A Practical Introduction to

LUCASNTT

Guillaume P. Hérault
Geneva, Switzerland

August 29, 2025

ABSTRACT

Part of the `LucasNTT` project ([1]), the `LucasNTT` program is a GPU-accelerated implementation of the Lucas-Lehmer primality test for large Mersenne numbers using Number-Theoretic Transforms (NTTs) in the finite field $\mathbb{Z}/p\mathbb{Z}$. The program combines several techniques including the Pease FFT algorithm, Irrational Base Data Weighted Transforms (IBDWT), and multi-step FFT plans, all optimized for parallel execution with `Cuda` Nvidia GPUs. Performance evaluations demonstrate that, despite the intrinsic overhead of modular arithmetic, careful algorithmic choices can significantly reduce the gap with Complex FFT implementations.

1 Introduction

Prime numbers have fascinated people for centuries — not only mathematicians, but also the general public, due to the simplicity of their definition. Despite the elementary nature of their definition, prime numbers lie at the heart of several unresolved problems in mathematics, particularly concerning their distribution: the Riemann Hypothesis, Goldbach's Conjecture, and the Twin Prime Conjecture, to name a few. For centuries, people have sought efficient algorithms for primality testing and have taken part in the ongoing quest to discover the largest known prime number.

As of today, the seven largest known primes are all Mersenne numbers of the form $2^q - 1$, discovered by the GIMPS large-scale distributed project ([2]). The Lucas-Lehmer test is used to determine whether such a number is prime.

Let $M_q = 2^q - 1$ be the Mersenne number to test, with q prime. Define a sequence by $x_0 = 4$ and $x_{k+1} = x_k^2 - 2$.

Then M_q is prime if and only if

$$x_{q-2} = 0 \mod M_q$$

The sequence begins as follows: 4, 14, 194, 37634. After a few iterations, the values quickly exceed M_q, so modular reduction is applied at each step to keep the numbers manageable. The test itself is conceptually simple, but it requires repeatedly squaring and reducing very large integers modulo M_q. To perform these computations in a reasonable time, fast multiplication algorithms must be employed.

Currently, all major implementations rely on FFT-based multiplication in the complex field. However, floating-point operations inevitably introduce round-off errors, even in double precision. Significant efforts have been made to minimize these inaccuracies. As a result, there is no doubt about the primality of the largest known primes.

Nevertheless, no formal proof relying solely on pure-integer calculations has ever been performed. The LucasNTT Cuda program aims to provide a solid, efficient, and reliable method for running the Lucas-Lehmer primality test on giant Mersenne numbers, using exclusively pure-integer arithmetic.

2 Fast multiplication with pure-integer arithmetic

To implement the Lucas-Lehmer test, it is necessary to repeatedly square very large integers. For that purpose, LucasNTT is using Number Theoretic Transforms (NTT) in $\mathbb{Z}/p\mathbb{Z}$, where

$$p = 2^{64} - 2^{32} + 1$$

No Chinese Remainder Theorem ([3]), no Schönhage and Strassen ([4]) implementation: the NTT multiplication algorithm is as straightforward as complex FFT-based multiplication (described by Crandall in [5], p. 490). We simply use a forward NTT, followed by a pointwise square, followed by a backward NTT.

This is possible due to the unique attributes of this modulus. For a modulus to be *NTT-Friendly* with a transform of length n, a primitive n-th root of unity modulo p must exist, and n must divide $p - 1$. This is the case here. Since $p - 1 = 2^{32} \times 3 \times 5 \times 17 \times 257 \times 65\,537$ this modulus enables NTT lengths up to 2^{32}, with optional factors 3 and 5. There exist several primitive $(p - 1)$th roots of unity: 7, 554, among others. LucasNTT uses 554, so that the n-th roots of unity are given by $554^{(p-1)/n} \mod p$. The choice of 554 is motivated by the fact that $554^{(p-1)/64} \mod p = 8$, meaning that FFT of length 2^6 can be done with shifts. This opens doors for further optimizations.

In addition, the special form $2^{2k} - 2^k + 1$ enables fast computation of $x \times y \mod p$, avoiding costly modular reduction algorithms, as explained by Solinas ([6]) and in the companion document ([7]).

Today, the finite field $\mathbb{Z}/(2^{64} - 2^{32} + 1)\mathbb{Z}$, often referred to as the Goldilocks field, has become widely used in modern blockchain systems, notably within the Ethereum ecosystem, as it provides an efficient arithmetic environment for cryptographic protocols based on zero-knowledge proofs ([8] [9] [10]).

Surprisingly, this modulus was already mentioned in 1976 by Liu et al ([11]). In the context of NTT and the Lucas-Lehmer test, an efficient implementation was provided by Nick Craig-Wood ([12] [13]), whose code served as a valuable source of inspiration for the present work. I would like to acknowledge his contribution, which greatly facilitated my own research.

