Parallelization of the Number Theoretic Transform

Steph Cheng, Jeriah Yu

April 2023

Abstract

The Fast Fourier Transform algorithm is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of an object, converting between its frequency and spatial domains. FFT and its variants, such as the Number Theoretic Transform (NTT), are essential to many areas of computing, including signal processing and computer arithmetic. In the context of post-quantum cryptography, the Ring Learning with Errors (RLWE) assumption enables the use NTT to perform fast polynomial operations, making lattice-based cryptography the leading candidate for efficient post-quantum cryptosystems. Our experiments seek to analyze the parallelization of the NTT algorithm, and in particular the extent to which different parallelization techniques can provide empirical speedups when combined with pre-existing mathematical optimizations.

1 Introduction

Lattice-based assumptions have been instrumental to the area of post-quantum cryptography, enabling the first successful instantiation of fully homomorphic encryption by Gentry in [1], among others. Beyond just theoretical achievements however, The Ring Learning with Errors (RLWE) assumption has allowed these constructs to be realized and implemented in real cryptosystems, with marked improvements over other classical techniques [2].

Underlying the RLWE assumption are large-degree polynomials, and efficiency of RLWE-based systems heavily depends on the speed of polynomial operations. While the naive representation of polynomials by their coefficients makes these operations slow, representing polynomials instead by their evaluations on specific points enables both polynomial addition and multiplication to be significantly faster and highly parallelizable.

However, to convert between a polynomial's coefficient form and evaluation form efficiently, an NTT operation must be performed. Moreover, after each multiplication, an intermediate conversion to and from coefficient form is required, so multiplications incur a large amount of overhead. However, in the particular case of RLWE, polynomials are elements of the cyclotomic ring $\mathbb{Z}_q[x]/(x^d + 1)$. This ring enables a special kind of optimization, known as "negacyclic convolution", which eliminates the need for any intermediate NTT invocations. Negacyclic convolution uses the NTT algorithm entirely as a blackbox, so we defer its details to [3].

2 Background

2.1 Coefficient Representation

Classically, polynomials are represented in terms of their coefficients:

$$f(x) = a_0 + a_1 x + a_2 x^2 \ldots + a_{n-1} x^{n-1}$$

In this representation, polynomial addition and multiplication can be computed as follows:

$$(f+g)(x) = \sum_{i=0}^{n-1}(a_i + b_i)x^i \qquad (f \cdot g)(x) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j x^{i+j}$$

As shown, polynomial multiplication requires a sum over all n^2 pairs of coefficients, which can be very expensive when the degree of these polynomials are high.

2.2 Evaluation Representation

A well known result about polynomials states that n points uniquely define a polynomial with degree at most $n-1$. Polynomial addition and multiplication can be alternatively thought of as satisfying

$$(f+g)(x) = f(x) + g(x) \qquad (f \cdot g)(x) = f(x) \cdot g(x)$$

for all x. Thus, by representing a degree $n-1$ polynomial, f, as a vector of evaluations $(f(x_1), f(x_2), \ldots)$, we can compute $f+g$ and $f \cdot g$ by vectorized addition and multiplication, respectively:

$$f+g = (f(x_1)+g(x_1), \ldots, f(x_n)+g(x_n)) \qquad f \cdot g = (f(x_1)g(x_1), \ldots, f(x_{2n})g(x_{2n}))$$

This reduces the time needed for a single multiplication by a factor of n. However, since $f \cdot g$ will have degree at most $2(n-1)$, we need to use more evaluation points in order to construct $f \cdot g$ uniquely. However, in the RLWE case, polynomials are taken modulo $x^d + 1$, which can only be computed efficiently in coefficient space. Thus, computing $f \cdot g$ naively with the method above requires a conversion to coefficients, a reduction modulo $x^d + 1$, and a conversion back into evaluation space. Note that negacyclic convolution, mentioned earlier, avoids this issue entirely.

