Quantum Algorithms for Exponentiation in a Group

Xavier Bonnetain
Université de Lorraine, CNRS, Inria
Nancy, France

Pierrick Gaudry
Université de Lorraine, CNRS, Inria
Nancy, France

Medhi Kermaoui
Université de Lorraine, CNRS, Inria
Nancy, France

***Abstract*—Modular exponentiation, and more generally, exponentiation of a group element, is an elementary operation that is ubiquitous, and its efficiency is critical for many applications in cryptography and computer algebra.**

In quantum algorithms, the case where the group element is fixed and the exponent is in superposition is well-studied, because it is part of Shor's factoring and discrete logarithm algorithms. The dual case where the exponent is fixed and the group element is in superposition is far less studied, but is also important, as it appears, for instance, in Kuperberg's hidden subgroup algorithm for the cryptanalysis of isogeny-based cryptography.

In this work, we propose a generic quantum circuit for the exponentiation in this context. It builds upon reversible square-and-multiply (which is fast, but uses a number of qubits that is linear in the length of the exponent) and the Fibonacci addition chain (which is slower, but uses constant space). Our circuit is parametrizable and provides relevant time-memory trade-offs. In particular, it is well suited in cases where the number of available qubits is moderate, which is a typical situation when the exponentiation is part of a larger quantum computation.

While our approach is generic, we focus on two use-cases with distinct cost profiles: modular exponentiation and scalar multiplication of a point on an elliptic curve.

***Index Terms*—modular exponentiation, quantum algorithms, Fibonacci decomposition.**

I. Introduction

The algorithmic toolbox for quantum computers is still in its infancy compared to what has been developed for classical processors for decades. Basic arithmetic circuits, as, for instance, for modular arithmetic or elliptic curves, benefit from a high development [17], [23], [24], due to their practical impact in Shor's algorithm [22] for discrete logarithm and integer factorization. In this algorithm, some form of modular exponentiation (or scalar multiplication, in the case of elliptic curve discrete logarithm) is required, before running a quantum Fourier transform: the group element is a classical value, while the exponent is quantum. In this setting, the well-known square-and-multiply approach gives the expected complexity in time and space.

For other quantum algorithms, for instance Regev's factoring algorithm [20] or Kuperberg's hidden subgroup algorithm [12], the group elements to exponentiate are no longer classical. Kuperberg's algorithm, or variants of it, provides the best known attack on cryptosystems based on an abelian group action, such as the famous isogeny-based system CSIDH [4]–[6]. A naive translation of the classical binary exponentiation algorithms leads to a space complexity that grows linearly with the length of the computation. While it is currently hard to tell with certainty what will be the practical constraints on a potential future large-scale quantum computer, it is reasonable to bet that the number of available qubits (i.e. the space complexity) will be at least as important as the number of operations to apply on them. In this work we focus on this question of exponentiation in a group, where the group element is quantum (i.e. a superposition of potentially exponentially many classical group elements), while the exponent is classical.

We set ourselves in a computation model where we do not allow any measurement during the exponentiation. Furthermore, we take a generic group approach, where we assume that we are given quantum circuits for the group operations, and we build our exponentiation circuit on top of them. Therefore, in the end, the circuits we design are just classical reversible circuits, so that when we feed them with qubits instead of bits, they become quantum circuits. We also stick to the notion of logical qubits; a concrete implementation would require an implementation-dependent layer of quantum error correction. We choose a multiplicative notation for the group (even though elliptic curves consist of one of our main applications).

The principal cause of trouble with respect to reversibility is the square operation in the group. In many groups, the known (classical) algorithms for computing a square root are much slower than the ones for multiplication. (A notable exception are elliptic curves in characteristic 2, where point-halving can be very efficient [11].) As in-place reversible squaring would be equivalent to in-place reversible square root, we do not expect to have such an efficient circuit, and rely on an out-of-place circuit that computes x^2 in a new register and copies x. A chain of square-of-multiply that involves n squares therefore requires n temporary registers[1].

For the multiplication, the situation is different in the case where the inversion is cheap (as in the case of elliptic curves), or in the case where we have some precomputed inverses. It is then possible to design a reversible and in-place multiplication circuit. This opens the door to Fibonacci addition chains *à la* Meloni [14], for which the quantum space complexity remains constant in the length of the exponentiation [19].

The exponentiation is never the final application, but just an arithmetic building block in a larger circuit. In this context, while spending n temporary registers of space for this primi-

This work has been partially funded by the France 2030 grants ANR-22-PETQ-0007 EPiQ and ANR-22-PETQ-0008 PQ-TLS.

[1]Recall that to reset a register reversibly, we would need to *uncompute* it, that is, apply the operation that produced the value in the register in reverse.

tive might often be prohibitive, other parts of the circuits will frequently require at least a few additional temporary registers that could be used during the exponentiation. This raises the question of a time-space trade-off: when, say, a dozen of temporary registers are available, binary exponentiation (the fastest) is not feasible due to space constraints, and the Fibonacci addition chain is slower but does not make use of all the available space. We therefore explore the interpolation between both situations.

