A Practical Introduction to

# LucasNTT Code

**Guillaume P. Hérault**
Geneva, Switzerland

September 2, 2025

## 1 Introduction

Part of a wider project ([1]), LucasNTT is a C++/Cuda program implementing the Lucas-Lehmer primality test for Mersenne numbers. It translates the mathematical principles described in the companion document ([2] into an efficient and modular source code. LucasNTT was not designed to prospect for new primes, but to rigorously prove the primality of a given candidate using integer arithmetic with no round-off errors.

Beyond its use in primality testing, this document aims to present how to implement a Number Theoretic Transform (NTT) efficiently on GPU hardware. Since the programming strategies and design patterns used for NTTs are largely similar to those used for complex FFTs, the code structure and optimization techniques presented here are of broader interest. Students and researchers working on fast multiplication algorithms or high-performance Fourier transforms may find useful insights, even outside the context of number theory.

The following sections describe how to compile and execute the program, how the code is structured, and how the mathematical operations are mapped onto the GPU. Particular attention is given to readability, maintainability, and performance, with a detailed explanation of the role of each class and the design choices made throughout the implementation.

## 2 Compiling LucasNTT

The LucasNTT program is written in standard C++11 and CUDA, with no external library dependencies. It can be built either from the command line using provided batch scripts, the provided `Makefile` or from within Visual Studio on Windows for debugging purposes.

Two compilation scripts are included in the repository: one for Windows (`_build_windows.cmd`) and one for Linux (`_build_linux.sh`). These scripts scan the source directory and build all files in the directory. On Linux, the `make` command can simply be used instead.

**Please modify these scripts according to your needs, by specifying the compute capability of the targeted GPU at the top of the file.**

The minimum software requirements are CUDA Toolkit 11.4 and a GPU with Compute Capability 3.5. However, to benefit from recent performance improvements and features (such as better register usage, reduced launch latency, and improved occupancy), it is strongly recommended to compile with CUDA Toolkit 12.8 and execute on a Nvidia GPU with Compute Capability 7.0 or higher.

## 3 Execution

The LucasNTT program can be executed via the command line, with optional parameters that override the default values specified in an optional `LucasNTT.ini` file. A sample `.ini` file is included in the repository and documents all configurable options, such as output paths, FFT layout, and iteration control.

By default, running the program with a single parameter launches a full Lucas-Lehmer test:

```
LucasNTT 9689
```

This command tests the primality of the Mersenne number $2^{9689} - 1$.

To execute a fixed number of iterations without completing the full test:

```
LucasNTT 82589933 1000
```

This runs 1000 Lucas-Lehmer iterations on $2^{82589933} - 1$.

To specify the FFT plan and output files explicitly:

```
LucasNTT 82589933 1000 "/lucasntt/results/file" "2048:2048"
```

This uses a two-step FFT plan of size $2^{11} \times 2^{11}$, and creates output files such as `/lucasntt/results/file.txt` and `/lucasntt/results/file.env`.

Indeed the program can generate three output files: the `.txt` file contains runtime statistics, the chosen FFT path and the primality result. The `.env` file contains hardware/environment info. The `.dat` file contains the residue of the Lucas-Lehmer test, in a readable hex format. The residue is dumped every hour for safety reasons, so that we can continue the calculation after a crash without restarting from the beginning. The program also handles the manual `CTRL-C` interruption, saves the residue, so that you can continue the process without restarting from the beginning.

If no arguments are provided at command line, all values are taken from `LucasNTT.ini`. Command-line parameters always override the corresponding entries in the `.ini` file.

## 4 Coding Lucas-Lehmer Test

For the sake of illustration, let us detail how we can test with certainty the primality of $M_q = 2^q - 1$ where $q = 82{,}589{,}933$, in $\mathbb{Z}/p\mathbb{Z}$ with $p = 2^{64} - 2^{32} + 1$. Although `LucasNTT` is able to run the pseudo-primality PRP test by setting `PRP=1` in the `.ini` file, we describe here the implementation of the regular Lucas-Lehmer test. Starting with $x_0 = 4$, we compute repeatedly $x_{k+1} = x_k^2 - 2 \mod M_q$. We leverage Fast Fourier Transforms (FFT) to efficiently square $x$.