2.3 Number Theoretic Transform

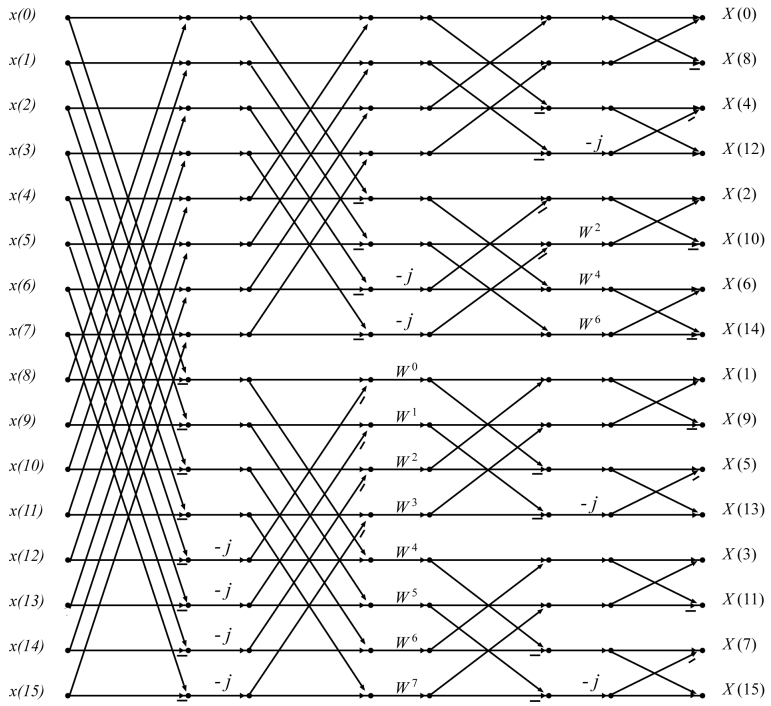
To convert between coefficient and evaluation representations, one could naively pick arbitrary values for $x_1, \ldots, x_n$ and evaluate the polynomial at these points. However, this would take $O(n^2)$ time, since evaluating the polynomial takes $O(n)$ time already.

Instead, we can use the Number Theoretic Transform. The key insight is to think of f as consisting of two polynomials, f_{even} and f_{odd}:

$$f_{\text{even}} = a_0 + a_2 x + a_4 x^2 + \ldots \qquad f_{\text{odd}} = a_1 + a_3 x + a_5 x^2 + \ldots$$

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots = f_{\text{even}}(x^2) + x \cdot f_{\text{odd}}(x^2)$$

Suppose two values, x_1 and x_2, satisfy $x_1^2 = x_2^2$. Then, we only need to compute f_{odd} and f_{even} on a single value, and combine the results to compute $f(x_1)$ and $f(x_2)$. By doing this recursively, and using roots of unity as values of x to maximize these shared results, we get the classic Cooley-Tukey butterfly algorithm:



In the serial case, this yields a work efficient algorithm that is best for most purposes. However, in a parallel setting, this benefit of sharing intermediate results is not as clear cut: sharing improves work efficiency, but requires synchronization and can impair parallelism.

2.4 Dimension and Radix

The algorithm outlined above is considered the 2-radix version of the Cooley-Tukey algorithm, but it can be generalized to other radices. In particular, f_{even} and f_{odd} can be generalized to get the k-radix algorithm:

$$f(x) = f_0(x^k) + x \cdot f_1(x^k) + x^2 \cdot f_2(x^k) + \ldots + x^{k-1} f_{k-1}(x^k)$$

With radix k, each of the n nodes on each layer will compute a sum of k values, and the total number of layers will be $\log_k n$. Thus, the total runtime is $O(nk \log_k n)$. For constant values of k, this is work efficient, and $k = 2$ minimizes the constant factor. However, for large values of k comparable to n, this algorithm is no longer work efficient – the number of layers, $\log_k n$, will be constant, and the runtime is about $O(n^2)$. Note that the number of layers is known as the dimension, and the tradeoff between radix and dimension roughly corresponds with the tradeoff between work efficiency and parallelizability. In particular, the 2-radix, high-dimension NTT is the optimal serial algorithm, whereas the high-radix, 1-dimension NTT corresponds to the naive evaluation of all points.