Our main contribution is a general quantum circuit that combines many techniques: small chains of square and multiply, Fibonacci sequences, window techniques, and the Bennett trade-off strategy. This circuit depends on several parameters that allow to control the number of temporary qubits that are used. It naturally relies on a decomposition of the exponent in multiple bases, for which we use a greedy algorithm. The analysis shows that it provides a very good answer to our interpolation question.

Our analysis is generic enough to handle the two important cases, which are the multiplicative group of a finite field, and the group of points on an elliptic curve. The former is more costly than the latter, because the inversion is expensive.

The article is organized as follows. First, in Section II, we briefly recall the basics of quantum circuits, and list the set of subcircuits for group operations that we take as building blocks. In Section III we give the known algorithms, that are then combined in Section IV into our general circuit. The associated algorithm for the decomposition of the exponent is described in Section V. Finally experiments and analysis are done in Section VI.

II. Context and Background

A. Quantum Circuits

We present in this section a brief introduction to quantum computing, and refer to [16] and [10] for a detailed presentation of the topic. A qubit is a unitary vector in $\mathbb{C}^2$, it can be written as $|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ with $|\alpha_0|^2 + |\alpha_1|^2 = 1$ and where $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$ is the computational basis. A measurement on $|\psi\rangle$ makes it collapse into $|0\rangle$ with probability $|\alpha_0|^2$ and into $|1\rangle$ with probability $|\alpha_1|^2$.

More generally, a register of n qubits is a unitary vector in $(\mathbb{C}^2)^{\otimes n}$. It can be written as $|\psi\rangle = \sum_{j=0}^{2^n-1} \alpha_j |j\rangle$ with $|j\rangle := |j_{n-1}\rangle \otimes \cdots \otimes |j_0\rangle$ where $j = j_{n-1} \dots j_0$ is the binary decomposition of j. The orthonormal basis $|j\rangle_{0 \le j \le 2^n-1}$ is called the computational basis, and we say that $|\psi\rangle$ is in *superposition* of this computational basis. As in the one dimensional case, measuring $|\psi\rangle$ makes it collapse to $|j\rangle$ with probability $|\alpha_j|^2$. A quantum algorithm is described as a quantum circuit, represented as a sequence of gates, each being a unitary operator, acting on some of the qubits, along with measurements (in our setting, these are only at the end).

One major difference with classical computing is the fact that, excluding measurements, quantum computing is reversible. For any unitary gate U we denote by $U^\dagger$ its inverse, which is the uncomputation of U.

As for boolean circuits, the cost metrics used to describe the complexity of a quantum circuit are its *size* (the number of gates), *width* (the number of registers) and *depth* (length of the longest path from an input gate to an output gate i.e the critical path of the circuit).

We assume that we are given a universal set of gates (for example Clifford+T) which are reversible and for which the inverse gates also count for 1. In this setting, the cost in depth, width and size will be the same for U and $U^\dagger$.

In our work, we will concentrate on the complexity in size and width. We assume our set of gates contains basic building blocks such as $CNOT : |x\rangle |y\rangle \mapsto |x\rangle |x \oplus y\rangle$ (or that it is cheap to implement with our gates), and that it is negligible compared to the cost of group operation circuits.

B. Abelian Groups

Let $(G, \times)$ be a finite abelian group with 1_G its neutral element. We assume that any element x of G has a bit-string representation $x = (x_{n-1}, \dots, x_0)$ of length n. We define the *group register* associated to x as the register of n qubits $|x\rangle := |x_{n-1}, \dots, x_0\rangle$. For simplicity we will denote by $|0\rangle$ the all-0 n-qubit register $|0\rangle^{\otimes n}$. It doesn't have to represent an element of the group.

We want to build a circuit $C : |x\rangle |0\rangle^{\otimes K} \mapsto |\star\rangle \left|x^N\right\rangle$ with N a classical integer and $|\star\rangle$ working registers used for the computation, called *ancillaries*.

Should one want to clear the $|\star\rangle$ registers to reuse them for further computations, it suffices to copy $\left|x^N\right\rangle$ in an additional register and then uncompute C. That is, the operation is $|\star\rangle \left|x^N\right\rangle |0\rangle \overset{CNOT}{\mapsto} |\star\rangle \left|x^N\right\rangle \left|x^N\right\rangle \overset{C^\dagger}{\mapsto} |x\rangle |0\rangle^{\otimes K} \left|x^N\right\rangle$.

Hence $\bar{C} : |x\rangle |0\rangle^{\otimes K} |0\rangle \mapsto |x\rangle |0\rangle^{\otimes K} \left|x^N\right\rangle$ can be easily built from C, at twice its cost.

To compute this exponentiation, we assume having access to the gates:

- $S : |x, 0\rangle \mapsto \left|x, x^2\right\rangle$
- $M : |x, y, 0\rangle \mapsto |x, y, xy\rangle$
- $I : |x\rangle \mapsto \left|x^{-1}\right\rangle$

and if $(G, \times) \subset (A, +, \times)$ with A a commutative ring, also access to the addition, and to a fused multiply-add $FMA : |x, y, z\rangle \mapsto |x, y, z + xy\rangle$.