Let $n$ be the total length of the FFTs used during this process. We split the binary representation of $x$ into $n$ chunks. Since $x$ has $q$ significant bits, each chunk can have up to $\lceil q/n \rceil$ bits, meaning each chunk can hold values up to $m = 2^{\lceil q/n \rceil} - 1$. The first step is to compute the smallest power-of-two $n$ such that the inequality $2nm^2 < p$ holds. This condition ensures that there is no overflow during the calculation of $x^2$.

After reading the command-line parameters and the `.ini` file, the `MainProcess` class launches the calculation of $n$ with the help of the `NttLengths` class. For $q = 82{,}589{,}933$, we compute $n = 2^{22} = 4{,}194{,}304$ (4M). Each element of the FFT thus holds at most $\lceil q/n \rceil = 20$ significant bits. We then allocate a memory buffer $x$ as an array of $n$ 64-bit words in global GPU memory. Although all FFTs are computed in-place, a double-sized buffer must be allocated to perform matrix transpositions. In this example, the memory buffer is $4M \times 8 \times 2 = 64$ MB. Each word initially contains only 19 or 20 significant bits at the beginning of a Lucas-Lehmer iteration, but all 64 bits may be used during the FFT computation. At the end of each iteration, a carry-propagation step ensures that each chunk returns to 19–20 significant bits, ready for the next iteration. This is the role of the `reduce2base` kernel.

To reduce $x$ modulo $M_q$ at each iteration, `LucasNTT` uses the IBDWT algorithm ([3]). The chunk sizes (19 or 20 bits) are chosen to approximate the irrational theoretical base $2^{q/n}$ as closely as possible. These widths are precomputed and stored in GPU global memory before starting the Lucas-Lehmer process, along with the IBDWT weights that are applied to each element prior to the FFT. To complete the precalculation step, kernels derived from `Ntt` precompute the $n^{\text{th}}$ roots of unity, and kernels derived from `NttFactor` precompute the twiddle factors. These twiddle factors are stored in bit-reversed order to allow efficient coalesced memory access during later computation.

During the process, all arithmetic operations are performed modulo $p$. Modular addition and subtraction are straightforward, requiring at most one correction step. Multiplication is more involved, as using the `%` operator or even fast reduction algorithms ([4] [5] [6]) would be too costly. Instead, `LucasNTT` exploits the special structure of $p$. Notably, $2^{96} \equiv -1 \mod p$ and $2^{64} \equiv 2^{32} - 1 \mod p$, which allows the high and middle bits of a 128-bit product to be efficiently folded back into the 64-bit range ([7] [8]). As a result, multiplication modulo $p$ is achieved using only two multiplications, three additions, and three conditional tests.

Once the precalculation phase is complete, LucasNTT evaluates the performance of multiple candidate FFT configurations. In the previous example, FFTs of length $n = 2^{22}$ can be executed directly in global memory with 22 sequential steps (although this is known to be suboptimal), or using two stages in shared memory (denoted 2048:2048 or 2K:2K), or with three stages (e.g., 256:128:128). As many parameters can vary (radix size, thread usage, transposition strategy, etc.), all candidates are benchmarked as described in the next section.

Once the fastest configuration is selected, the Lucas-Lehmer test begins. Each iteration consists of the following seven steps:

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

For performance reasons, LucasNTT attempts to group as many steps as possible within each CUDA kernel. On the other hand, the FFT steps may involve sub-steps, increasing the number of kernels.

## 5 Explanation of the Source Code and Class Architecture

All classes responsible for executing the Lucas-Lehmer process inherit from the base class Process. Each Process instance may own from 0 to $n$ SubProcesses, which also inherit from Process. The lowest level of this hierarchy consists of CUDA kernels. Before selecting its SubProcesses, each Process evaluates the performance of available candidates. The fastest ones are selected using the GetBestPerformers() method.