3 Reducing modulo q with an IBDWT

During the Lucas-Lehmer test, it is necessary to reduce the result modulo M_q, the Mersenne number. One method is to use a double-length DFT, split the result exactly after q bits and add the two halves together ([6]. Another method, which does not need a double-length DFT, takes advantage of the fact that Discrete Fourier Transforms naturally perform a cyclic convolution, which is equivalent to a reduction modulo $\beta^n - 1$, where β is the base used to represent input numbers.

It turns out that by adjusting the base of the input numbers, the cyclic convolution can effectively perform a reduction modulo $M_q = 2^q - 1$. This approach is known as the *Irrational Base Data Weighted Transform* (IBDWT), introduced by Crandall ([14]). The idea is to represent the input numbers using a variable base as close as possible to the theoretical base $2^{q/n}$, which is impractical because q/n is irrational, but would naturally perform a reduction modulo $2^q - 1$.

To achieve the desired reduction with this variable base, it is necessary to multiply each element by a weighting factor, so that the convolution in the variable base behaves like an unweighted convolution in the irrational base. As a result, the convolution directly performs a reduction modulo M_q, without the need to double and zero-pad the DFT input.

Historically, the IBDWT has been applied in the complex field. But it can also be performed in $\mathbb{Z}/p\mathbb{Z}$ by weighting the input signal with n^{th} roots of *two* modulo p ([15] [16]).

However, using an IBDWT introduces two consequences. Firstly, a primitive n-th root of two must exist in $\mathbb{Z}/p\mathbb{Z}$. This requirement can limit the possible length n of the transform. Secondly, because of the weighting factors, overflows may occur more easily during the convolution, which means that the transform length must sometimes be reduced.

Let us now examine these consequences when using our chosen modulus $p = 2^{64} - 2^{32} + 1$. With a suitable algorithm ([17]), it can be shown that the primitive n-th root of two is

$$b = 7^{5(p-1)/192n}$$

And since

$$5(p-1)/192 = 2^{26} \times 5^2 \times 17 \times 257 \times 65,537$$

the maximum length of the NTT that can be performed with an IBDWT is 2^{26}, with an optional factor 5 (instead of 2^{32} with an optional factor 3). LucasNTT does not implement yet the FFT-5 and is currently limited to powers of two. However, with a 2^{26}-point NTT, each element would carry 18 significant bits, allowing us to test Mersenne numbers with up to $18n - 1 = 1,207,959,551$ bits, far beyond the size of the largest known prime.

4 Algorithm

We can now describe the algorithm used to square a large integer modulo M_q in $\mathbb{Z}/p\mathbb{Z}$.

Algorithm 1. Square modulo q
Given a large integer x with q significant bits, the following algorithm computes $x^2 \mod M_q$ using pure-integer arithmetic in $\mathbb{Z}/p\mathbb{Z}$.

```
Pre-computation steps:
[1] Calculate n, the minimum length of the NTT to avoid overflow.
[2] Compute the n-th roots of unity in Z/pZ, and their inverses.
[3] Compute the IBDWT weights, and their inverses.
[4] Determine the widths of the variable base for the IBDWT.

Squaring steps:
[5] Split the input number into n chunks, according to the variable widths.
[6] Apply the calculated weights to each element.
[7] Perform a forward NTT using the precomputed n-th roots of unity.
[8] Square each element pointwise.
[9] Perform the backward (inverse) NTT using the inverse n-th roots of unity.
[10] Unweight each element using the precomputed inverse weights.
[11] Convert the result to the variable base representation via a base change with carry propagation.
```

It should be noted that during the Lucas-Lehmer test, the four pre-computation steps are performed only once. In contrast, steps [5] to [11] are executed millions of times. To minimize costly bit manipulations, the `LucasNTT` program retains the result in the same variable base throughout all iterations: the final step of each iteration performs a carry propagation and simultaneously prepares the input for the next iteration, eliminating the need to repeat step [5].

As a result, each `LucasNTT` iteration reduces to a seven-step process:

```
Weight, Forward NTT, Pointwise Square, Backward NTT, Unweight, Propagate carry, Minus 2
```

Among these seven steps, five scale efficiently as n increases and are straightforward to implement. The two critical and computationally intensive steps are the forward and backward Fourier transforms. Consequently, the initial objective is now reduced to the problem of accelerating the NTT computation.

5 Boosting the NTT with Fast Fourier Transforms

Since no existing FFT library operates directly in $\mathbb{Z}/p\mathbb{Z}$, we must implement our own FFT. This task is often considered complex, as it requires a solid understanding of various FFT algorithms, which are typically described in academic papers, sometimes difficult to read. Additionally, most FFT algorithms are initially presented in the Complex field, making it unclear which of them can also be applied in a finite field setting. Even today, hundreds of papers related to FFT algorithms are published every year, making it challenging for newcomers to classify them and understand how a particular algorithm fits within the broader FFT literature.