Remark 1. 1) *As explained in introduction, we can not expect to have an in-place gate $|x\rangle \mapsto \left|x^2\right\rangle$, because this would imply an efficient square-root.*

2) *The cost of the inversion gate depends on the context. While for elliptic curve, inverting is almost for free, it is a costly operation for finite fields, which can be achieved using extended Euclidean algorithm based methods [21] or Fermat's little theorem based methods [13].*
In the case where I is expensive, when exponentiating $|x\rangle$, we assume that $\left|x^{-1}\right\rangle$ is also given as input, or that we compute it at the very beginning, but then we don't use I anymore in the exponentiation algorithm.

From these building blocks, we define $M_r : \left|x, x^{-1}, y, 0\right\rangle \overset{M}{\mapsto} \left|x, x^{-1}, y, xy\right\rangle \overset{M^\dagger}{\mapsto} \left|x, x^{-1}, 0, xy\right\rangle$ and

an in-place multiplication which we call the Fibonacci gate $F : |\widehat{x}, \widehat{y}, 0\rangle \mapsto |\widehat{x}, \widehat{xy}, 0\rangle$ where $|\widehat{x}\rangle := |x\rangle \left|x^{-1}\right\rangle$ is called the *augmented register* associated to x.

Depending on the context, F can be implemented using:

- 2 calls to M_r in the generic context. To get this, we apply M_r on the registers $|x\rangle \left|x^{-1}\right\rangle |y\rangle |0\rangle$ to get $|x\rangle \left|x^{-1}\right\rangle |xy\rangle |0\rangle$ and then apply M_r on $\left|x^{-1}\right\rangle |x\rangle \left|y^{-1}\right\rangle |0\rangle$.
- 3 calls to FMA when $(G, \times) \subset (A, +, \times)$ [18, Lemma 3.7],
- 1 call to M if G is an elliptic curve [7]. In that case, we also have $M_r = M$.

For the Fibonacci F-gate, and more generally for all other gates representing group operations, we will sometimes change the order of the input or output registers. This corresponds to a re-organization of the wires between the gates of the circuit, which is free. For simple cases, this will not cause any ambiguity. In the next section, we will give a graphical representation of the circuit, where we add a visual indication for the roles of the output wires.

Proposition 1. *The Fibonacci gate has the following properties:*

- $\left|\widehat{x^{f_{i+1}}}\right\rangle \left|\widehat{x^{f_i}}\right\rangle |0\rangle \overset{F}{\mapsto} \left|\widehat{x^{f_{i+2}}}\right\rangle \left|\widehat{x^{f_{i+1}}}\right\rangle |0\rangle$,
- $|\widehat{x}\rangle \left|\widehat{1_G}\right\rangle |0\rangle \overset{F^i}{\mapsto} \left|\widehat{x^{f_{i+1}}}\right\rangle \left|\widehat{x^{f_i}}\right\rangle |0\rangle$,

where f_i denote the Fibonacci sequence defined by $f_0 = 0, f_1 = 1$ and $f_{i+2} = f_{i+1} + f_i$.

In our circuits, we will sometimes need to apply the square operator to a group element in augmented representation. We denote the corresponding out-of-place operator $\hat{S}$, which is just an application of S on both $|x\rangle$ and $\left|x^{-1}\right\rangle$.

From now on, we will often skip mentioning the ancilla registers used by in-place group operators, since they are restored after the operation, and can be used by subsequent operators. Their contribution in the total width of the circuit is therefore a small additive constant.

III. Basic Techniques

A natural way to implement quantum exponentiation is by adapting classical algorithms to the quantum context.

Square-and-multiply (SM). Let N be an integer and $(a_{r-1}, \ldots, a_0)_2$ be its binary expansion and h its Hamming weight. Based on the Square-and-multiply algorithm, we can build a quantum circuit $|x\rangle \underbrace{|0\rangle^{\otimes r-1}}_{\text{squares}} \underbrace{|0\rangle^{\otimes h-1}}_{\text{multiplies}} \mapsto \underbrace{|\star\rangle}_{\text{workspace}} \left|x^N\right\rangle$. This circuit uses $r-1$ iterations of S and $h-1$ iterations of M, leading to a need of $r-2+h$ ancilla registers.

Window-versions. Time-space tradeoff variants of the SM method exist, which can also be adapted to the quantum context. A generic classical algorithm is given by Algorithm 1.

Let T be a finite set of small integers, we call a $T-$decomposition of N a sequence of $n_i \in T$ where $N = \sum_{i=0}^{r-1} n_i 2^i$. We represent this decomposition by $N = (n_{r-1}, \ldots, n_0)_T$ and denote by h_T its Hamming weight. When the size of T grows, this Hamming weight decreases. Excluding the precomputation of the table, the corresponding quantum circuit calls the S gate $r-1$ times and the M gate $h_T - 1$ times, using $r - 2 + h_T$ ancillas.

Algorithm 1 Window variants of SM

Input: $\{x^i, i \in T\}$ a table of precomputed exponents of x and a $T-$decomposition of $N = (n_{r-1}, \ldots, n_0)_T$
Output: The element $x^N \in G$.
$y \leftarrow 1$
for $i = (r-1) \cdots 0$ **do**
  $y \leftarrow y^2 \times x^{n_i}$ {// *skip mult. if* $n_i = 0$}
end for
return y

Fibonacci. The methods described previously use a linear number of registers in the length of the exponent N, which is a practical issue considering the technical challenge to build logical qubits.