3 Parallelization and Implementation of NTT

As mentioned earlier, the goal of our work is to examine the degree to which the NTT algorithm can be parallelized, and in particular with respect to the RLWE setting of high-degree polynomials in the cyclotomic ring. Since the negacyclic convolution is widely used in this setting, we found it fitting to test how it can impact these parallel implementations. Specifically, for all of the implementations of NTT (namely, Serial, Multicore, and GPU), we also implemented a version using the negacyclic convolution. As a reminder, negacyclic convolution is a black-box, mathematical optimization that reduces the number of intermediate NTTs necessary. Thus, our negacyclic variations use the exact same underlying NTT algorithm as their standard counterparts, to highlight as best as possible how the reduced number of NTTs will effect the empirical speedup.

The overall testing suite, as well as most of the algorithms themselves, were implemented in Rust. Rust has become a popular language in applied cryptography research, since it can provide memory safety guarantees while still remaining competitive in speed to C and C++. Moreover, for our work, Rust's type system would help to ensure correctness in a concurrent environment, and could be easily leveraged to correctness tests and performance benchmarks that generalized over all of our implementations. However, this did not come without some drawbacks, which will be discussed shortly.

In the next sections, we briefly detail each specific implementation of NTT that we ended up testing, as well as any implementation techniques or workarounds needed. We also provide a brief background on the motivation for each implementation. These implementations are presented roughly in order from most work efficient to least work efficient.

3.1 Serial NTT

For the baseline, best serial algorithm, we implemented the standard radix-2 Cooley-Tukey algorithm. The specific implementation follows standard practices for implementing fast NTT: it uses an iterative implementation rather than a recursive one, computes the NTT in-place directly on the array, and precomputes lookup tables for commonly used powers of the roots of unity.

3.2 Multicore NTT

The multicore, shared-memory implementation of NTT attempts to parallelize the radix-2 Cooley-Tukey algorithm directly. Each layer requires the computation of n values, but these values are computed in independent pairs. This leads to a straightforward and efficient approach to parallelization, with optimal load balancing: for each layer, divide the work evenly so that each thread works on the same number of pairs, and synchronize on each layer. The only drawback to this approach comes from Rust itself — namely, even though the algorithm has multiple threads accessing different indices, Rust requires some form of mutual exclusion guarantee on the shared array. Thus, the multicore implementation must use atomic integers and load/stores, which may incur a bit of overhead.

3.3 GPU NTT

For the GPU implementation, we switch to the 1-dimension FFT, which is equivalent to naively evaluating $f(x)$ on all inputs. Though this algorithm is work-inefficient, it avoids synchronization entirely, opting instead to leverage the GPU's massively parallel architecture to compute all $f(x)$ at once. Though Rust does not natively support CUDA, the RustaCUDA crate provides an interface to transfer memory to the GPU and launch CUDA kernels. Moreover, it tries to provide as many of the Rust safety guarantees as possible.

3.4 Trivial Serial Implementation

For correctness purposes, we also implemented the trivial, serial polynomial operations. Addition is computed as a sum of coefficients, and multiplication is the grade school polynmomial multiplication, outlined in Section 2.1. The original intent was to use the rustFFT crate, which utilizes AVX instructions to provide some parallelism, as the starting point for our other implementations. However, we quickly discovered that rustFFT only supports the general FFT algorithm over complex numbers, and is not suited for the specific case of integers in the NTT.

