSuite algorithms running on three vector size on Flat, Pease, Pease No-copy, Six-step, and Stockham algorithms (which are improvement of traditional DIT and DIF). Each bar shows FPGA (blue) or GPU (green) speedup normalized to the CPU runtime. We find that that the Stockham Algorithm performs better on GPU compared to FPGA. Pease, Pease_nc, and Six-step perform better on FPGA. Although DIF and DIT perform better on GPU, results on FPGA are much slower than running on CPU. According to the result in Table I, in each algorithm, as the vector size increases, the latency time increases proportionally to the vector size's increment.

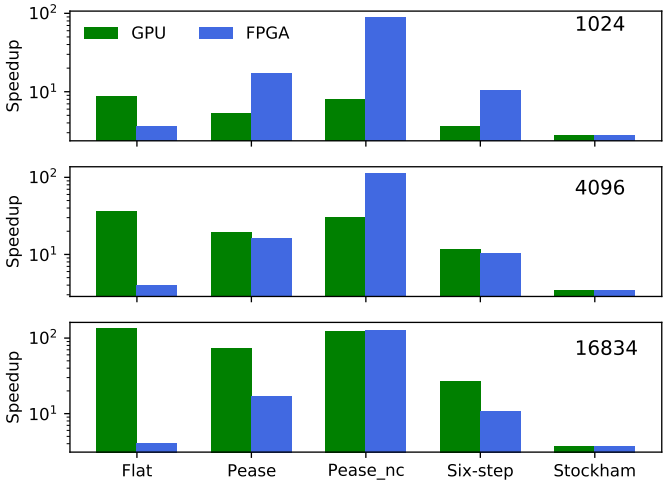

Fig. 2. GPU and FPGA speedup relative to the CPU. Problem size is noted in the top right of each plot.

The Pease_nc algorithm is by far the fastest algorithm on the FPGA. To achieve this performance, it also consumes the most hardware resource compared to the hardware resource consumed by other algorithms on FPGA.

Since the total on-chip power of FPGA is only 1.752W compared to the GPU, which is several hundred watts, our approaches are also energy-efficient for mass-production in data centers and even on embedded systems, given the result of the power analysis by Vivado shows we only used a few watts.

Both DIT and DIF receive few or even negative speedup on FPGA and GPU due to the memory access pattern in which more data dependency exists resulting in low arithmetic intensity. The nested loop structure of DIT and DIF leads to difficulties for HLS to unroll and pipeline loops since iterations are not fixed and complex dependency cann't be resolved. Since the FPGA usually has lower clock frequency, poor performance is expected. A similar issue arises with our GPU implementation since the nested loop will lead to extra function calls on GPU. Flatting them easily resolve the issues, just like the Flat-NTT algorithm did. In this case, a powerful multi-core CPU is more cost-efficient since instruction-level parallelism resolve the data dependency issue better than the utilizing SIMD in GPU and the scratch-pad block memory in FPGA. THere are other work [2] implement these two algorithm directly using Verilog and deployed to FPGA but don't achieve much speedup as other algorithm we implement using HLS in this paper.

Flat, Pease, Pease No-copy, Six-step, and stockham algorithms receive huge boost from FPGA while DIT and DIF runs slower than CPU. The FPGA platform has unique hardware resource including the LUT, LUTRAM, BRAM, FF, and DSP while GPU and CPU does not have. LUTs and DSPs faciliate parallel computation while large BRAM and LUTRAM can be used as scratch pad memory to reduce data access time and number of data movement comparing to multi-level caches in CPUs and GPUs. In addition, during the Synthesis process, computation logic is further optimized for speed which also contributes to the overall speedup.

V. Related Work

Previous works [6]–[8] focus on optimizing the NTT algorithm itself as well as modular reduction [9], [10], butterfly on traditional CPUs. Other focus on Lattice-based computations directly [11]. While HE is not the focus of NTTSuite, previous work in HE also propose several optimization of NTT algorithms since Gentry [12] present the first boostrabable Somewhat Homomorphic Encryption Scheme. In our work, we focus on six algorithms: Decimation-in-Time (DIT [13]), Decimation-in-Frequency (DIF [14]), Flat-NTT [4], Pease Algorithm [3], Stockham [15]), and Six-step [16] that could be applied in any research using NTT. Other works [17], [18] propose GPU accelerated solutions. Kim, Jung, Park, and Ahn [19] compares different algorithms on GPU and analyze their performance and limitations at the same time. Several works [20]–[22] present FPGA solution that focus on optimizing algorithms and implementation using verilog. Recent work [21] also present optimized solution using Vivado HLS and synthesis on FPGA. Reagen, et.all [1] first present algorithm solution with implementation on both FPGA and CUDA.

In summary, while there have been many prior works on accelerating NTT, to the best of our knowledge there are no common set of implementations that evaluates on all three platform: CPU, GPU, and FPGA. NTTSuite aims to fill this gap be providing open source implementations and optimizations that perform on par with the state-of-the-art.

VI. Acknowledgements

This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] B. Reagen, W. Choi, Y. Ko, V. Lee, G.-Y. Wei, H.-H. S. Lee, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," 2020.

[2] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," 2020.

[3] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *J. ACM*, vol. 15, pp. 252–264, 1968.

[4] K. Kawamura, M. Yanagisawa, and N. Togawa, "A loop structure optimization targeting high-level synthesis of fast number theoretic transform," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, 2018, pp. 106–111.

[5] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, vol. 12, no. 4, p. 133–136, 2020.

[6] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *Cryptology and Network Security*, S. Foresti and G. Persiano, Eds. Cham: Springer International Publishing, 2016, pp. 124–139.

[7] G. Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography," 2018, report 2018/039.

[8] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel HEXL: accelerating homomorphic encryption with intel AVX512-IFMA52," *CoRR*, vol. abs/2103.16400, 2021. [Online]. Available: https://arxiv.org/abs/2103.16400

[9] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[10] T. Yanik, E. Savas, and C. Koc, "Incomplete reduction in modular arithmetic," in *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 2, 2002, pp. 46–52.

[11] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," Cryptology ePrint Archive, Report 2016/504, 2016, https://ia.cr/2016/504.

[12] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 169–178. [Online]. Available: https://doi.org/10.1145/1536414.1536440

[13] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, vol. 19, pp. 297–301, 1965.

[14] H. Murakami, "Real-valued decimation-in-time and decimation-in-frequency algorithms," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 41, no. 12, pp. 808–816, 1994.

[15] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, D. Nelson, C. Rader, and P. Welch, "What is the fast fourier transform?" *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1664–1674, 1967.

[16] D. Takahashi, *High-Performance FFT Algorithms*, 10 2019, pp. 41–68.

[17] O. Ozerk, C. Elgezen, A. C. Mert, E. Ozturk, and E. Savas, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," Cryptology ePrint Archive, Report 2021/124, 2021, https://ia.cr/2021/124.

[18] 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, p. 70–95, May 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/875

[19] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 264–275.

[20] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," Cryptology ePrint Archive, Report 2019/160, 2019, https://ia.cr/2019/160.

[21] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt, "Exploring energy efficient quantum-resistant signal processing using array processors," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 1539–1543.

[22] T. Ye, Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Fpga acceleration of number theoretic transform," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 98–117.