Based on the works of Meloni [14] and Kaliski [9], S. Ragavan and V. Vaikuntanathan proposed a space-efficient quantum exponentiation [19]. The first step, formalized in Proposition 1, is to observe that iterating F multiple times on $|\widehat{x}\rangle \left|\widehat{1_G}\right\rangle$ makes the Fibonacci sequence appear in the exponents. If we combine this to the fact that any integer can be decomposed as a sum of Fibonacci numbers (see Theorem 1), we get Algorithm 2.

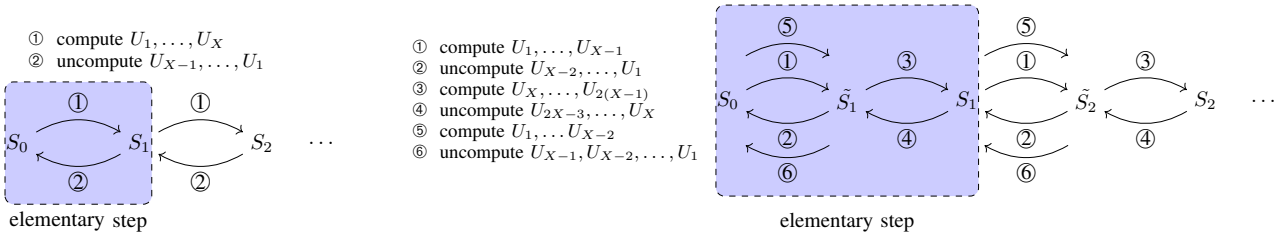
Theorem 1 (Zeckendorf representation)**.** *Let N be a positive integer and $(f_i)_{i \geq 0}$ the Fibonacci sequence, then N can be uniquely written as $N = \sum_{i=2}^{k} c_i f_i$ with $c_i \in \{0, 1\}$ and $c_{i+1} c_i = 0$. We write by $N = (c_k, \ldots, c_2)_Z$ this representation.*

Algorithm 2 Fibonacci exponentiation

Input: The state $|\widehat{x}\rangle \left|\widehat{A}\right\rangle \left|\widehat{B}\right\rangle$, where $\left|\widehat{A}\right\rangle$ and $\left|\widehat{B}\right\rangle$ are working registers initialized at $|\widehat{x}\rangle$, and the Zeckendorf representation of $N = (c_k, \ldots, c_2)_Z$
Output: The state $|\widehat{x}\rangle \left|\widehat{x^N}\right\rangle \left|\widehat{x^{N_-}}\right\rangle$ where $N_- = \sum_{i=3}^{k} c_i f_{i-1}$
for $i = (k-1) \cdots 2$ **do**
  if $c_i = 1$ **then**
    apply F on $|\widehat{x}\rangle \left|\widehat{A}\right\rangle$
  end if
  apply F on $\left|\widehat{A}\right\rangle \left|\widehat{B}\right\rangle$
  swap $\left|\widehat{A}\right\rangle$ and $\left|\widehat{B}\right\rangle$
end for

Computing $\left|\widehat{x^N}\right\rangle$ using Algorithm 2 requires three augmented registers and one ancilla (for applying the F-gate), and calls the F-gate $k - 3 + h_Z$ times, where h_Z denote the Hamming weight of $(c_k, \ldots, c_2)_Z$.

Algorithm 2 corresponds to the circuit below. In this circuit, the red gates are removed when the corresponding c_i is zero.

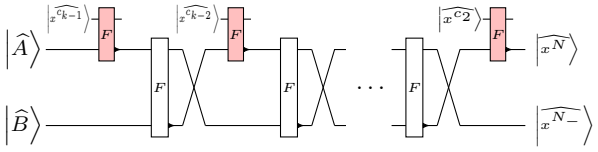


(a) Level-1 time-space tradeoff.

(b) Level-2 time-space tradeoff.

Fig. 1. Two non-asymptotic variants of Bennett's time-space tradeoffs. In both cases, the elementary steps, represented in blue, consume one register and are repeated until the memory is filled.

Otherwise $c_i = 1$, and $\left|\widehat{x^{c_i}}\right\rangle$ is just $|\widehat{x}\rangle$. Also, the small black triangle in the output wire of an F-gate tells in which wire the product is sent.



Remark 2. *A variant of this decomposition introduced by Alpert [2], called the far difference representation of N, allows $c_i = -1$. It is usable in Algorithm 2, as $\left|\widehat{x^{-1}}\right\rangle$ is simply $|\widehat{x}\rangle$ if we swap the order of its two registers.*

Bennett uncomputation. Bennett [3] proposed a generic framework to reduce the space required by a circuit, at the expense of more computations. This includes asymptotic considerations that are mostly of theoretical interest, but the general idea can be applied to practical circuits.

Let $U_1, U_2, \ldots,$ be a sequence of out-of-place gates that constitute a circuit, where each of the U_i consumes one ancilla register. If we have only access to X ancillas, then we can apply the X out-of-place gates, up to U_X before filling-up the memory.