3.5 Trivial GPU Implementation

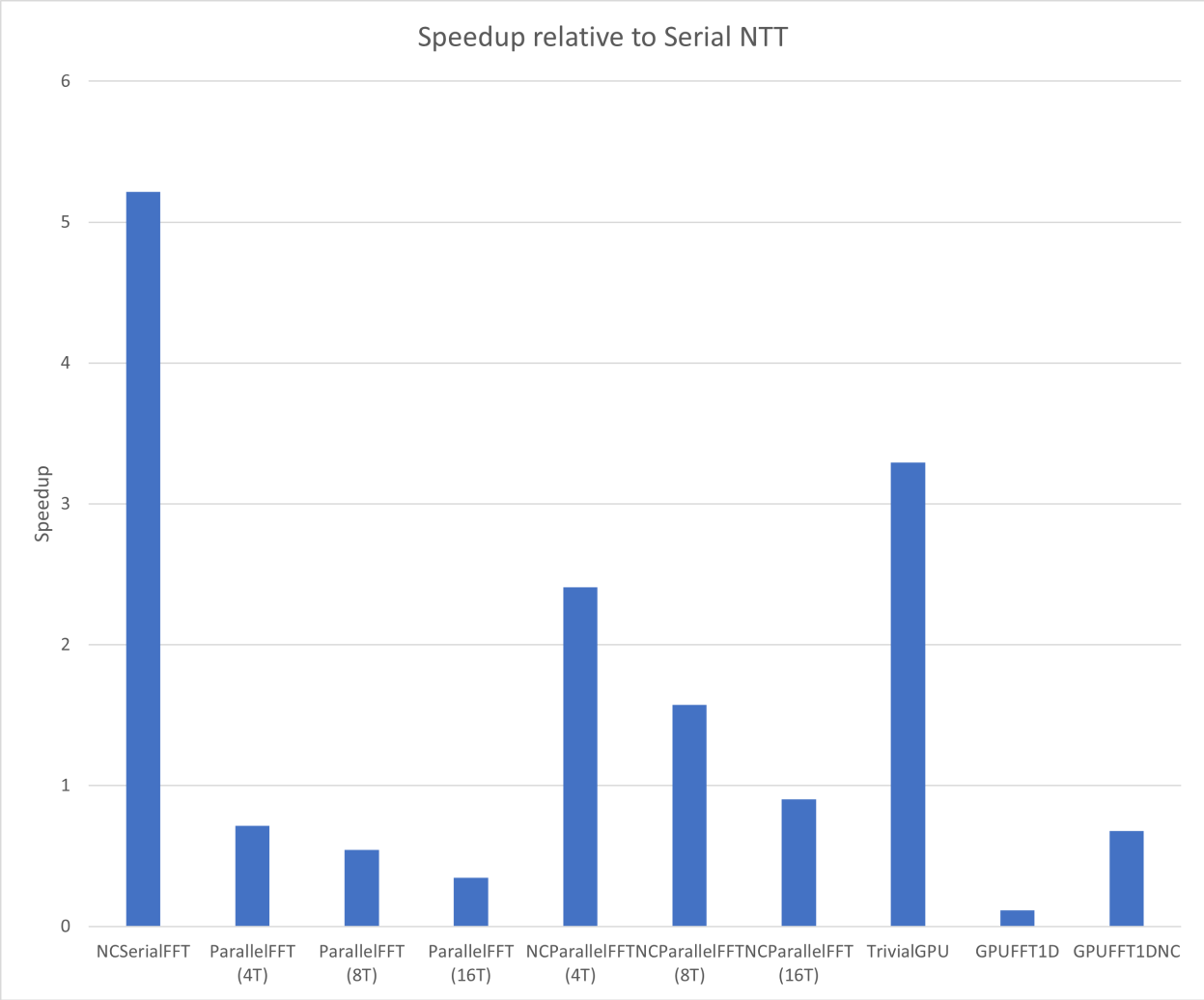
To ensure correctness of the GPU interactions, we also implemented the same trivial polynomial operations on the GPU. Unlike the serial case, the trivial GPU implementation is not unreasonable as an algorithm itself, since it avoids all NTT operations entirely in favor of having highly parallelizable operations. However, the trivial implementation also cannot benefit from negacyclic convolution, like the other implementations can.