The most common, widely discussed, and implemented FFT algorithms are what we refer to as *Radix algorithms*, among which the Cooley-Tukey algorithm ([18]) is the most notable example. These algorithms emerged in the 1960s and form the foundational basis for almost all FFT implementations today. They are fundamental in the sense that, although large FFTs are typically built from smaller FFTs for performance reasons using various techniques, they

eventually rely on a Radix algorithm involving *butterfly* calculations. A butterfly operation is essentially a small DFT, either pre-multiplied (in decimation-in-time) or post-multiplied (in decimation-in-frequency) by twiddle factors.

In this family of FFT algorithms, the `LucasNTT` program uses the *Pease algorithm*. This algorithm, due to its simplicity and regularity, is relatively easy to implement. It was published by Pease in 1968 [19], following earlier discussions with Singleton, who had already described the method in [20]. Unlike the Cooley–Tukey algorithm, which operates in-place, or the Stockham algorithm ([21] [22]), which is autosorting, the Pease algorithm is neither. It is out-of-place and relies on bit-reversed addressing. These characteristics may explain why the Pease algorithm has received comparatively less attention in the literature, with the notable exception of the work by Korn and Lambiotte [23]. However these two drawbacks can be mitigated in practice. Firstly, a bit-reversed output is not a problem when computing a convolution, as the subsequent inverse transform will reorder the results as expected. Programmers must, however, carefully manage the twiddle factors when assembling multiple FFTs together. Secondly, it is true that the Pease algorithm is naturally out-of-place when implemented iteratively. But in a parallelized execution, where threads are synchronized to operate concurrently, it can be efficiently implemented in-place. The `LucasNTT` implementation avoids using a double buffer for FFTs.

To further improve memory locality and performance, large FFTs are split and computed in 2 or 3 stages using a row-column decomposition approach ([24]. For example, to compute a 2^{21}-point FFT, `LucasNTT` first performs 2048×2^{10}-point FFTs. Each FFT of size 2^{10} is loaded in shared memory, by accessing data in columns. The output is then multiplied by a twiddle factor (requiring careful handling of bit-reversal), and the computation proceeds with 1024×2^{11}-point FFTs, this time accessing data in rows. This 2^{21}-point FFT process is denoted as a $1024 : 2048$.

6 Coding LucasNTT Program

The chosen programming language is `C++` combined with `Cuda` technology, as general-purpose computing on graphics processing units (GPGPU) appears to be the best option to achieve high performance with parallel processing. In this GPGPU architecture, it is crucial to minimize data transfers between the CPU and the GPUs. Within the Lucas-Lehmer loop, the role of the CPU is reduced to synchronizing the threads by simply launching the kernels. Throughout the entire computation, data remains in the global memory of the GPU, avoiding costly transfers back and forth with the CPU.

When starting to code an FFT with `Cuda`, it is not obvious to predict in advance which FFT dataflow will offer the best performance. For example, when computing a 2^{22}-point FFT, should we opt for a 2-step decomposition (such as $2024 : 2048$) or a 3-step decomposition (such as $128 : 128 : 256$)? Is it faster to assign each thread to compute a single Radix-2 butterfly, or to reduce the number of threads by increasing the radix, or by assigning multiple butterflies to each thread?

All these hypotheses have been implemented and are evaluated at runtime. The `LucasNTT` program is *auto-tuning*, meaning that it measures the performance of each kernel configuration and automatically selects the fastest option. Further details and explanations regarding the program's implementation and source code are available in the companion document on the `LucasNTT` website ([7]).

7 Performance Results

The main objective of the `LucasNTT` program is to provide a formal proof once a probable-prime has been found using a PRP method. Therefore, performances do not matter as much as for a program whose main objective is prospecting new primes.

And NTTs cannot compete with Complex FFTs. The author has compared the performance of a single multiplication of two complex numbers versus two 64-bit integers modulo p. The multiplication in $\mathbb{Z}/p\mathbb{Z}$ takes roughly 3 times more clock cycles. Moreover, it also requires 3 times more `Cuda` registers, reducing occupancy. NTTs and Complex FFTs definitely do not compete in the same category.

However, we are able to close the gap with a Complex FFT thanks to optimizations of the FFT algorithms. Some performance results have been gathered in an Excel file ([25]). The author successfully proved the primality of $2^{82,589,933}$ in 8 hours 48 minutes, achieving 0.38 milliseconds per iteration with a `NVIDIA RTX 5090` (Ubuntu 22.04, Cuda 12.8).