To go further, it is necessary to uncompute part of the computations, to free some ancillas. The idea will be to do X steps, then to keep the state S_X reached after U_X in an ancilla register and apply $U_{X-1}^\dagger, \ldots, U_1^\dagger$ to restore the other $X-1$ ancillas. It allows then to continue the computation with $X-1$ free ancillas. This approach therefore allows to perform up to $\frac{X^2+X}{2}$ out-of-place operations with only X ancillas, at a cost of roughly doubling the number of gates.

We refer to this as a level-1 Bennett tradeoff. It can be summarized by the state diagram in Figure 1(a), where the S_i in a node is the state reached after applying a sequence of U_j's, such that between S_{i+1} and S_i, we consume one ancilla.

A more aggressive tradeoff is represented by the state diagram in Figure 1(b). This level-2 tradeoff is obtained by iterating the following elementary step, which is using X ancillas. It consumes one of them, and performs $2(X-1)$ out-of-place operations.

Adding steps ⑤ and ⑥ to the first tradeoff almost doubles the number of operations we can perform before filling the memory, allowing now $X^2 - X$ operations for X ancillas. However these operations need to be performed approximately three times due to the uncomputations.

The same pattern of computation / uncomputation can be applied recursively, leading to an asymptotic situation where a circuit that is a chain of n out-of-place operations can be transformed in a circuit that uses roughly $\log n$ ancillaries, but a cost in number of gates that becomes exponential in n, due to multiple uncomputations. In our case where we target practical circuits, we will stick to level-1 uncomputations.

IV. A General Quantum Exponentiation Circuit

We propose a general quantum circuit that allows to combine all the basic techniques of the previous section.

A. Setting

We recall that our target is to have a quantum circuit $|x\rangle\,|0\rangle^{\otimes K} \mapsto \left|x^N\right\rangle|\star\rangle$, built on top of the S, M, I, F gates, that takes a superposition of elements of G and returns its N-th power. Here N is a fixed classical integer, known at compile-time, i.e. the circuit is specific to the value of N. As explained in Section II, if need be, it can be turned into an in-place circuit that returns clean ancillas by uncomputing the whole circuit, thus paying a factor of 2.

Everywhere in the circuit, the registers will contain either $|0\rangle$ or a power $\left|x^k\right\rangle$ of the input. We will therefore simplify the notation by using $|k\rangle$ for $\left|x^k\right\rangle$. This leads to a small ambiguity for $|0\rangle$ that can be either $\left|x^0\right\rangle$, that is the neutral element of G, or the all-zero qubit register. The context easily allows to decide between both. With this notation the linear nature of the S, M, I, F gates becomes apparent (they are linear in the discrete logarithm in base $|x\rangle$). Similarly, we let $\left|\widehat{k}\right\rangle$ denote the augmented register $\left|\widehat{x^k}\right\rangle$.

B. The SMF_c Sub-circuit

The starting point is the Fibonacci approach, which is the best choice when the number of available registers is very small. It works with augmented registers. If a few more registers are available, we can use them in various ways. First, a table of multiples $\left|\widehat{i}\right\rangle$ can be precomputed for small integers

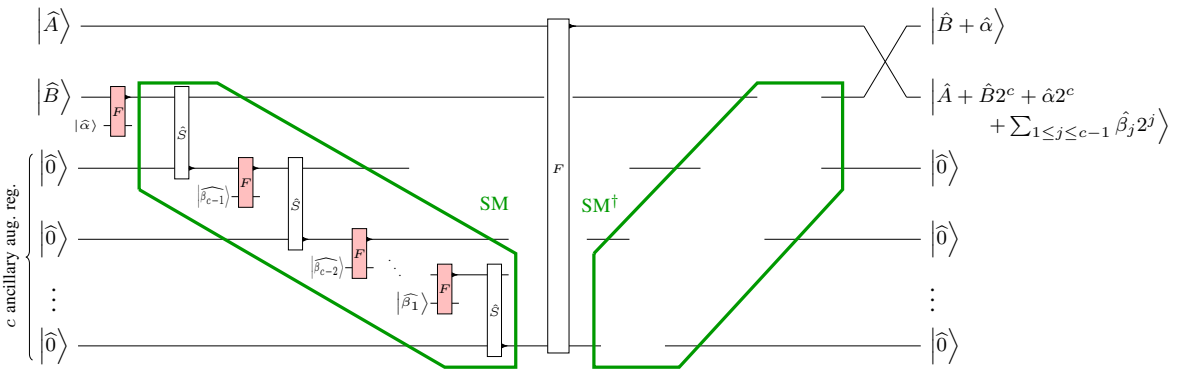


Fig. 2. SMF_c circuit. The extra inputs of the red F-gates are taken from a precomputed table. If the element is zero, the corresponding F-gate (and its uncomputation, if part of the SM block) is skipped.

i at the beginning of the circuit to allow window approaches. Second, between the F-gates of the Fibonacci sequence, it is possible to insert a small square-and-multiply chain, on the largest register, also with augmented registers. This small chain has to be uncomputed after the F-gate, to allow the use of the spare ancillas again. Third, at the end of the Fibonacci sequence, it is possible to finish with a possibly longer square-and-multiply sequence. This sequence, being the end of the computation, does not need to be uncomputed, and therefore can work on non-augmented registers. It can still be combined with a Bennett-like approach.