There are four hierarchical levels in the Process class architecture, organized as follows:

$$\texttt{MainProcess} \longrightarrow \texttt{LucasPRP} \longrightarrow \texttt{KernelPool} \longrightarrow \texttt{Kernel}$$

Here, the arrows represent a parent-to-child *subprocess* relationship, not class inheritance (which is depicted in figure 1. Each parent Process instance owns one or more child SubProcesses, forming a recursive execution structure.

The entry point of the program is main.cpp, which instantiates exactly one instance of MainProcess. This class is responsible for analyzing the .ini file and creating as many LucasPRP candidates as there are FFT length configurations. In the previous example where $q = 82{,}589{,}933$, MainProcess can potentially instantiate a LucasPRP candidate in global memory for the full size $4{,}194{,}304$, another with the FFT configuration 2048:2048, and another with 256:128:128. These are the LucasPRP candidates for executing the Lucas-Lehmer test. The fastest candidate is selected among them.

The LucasPRP class is responsible for performing the main Lucas-Lehmer loop $x_{k+1} = x_k^2 - 2 \mod M_q$. It allocates memory buffers and precomputes the widths and weights used by the IBDWT. Depending on its FFT length, the LucasPRP class will instantiate different pools of kernels. For example, the 2048:2048 FFT configuration can be implemented by three different means: a pool of kernels processing the first FFT by columns without matrix transposition (FFT_2SMEM), another pool using matrix transpositions so that the first FFT is taken by rows with coalesced accesses (FFT_2SMEM_TRANSPOSE), or another pool storing the input number in a transposed array to avoid the initial transposition step (FFT_2SMEM_STORE_TRANSPOSED). In this case, the LucasPRP class contains three KernelPool candidates, which also inherit from Process. Again, the fastest candidate is selected among them.

Each KernelPool is responsible for identifying the appropriate CUDA kernels to use. A pool consists of different *families* of kernels. For instance, in the FFT_2SMEM configuration, the KernelPool is responsible for finding kernels of the following *families*:

- WEIGHT_FFTCOL_TWIDDLE
- FFTROW_SQUARE_IFFTROW
- TWIDDLE_IFFTCOL_UNWEIGHT
- REDUCE2BASE
- MINUS2

A Factory class is responsible for instantiating the appropriate Kernel candidates. The Kernel class, which wraps the actual CUDA kernel launch, lies at the bottom of the hierarchy.

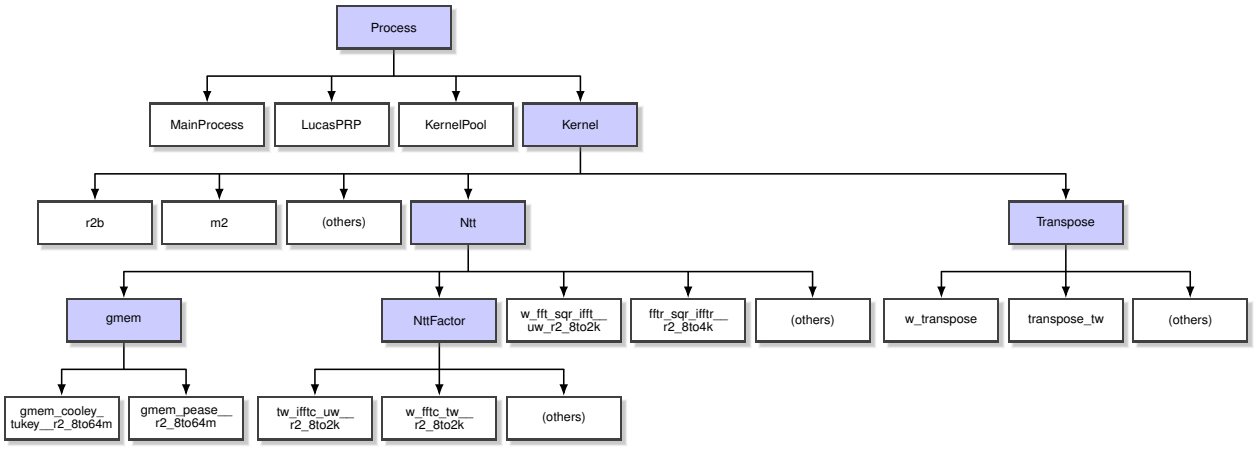


Figure 1: Class Inheritance from `Process` (partial diagram)