8 Conclusion

This paper presented the design and implementation of `LucasNTT`, a program dedicated to proving the primality of large Mersenne numbers using the Lucas-Lehmer test, entirely within the finite field $\mathbb{Z}/p\mathbb{Z}$ and accelerated by Nvidia GPU architectures with `Cuda`.

The core contribution lies in adapting techniques traditionally used in the Complex domain (such as the FFT algorithms and the Irrational Base Data Weighted Transform) to an integer-only context, enabling efficient Number Theoretic Transforms (NTTs). The program's ability to auto-tune its FFT configurations, and its memory-efficient implementation of multi-step FFT plans, have allowed it to close the performance gap with complex-domain FFT programs to an unexpected extent.

While the performance of NTTs remains inherently lower than their Complex counterparts due to the higher cost of modular arithmetic, the experiment has shown that careful algorithmic and architectural choices can mitigate this limitation.

An interesting open question remains: what would be the performance of a `Cuda` program implementing exactly the same Pease-based FFT architecture, with the same multi-step and transpose strategy, but operating on Complex numbers instead of integers in $\mathbb{Z}/p\mathbb{Z}$? Such a comparison could provide deeper insights into the relative cost of arithmetic operations in both domains, and perhaps reveal further optimization strategies transferable between them.

References

[1] Guillaume P. Hérault. LucasNTT Website. https://lucasntt.github.io, 2025.

[2] GIMPS - Great Internet Mersenne Prime Search. https://www.mersenne.org/, 1996.

[3] J. M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25(114):365–374, 1971.

[4] Schönhage and Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7(3–4):281–292, September 1971.

[5] Richard Crandall and Carl Pomerance. *Prime Numbers*. Springer-Verlag, 2005.

[6] Jerome A. Solinas. Generalized Mersenne Numbers, 1999.

[7] Guillaume P. Hérault. LucasNTT Website - Documents. https://lucasntt.github.io/documents.html, 2025.

[8] Polygon Zero. Plonky2: Fast recursive snarks. https://github.com/0xPolygonZero/plonky2, 2022.

[9] Polygon Labs. Polygon zkevm documentation. https://wiki.polygon.technology/docs/zkEVM/overview, 2023.

[10] zkSync Team. Boojum upgrade: zksync era's new high-performance proof system for radical decentralization. https://zksync.mirror.xyz/HJ2Pj45EJkRdt5Pau-ZXwkV2ctPx8qFL19STM5jdYhc, July 2023.

[11] K. Y. Liu, I. S. Reed, and T. K. Truong. Fast Number-Theoretic Transforms for Digital Filtering. *Electronics Letters*, 12(24):644, 1976.

[12] Nick Craig-Wood. Integer DWTs mod $2^{64}-2^{32}+1$. https://www.craig-wood.com/nick/armprime/math/, 1994.

[13] Nick Craig-Wood. iprime: All-integer mersenne prime checker. https://github.com/ncw/iprime, 2014.

[14] Richard Crandall and Barry Fagin. Discrete Weighted Transforms and Large-Integer Arithmetic. *Mathematics of Computation*, 62(205):305–324, 1994.

[15] Peter-Lawrence Montgomery. Using dft's modulo a prime (long). https://www.mersenne.org/various/intfft.txt, July 8 1996.

[16] Guillaume P. Hérault. LucasNTT Website - Fast Multiplication. https://lucasntt.github.io/multiplication.html, 2025.

[17] Guillaume P. Hérault. LucasNTT Website - NTT Analysis. https://lucasntt.github.io/nttanalysis.html, 2025.

[18] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–297, may 1965.

[19] Marshall C. Pease. An Adaptation of the Fast Fourier Transform for Parallel Processing. *Journal of the ACM*, 15(2):252–264, apr 1968.

[20] Richard C. Singleton. A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage. *IEEE Transactions on Audio and Electroacoustics*, 15(2):91–98, jun 1967.

[21] Thomas G. Stockham. High-speed convolution and correlation, 1966.

[22] William T. Cochran, James W. Cooley, David L. Favin, Howard D. Helms, Reginald A. Kaenel, William W. Lang, George C. Maling, David E. Nelson, Charles M. Rader, and Peter D. Welch. What is the Fast Fourier Transform. *Proceedings of the IEEE*, 55(10):1664–1674, 1967.

[23] David G. Korn and Jules J. Lambiotte. Computing the Fast Fourier Transform on a Vector Computer. *Mathematics of Computation*, 33(147):977–992, 1979.

[24] David H. Bailey. FFTs in External or Hierarchical Memory, 1989.

[25] Guillaume P. Hérault. LucasNTT - FFT Lengths And Performances. https://github.com/LucasNTT/documents/tree/main/Performances, 2025.