We therefore define a basic circuit element that we call SMF_c, which works in place on $c+2$ augmented registers, c of them being ancillary that must be zero and will be returned zero, and 2 of them being the working registers, that we call $\left|\widehat{A}\right\rangle$ and $\left|\widehat{B}\right\rangle$. It assumes that a table T of small multiples has been precomputed. This SMF_c circuit, shown on Figure 2, does the following:

1) Add in place an element $|\widehat{\alpha}\rangle \in T$ to $\left|\widehat{B}\right\rangle$.
2) Do a square-and-multiply sequence of length c, starting with $\left|\widehat{B}\right\rangle$, inserting elements $\left|\widehat{\beta_j}\right\rangle$ for $j = c-1, \ldots, 1$, and using the c ancillaries.
3) Add in place the result of the SM sequence to $\left|\widehat{A}\right\rangle$.
4) Uncompute the SM sequence, to restore the ancillaries.
5) Swap the $\left|\widehat{A}\right\rangle$ and $\left|\widehat{B}\right\rangle$ registers.

There are other places where we could insert element from the table: at the beginning of the circuit, we could add an element $\left|\widehat{\beta_0}\right\rangle$ to the register $\left|\widehat{A}\right\rangle$. This is however functionally equivalent to adding the same value to $\left|\widehat{B}\right\rangle$ at the very end of the circuit. Since the SMF_c circuits are going to be chained, this does not bring anything: adding $\left|\widehat{\beta_0}\right\rangle$ to $\left|\widehat{A}\right\rangle$ can be replaced by doing this addition to $\left|\widehat{B}\right\rangle$ at the beginning of the next occurrence of SMF_c. Furthermore, within the SMF_c circuit, we could insert a value after the last S of the SM sequence. It has however the same effect as inserting the same value in $\left|\widehat{A}\right\rangle$, which is useless.

The SMF_c circuit, parametrized by c small integers $(\alpha, (\beta_j)_{1\le j\le c-1})$, computes

$$\left|\widehat{A}\right\rangle\left|\widehat{B}\right\rangle \mapsto \left|\hat{B}+\hat{\alpha}\right\rangle\left|\hat{A}+\hat{B}2^c+\hat{\alpha}2^c+\sum_{1\le j\le c-1}\hat{\beta}_j 2^j\right\rangle.$$

C. Combining Several SMF_c into a Complete Circuit

Our general circuit $\mathsf{GenCirc}_{c,w,n,d}$ proceeds as follows:

1) Compute the augmented representation of the input.
2) Compute a table T of all small multiples up to w in augmented representation, and initialize $\left|\widehat{A}\right\rangle$ and $\left|\widehat{B}\right\rangle$ to the neutral group element.
3) Apply n times the SMF_c circuit.
4) Apply an SM sequence of length d to $\left|\widehat{B}\right\rangle$, with a level-1 Bennett trade-off.

In order to analyze the operation computed by GenCirc, we first assume that the final SM sequence has length c, i.e. $d = c$. We let α_i and $\beta_{i,j}$, for $1 \le i \le n$, and $1 \le j < c$, denote the parameters (all belonging to T) of the $(n+1-i)$-th SMF_c circuit. The circuit is linear in all the α_i and $\beta_{i,j}$, and in the values of the input registers $\left|\widehat{A}\right\rangle$ and $\left|\widehat{B}\right\rangle$. We start by studying the computation done when all the parameters are zero (we call $\mathsf{GenCirc}^0$ the corresponding circuit, formed of SMF_c^0 sub-circuits), and one of the two input registers is $\left|\widehat{1}\right\rangle$.

Let G be the 2×2 matrix $G = \begin{pmatrix} 0 & 1 \\ 1 & 2^c \end{pmatrix}$, and let us define the 2^c-Fibonacci sequence $(\bar{g}_i)_{i\ge 0}$ by $\begin{pmatrix} \bar{g}_i \\ \bar{g}_{i+1} \end{pmatrix} = G^i \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. (Note that a similar sequence was used by Ragavan in [18].) By construction, a sequence of n SMF_c^0 circuits, with $\left|\widehat{0}\right\rangle$, $\left|\widehat{1}\right\rangle$ as inputs produces the values $|\widehat{\bar{g}_n}\rangle$ and $\left|\widehat{\bar{g}_{n+1}}\right\rangle$. Therefore, $\mathsf{GenCirc}^0$ gives $\left|\widehat{2^c\bar{g}_{n+1}}\right\rangle$. In order to align the notation and avoid a shift of indices, with define the sequence

i	-1	0	1	2	3	4
g_i	0	1	8	65	528	4289
h_i	0	1	8	64	520	4224
$\tilde{h}_{i,j}$	0	0: 1 1: 2 2: 4	0: 8 1: 16 2: 32	0: 64 1: 128 2: 256	0: 520 1: 1040 2: 2080	0: 4224 1: 8448 2: 16896

Fig. 3. First values of g_i, h_i and $\tilde{h}_{i,j}$ for $c = 3$.

$g_i = \bar{g}_{i+1}$, which starts with $g_{-1} = 0$ and $g_0 = 1$. Therefore the circuit computes $\left|\widehat{2^c g_n}\right\rangle$.