Most kernels inherit from the `Ntt` class and use precomputed roots of unity. For each kernel family, several implementation variants typically exist. For example, within the `FFTROW_SQUARE_IFFTROW` family, which performs in shared memory $n$ FFTs by rows, then a pointwise squaring, followed by $n$ backward FFTs, there are several options: one kernel might assign one Radix-2 butterfly per thread, another may assign multiple butterflies per thread, or use a Radix-4 implementation. Even at this level of the hierarchy, we have candidates. And again, the fastest candidate is selected among them. Note that most of the kernels are using C++ templates extensively to minimize runtime overhead. All kernel class names and filenames follow a naming convention that describes first their role, followed by a double underscore, followed by the methodology and the scope of work.

The `Factory` class is responsible for instantiating all possible kernel variants. The factory design pattern is particularly useful in this context, as this is the only class that needs to be aware of all kernel implementations. From a development standpoint, this is the only place where a new `#include` must be added when introducing a new kernel.

The `MainProcess` executes in two phases: first, it selects the fastest candidates using the `GetBestPerformers()` method, which recursively evaluates the performance of all `Process` instances; then, it launches the Lucas-Lehmer test using the `Run()` method, also executed recursively. These two recursive methods (among others) in the `Process` base class are of particular interest from a programming perspective.

In addition to the `Process`-derived classes, several helper classes assist with specialized tasks. Notably, the `NttLengths` class computes, analyzes, and displays possible FFT lengths. The `Logger` class handles output messages to the user and implements the singleton design pattern, another instructive feature from a software architecture perspective.

## 6 FFT programming

All `Ntt` kernels implement the Pease FFT algorithm ([9]). Like the well-known Cooley-Tukey ([10]) and Stockham ([11] [12]) algorithms, it is a Radix-based algorithm, but it is often simpler and faster to implement. This is due to its regular structure: the computation of butterflies does not vary across stages. In the Radix-2 version of the algorithm, the outer loop runs for $\log_2(n)$ iterations (as with Cooley-Tukey), while the inner loop processes each of the $n/2$ butterflies using a fixed stride of $n/2$ when loading values. Regardless of the iteration, the results are consistently written back to the output with no stride.

Here is how the `LucasNTT` kernels operate: load data from global memory into shared memory, perform the FFT computation entirely in shared memory, and then write the result back to global memory. In a straightforward Radix-2 Pease implementation, the shared-memory computation uses $n/2$ threads per block, which limits the FFT size to $2^{11}$ when using 1024 threads, the upper limit of Cuda[1]. The algorithm consists of an outer loop from 0 to $\log_2(n) - 1$ and an inner butterfly computation where each thread operates as follows: load two values from shared memory with a stride of $n/2$, synchronize threads to ensure all reads are complete, compute the butterfly, write the results back to shared memory with no stride, and then synchronize again to ensure all threads have written their outputs before proceeding to the next stage.

[1] `https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model`

Here is the pseudo-code of a Decimation-In-Frequency transform in a Cuda kernel:

```
for (int j = 0; j <= log_n - 1; j++) {
  u = x[threadIdx.x];
  v = x[threadIdx.x + n/2];
  __syncthreads();
  x[threadIdx.x * 2] = u + v;
  x[threadIdx.x * 2 + 1] = (u - v) * w[threadIdx.x >> j << j];
  __syncthreads();
}
```

*Note: For the sake of clarity, this pseudo-code has been slightly simplified (in real life arithmetic operations are performed modulo $p$).*

If we assign multiple butterflies to each thread, or if we increase the Radix, we reduce the number of threads, and we can increase the length of the FFT being processed in shared memory, up to the limit of the shared memory size. Thanks to thread synchronization, the Pease algorithm, traditionally considered an out-of-place algorithm, can be implemented in-place. So we do not need to allocate a double buffer. As a result, FFTs of lengths up to $2^{12} = 4096$ can be successfully processed in shared memory.