4 Experiments

4.1 Testing Methodology

After verifying correctness, we benchmarked our implementation under realistic cryptographic load: generating 100 2048-degree cyclotomic polynomials and multiplying all of them together. This benchmark was run against all the developed algorithms (except the trivial serial implementation, which was only used for correctness). We chose to measure the end to end time of the entire process: starting with the initialization and precomputations for NTT, then converting all the polynomials via NTT (if necessary), computing their product, and ending after converting the result back via inverse NTT if necessary. This process was repeated 5 times per algorithm and averaged. Additionally, for NTT implementations, we also tested both the regular version and the negacyclic convolution version. For the multicore implementations, we also vary the number of threads to observe how it affects speedup.

4.2 Results



We defer specific analysis and speculation of the results to the next section, but there are a few patterns in our measurements to note. Firstly, the negacyclic convolution provides the most consistent speedup across all algorithms, which aligns with our expectations; negacyclic convolution is a black-box optimization that reduces the number of intermediate NTTs, which are the most expensive part of the polynomial multiplication pipeline. Secondly, most of the parallel implementations do not see any speedup over their serial counterparts, suggesting that the task is not compute bound. This is further supported by the fact that using more threads for our multicore implementations yields worse speedups.

5 Analysis

In this section, we provide a case analysis of each group of algorithms we implemented, and also provide some closing remarks on the overall outcome of our work.

5.1 Serial Algorithms

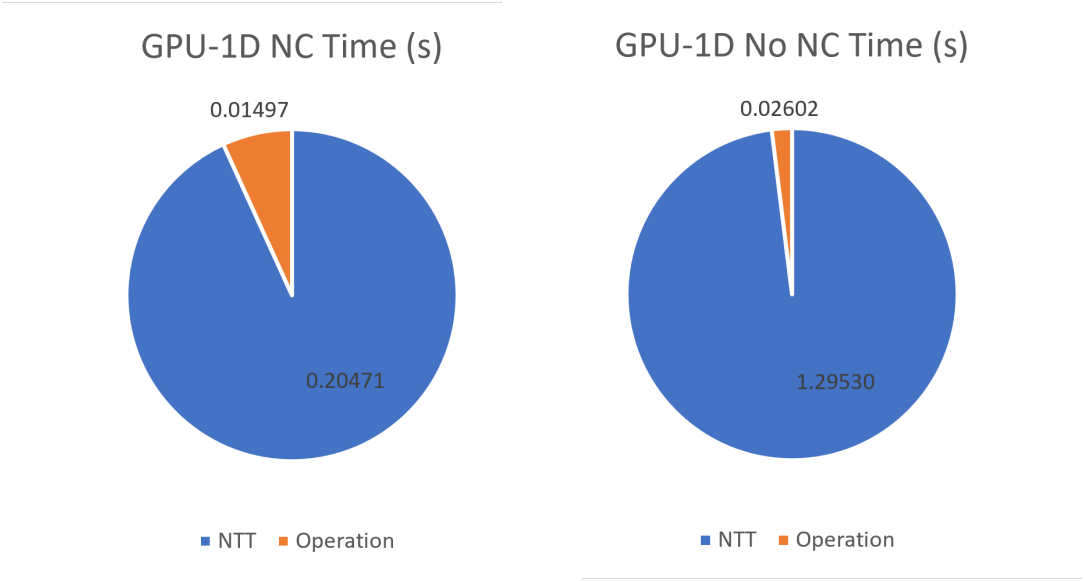
We found that the best performing algorithm was the serial NTT with negacyclic convolution. These results are similar to that of the parallel scan lab, in which we found that the overheads of communication were too costly to see massive speedups. Furthermore, the results of the serial NTT implementations shows that negacyclic convolution alone provides a speedup of over 5x, shockingly faster than what we had expected. Part of this speedup, which we unfortunately did not have time to look into, could potentially also be attributed to the Rust compiler; since the algorithm itself is largely iterative and only uses standard arithmetic operations, the Rust compiler may have been able to optimize much of the operations.