We also define the sequence $(\bar{h}_i)_{i\geq 0}$ by $\bar{h}_0 = 0$, $\bar{h}_1 = 1$ and for $i \geq 2$, $\begin{pmatrix} \bar{h}_i \\ \bar{h}_{i+1} \end{pmatrix} = 2^c G^i \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 2^c G^{i-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. By construction, the circuit $\mathsf{GenCirc}^0$ with $\left|\widehat{1}\right\rangle$ and $\left|\widehat{0}\right\rangle$ as inputs produces the value $\left|\widehat{\bar{h}_{n+1}}\right\rangle$. Re-indexing with $h_i = \bar{h}_{i+1}$, the circuit computes $\left|\widehat{h_n}\right\rangle$.

Let us consider the circuit $\mathsf{GenCirc}^0_{c,w,k,c}$, for a value $0 < k < n$, which is a sub-circuit of $\mathsf{GenCirc}^0_{c,w,n,c}$. The computation made by this sub-circuit on input $\left|\widehat{0}\right\rangle, \left|\widehat{B}\right\rangle$ is the same as the computation made, on input $\left|\widehat{0}\right\rangle, \left|\widehat{0}\right\rangle$ by the full circuit $\mathsf{GenCirc}^0_{c,w,n,c}$, where $\left|\widehat{B}\right\rangle$ is inserted at the beginning of the $(n-k+1)$-th SMF_c block, i.e. the parameter α_k is set to $\left|\widehat{B}\right\rangle$, and the others are kept to zero. Therefore, by linearity, we obtain that, on input $\left|\widehat{0}\right\rangle, \left|\widehat{0}\right\rangle$, the circuit $\mathsf{GenCirc}_{c,w,n,c}$, with $\beta_{i,j} = 0$ (i.e. we insert elements from T only at the beginning of each SMF_c block), computes $\sum_{1\leq i\leq n} 2^c \alpha_i g_i$.

In order to handle the insertions within the SM sequences of the SMF_c blocks, we introduce the sequence $\tilde{h}_{i,j}$ defined by $\tilde{h}_{i,j} = 2^j h_i$, for $j \in [0, c-1]$. Inserting the $\beta_{i,j}$ terms in the $(n+1-i)$-th SMF_c block of GenCirc is functionally equivalent to inserting the value $2\beta_{i,1}+4\beta_{i,2}+\cdots+2^{c-1}\beta_{i,c-1}$ to the last ancillary register, just before the F-gate, which is again equivalent to inserting this value in the working register $\left|\widehat{A}\right\rangle$ before the F-gate or in the $\left|\widehat{B}\right\rangle$ register at the beginning of the next SMF_c block. Therefore, the $\mathsf{GenCirc}_{c,w,n,c}$ circuit computes the value

$$\sum_{i=1}^{n} \left(2^c \alpha_i g_i + \sum_{j=1}^{c-1} \beta_{i,j} \tilde{h}_{i,j}\right).$$

In this formula, it must be remembered that the insertion of α_i's cost less than the $\beta_{i,j}$'s, since the corresponding F-gate does not need to be uncomputed.

It remains to insert non-zero values during the last SM sequence. This adds a $\sum_{0\leq i\leq c} \gamma_i 2^i$ term to the computed value, where γ_i is in T. If the circuit has a longer SM sequence at the end, then this formula must be seen as a prefix in a classical left-to-right algorithm. Putting everything together, the value computed by the $\mathsf{GenCirc}_{c,w,n,d}$ circuit is

$$N = 2^{d-c} \sum_{i=1}^{n} \left(\alpha_i 2^c g_i + \sum_{j=1}^{c-1} \beta_{i,j} \tilde{h}_{i,j}\right) + \sum_{i=0}^{d} \gamma_i 2^i \quad (1)$$

Algorithm	# registers	# gates
SM	$1.5n\,R$	$nS + 0.5nM$
SM-wNAF	$(1 + O(\frac{1}{\log n}))n\,R$	$nS + O(\frac{n}{\log n})M$
SM-Bennett	$\sqrt{n}\,R$	$2nS + o(n)M$
Fibonacci	$3\,\hat{R} + 1\,R$	$1.73nF$
Ours (no window)	$(c+3)\,\hat{R} + 1\,R$	$2n\hat{S} + \frac{2n}{3}(1 + \frac{1}{c})F$
(window)	$k\,\hat{R}$	$2n\hat{S} + \frac{2n}{\log k}F$

Fig. 4. Summary of asymptotic costs for an exponentiation by a number N of n bits. The "hat" notation corresponds to augmented registers, or operations that deal with them. In the case where inversions are non-free, they are typically doubled compared to non-hat version.

The validity of this formula has been experimentally checked with a basic (classical) implementation of the circuit that can be found in the supplementary material [1].

The last SM sequence can be done with non-augmented registers. We assume that d is large enough compared to c (but still $d \leq 2c^2 + c$) so that we use a level-1 time-space tradeoff. Without taking into account the cost of precomputing the table T (which depends on its form), the circuit makes $2nc$ calls to $\hat{S}$, $2d$ calls to S, $(\mathsf{h}_\alpha + 2\mathsf{h}_\beta)$ calls to F, and $2\mathsf{h}_\gamma$ calls to M_r, where we have denoted by h_α (resp. h_β and h_γ) the number of non-zero α_i (resp. non-zero $\beta_{i,j}$ and γ_i).