## 7 Future improvements

Here is a list of potential suggestions that could improve performances.

### 7.1 Small FFTs by batch

Large FFTs are commonly split into 2 or 3 smaller FFTs ([13]). Those small FFTs may potentially be accelerated. For example, when computing a $2^{22}$-point FFT with a 3-step decomposition such as $128 : 128 : 256$, we start by computing $32768 \times$ `FFT-128`. With a Radix-2 schema, this is done by launching a kernel with $32768$ blocks of $64$ threads. This is not optimal because the number of blocks that reside on the multiprocessor is limited by the number of registers available. So, it may be interesting to decrease the number of blocks, for example with $4096$ blocks of $512$ threads. Then each block would have to compute $8 \times$ `FFT-128` in a row. Here it would make sense to synchronize the threads by FFT, so instead of using a block synchronization with `__synchthreads()` we should use cooperative groups of $32$ threads, a warp synchronization.

Furthermore, on recent GPUs, we could use Multi-Stage Asynchronous Data Copies using `cuda::pipeline`, enabling the kernel to overlap memory transfers with computation. In other words, we could start computing the first small FFT once the first transfer has completed, and in parallel start loading in shared memory the elements of the second FFT, and so on. With this technique, the cost of memory transfers could be dramatically reduced, overlapped with FFT computation. Read the related NVIDIA blog[2] for additional information.

### 7.2 Higher Radix

The LucasNTT program currently does not fully exploits the potential of the Radix-4 and Radix-8 algorithms. The current implementation is limited to two classes, `fftr__r4_8to2k` and `fftr__r8_8to2k`, that can probably be improved. Indeed, the roots of unity are currently not stored in shared memory, so we make repeated accesses to the global memory. Secondly, only one butterfly is currently assigned per thread, meaning that we need $n/4$ threads to compute a Radix-4 FFT of length $n$. This is not optimal for a FFT-4096, used in the `FFT_2SMEM` schemes for Mersenne numbers with exponents between $83'886'080$ and $167'772'159$.

### 7.3 Using shifts

The multiplication modulo $p$ by the 64-th roots of unity can be performed by shifts instead of the `mul_Mod()` method. The code from Nick Craig-Wood ([7] has been adapted to C++/Cuda in `arithmetics.cu`, although `LucasNTT` does not currently make systematic use of this approach, apart from a limited implementation in the Radix-4 and Radix-8 kernels. Further investigation of this technique may prove beneficial.

[2] `https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/`

## 8 Conclusion

LucasNTT demonstrates how integer-based NTTs can be implemented efficiently on GPUs, with a focus on clarity, modularity, and performance. The class organization provides a clear separation of responsibilities, which strengthens maintainability and makes the code easier to extend. This structure turns the program into a flexible platform where new strategies, optimizations, and arithmetic techniques can be integrated without compromising readability. While the current framework is tailored to integer arithmetic, its modular nature suggests potential extensibility to the complex domain. Minimal structural changes would be required to the class hierarchy (kernel candidates within a kernel pool, orchestrated by the `LucasPRP` and `MainProcess` classes). Preliminary attempts by the author to generalize the arithmetic layer via an abstract `Element` class, supporting both integer and complex types through operator overloading, demonstrate conceptual feasibility. However, such polymorphism introduces significant performance overheads.

Consequently, a more pragmatic approach would be to fork the project for complex transforms, preserving the performance-critical optimizations of the integer-based implementation while enabling dedicated enhancements for complex arithmetic.

## References


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

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

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

[4] Peter L. Montgomery. Modular Multiplication Without Trial Division, 1985.

[5] Paul Barrett. *Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor*, page 311–323. Springer Berlin Heidelberg, 1987.

[6] Donald E. Knuth. The Art Of Computer Programming (volume 2) 2nd Edition, 1981.

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

[8] Alisah Ozcan, Arsalan Javeed, and Erkay Savas. High-Performance Number Theoretic Transform on GPU Through radix2-CT and 4-Step Algorithms. *IEEE Access*, 13:87862–87883, 2025.

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

[10] 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.

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

[12] 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.

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