5.2 Multicore Algorithms

In the multicore setting, we observe that the implementations get slower as more threads are added, in both the standard and negacyclic convolution variations. Though the multicore implementation provides near optimal load balancing, unlike the parallel scan, the NTT algorithm still requires large amounts of sharing data across the different processes, leading to many cache evicitions and misses. These results indicate that the NTT operation is too interconnected and not computationally intensive enough to benefit largely from parallelism (at least in the work efficient setting).

5.3 GPU Algorithms

The trivial algorithm on the GPU performs surprisingly well, greatly exceeding our expectations. In fact, it is the only parallel algorithm that beats its serial counterpart. (Multicore NTT with negacyclic convolution is faster than the standard serial NTT, but is still slower than the serial NTT that uses negacyclic convolution). It has much better coalescing of memory access contiguously through an array, while the other GPU implementations require more random access due to the modular division of index numbers. Moreover, its computations are entirely independent and can be parallelized very effectively. The trivial algorithm also provides a good baseline for how long GPU-related operations take, such as transferring memory to and from the device. However, the trivial GPU algorithm is unfortunately unable to take advantage of the negacyclic convolution.



For the 1D NTT, we found that the NTT operations alone take up the vast majority of computing time in the full algorithm. For both GPU-1D with negacyclic convolution and without, 93.18% and 98.03% of the time is spent on the NTT CUDA kernel, which use the same underlying algorithm. The stark contrast in the performance of the two algorithms therefore boils down to how many times NTT is run and with what size arrays. When analyzing just the running time of 100 iterations of CUDA NTT, it took 0.2 seconds with negacyclic convolution and 0.4 seconds without. This perfectly reflects the fact that negacyclic convolution cuts down the degree of the polynomial by exactly half. Furthermore, since negacyclic convolution eliminates intermediate NTTs, the benefits of negacyclic in the 1D-GPU setting are drastic.

5.4 Closing Remarks

The biggest takeaway from our research is the importance of the negacyclic convolution; by avoiding redundant transformations between coefficient and evaluation space after each multiplication, and reducing the degree of polynomials overall, negacyclic convolution provides major speedups across the board. Our attempts to speed up NTT through the use of different parallel architectures were largely unsuccessful all around, signifying that the structure of the NTT algorithm itself may not be suitable for major parallelism speedups.

However, it is also important to note that the negacyclic convolution is only applicable in the cyclotomic rings used by RLWE. Though the focus of our work was to determine which algorithms would provide the best speedup for this cryptographic setting, the results for implementations without negacyclic convolution readily generalize to broader areas in which FFT may be used.

Future improvements include testing on FPGAs, implementing additional low-dimensional (2D) work-efficient GPU FFT algorithms beyond 1D to test, and further optimizing our GPU and multicore implementations. Additional benchmarking subroutines for different settings, such as having a single polynomial be transformed, continuously multiplied to itself, and then converted back. These subroutines may provide better insights into where various parallel implementations may excel, since this should give a major advantage to the negacyclic algorithms as the vectorized multiplications are easily parallelized on a GPU, without having to worry about the costly NTT and inverse NTT conversions necessary for modular reduction. Finally, though our work tries to model the cryptographic setting as closely as possible, it is not a perfect reflection of what true cryptographic system would do; in particular, many other areas of the system, such as random sampling and matrix multiplication, can also be parallelized, and it very well may be the case that all of these optimizations combined would yield an overall speedup for parallel algorithms.

References

[1] Craig Gentry. (2008) *Computing Arbitrary Functions of Encrypted Data.*

[2] Hao Chen, Kim Laine, and Peter Rindal. (2017) *Fast Private Set Intersection from Homomorphic Encryption.*

[3] Patrick Longa and Michael Naehrig. (2016) *Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography.*