V. Decompositions of the Exponent

We assume we are given a c, and a window size w. To implement our circuit we need to decompose the exponent N in a sum whose general form is as in Equation (1). As we're doing windowed exponentiation, we have that $\max(|\alpha_i|, |\beta_{i,j}|, |\gamma_i|) \leq w$. This decomposition, to reduce the cost, should have a limited number of non-zero terms. We also force the non-zero α_i, $\beta_{i,j}$, γ_i to be odd numbers, in order to avoid overlap between the terms that come from $\tilde{h}_{i,j}$ for different j and between the terms that come from the 2^i.

Moreover, we are using different recurrent sequences, and the cost in the circuit depends on the sequence, namely:

- Insertion for $\alpha_i g_i 2^c$ costs one additional F gate;
- Insertion for $\beta_{i,j}\tilde{h}_{i,j}$ costs two additional F gates;
- Insertion for $\gamma_i 2^i$ costs two additional M_r gate, as we don't need the inverses here.

Thus, the decomposition should take into account that, depending on the context, some terms are more expensive than others. We decided to use a greedy algorithm that optimizes the "cost per bit gained" to choose the next term in the decomposition. Namely, for a candidate term X that has an associated cost C, we compute $\frac{1}{C}\log\left(\frac{N}{|N-X|}\right)$, with an infinite value if $N = X$. We choose the term that maximizes this value, and recursively apply the same process to $N - X$, until we reach 0. We have implemented this algorithm in Python, and the code is provided as supplementary material [1].

VI. Cost analysis

A. Asymptotic considerations

We consider the asymptotic behaviour of exponentiation of numbers on n bits. This is of course an abstract view, as in a

finite group it is pointless to compute a power larger than the group order, but it allows to give a simplified perspective of what is permitted by our algorithms.

1) Fibonacci exponentiation: This reference point uses 3 augmented registers and 1 register (the ancillary used by F). The Fibonacci sequence grows at a rate $\phi = \frac{1+\sqrt{5}}{2}$, thus the circuit will have an exponentiation sequence of length $\log_\phi(2)n \simeq 1.44n$. We need to add the insertions, and we know the average density of a far-difference decomposition is 0.2 [15]. Thus the sequence will use around $1.73n$ F-gates.

2) Square-and-multiply: Direct square-and-multiply uses n squares, and with inverses the density of a NAF decomposition is $1/3$. It can be made as small as $1/\log(n)$ using a window of size $n/\log(n)$. Thus this approach can cost n S-gates and $\mathcal{O}(n/\log(n))$ M-gates, at the expense of using $n+n/\log(n)$ registers.

3) Square-and-multiply tradeoffs: The memory can be reduced using uncomputations, but there is a hard limit of $\log(n)$ registers (at which point the algorithm uses 2^n squares overall, which is roughly as bad as exponentiating using only a multiplication by x operation). On the other side, the level-1 tradeoff can reach $2n$ square gates in memory of order $\sqrt{n}$, and use windowing to make the number of insertions negligible, without meaningfully increasing the number of registers.

4) Our algorithms: We stick to the case where we allow a fixed space. In that case, the final SM sequence plays a negligible role, and the whole circuit works with augmented registers. The g_i grows asymptotically like $\psi^i = \left(\frac{2^c+\sqrt{2^{2c}+4}}{2}\right)^i$. Thus, if we consider the sequence $2^c g_{i-1}, \tilde{h}_{i,0}, \tilde{h}_{i,1}, \dots \tilde{h}_{i,c}, \tilde{h}_{i+1,0}, \dots$, it grows at a rate $\psi^{\frac{1}{c+1}}$. We can remark that $2^c < \psi < 2^c + 2^{-c}$, thus $\log_2(\psi)^{\frac{1}{c+1}} > \frac{c}{c+1}$, which means we can bound the number of F and $\hat{S}$, excluding insertions, by respectively $\frac{n}{c}$ and $2n$. Without a window we can estimate that the density will lie between the ones of Fibonacci and NAF, $1/5$ and $1/3$. We make the (conservative) assumption that the insertions are regularly distributed over $2^c g_i$'s and $\tilde{h}_{i,j}$, costing respectively $1F$ and $2F$. The insertions therefore add an estimated cost of $n(\frac{2}{3} - \frac{1}{3c})F$. Thus in total, the algorithm will use less than $\frac{2n}{3}(1+\frac{1}{c})$ F-gates and $2n$ $\hat{S}$-gates, using $c+3$ augmented registers.

Finally, with a window of size w we can estimate that the weight of the decomposition will be of order of $n/\log(w)$. Hence, the cost becomes $2n\hat{S} + \frac{n}{c}F + \frac{n}{\log(w)}(2-\frac{1}{c})F$. If we have a fixed number of augmented registers k to allocate, we need to balance the values of c and w to minimize this formula with the constraint $k = w+c$. We consider that the last $\frac{1}{c}$ term in the formula is negligible, and therefore we must minimize $(\frac{1}{c} + \frac{2}{\log(w)})$. This resolves in $c = \sqrt{w/2}\log(w)$. Basically, it means that asymptotically, most of the temporary registers are used for the window. For large enough k, we get a cost of about $2n\hat{S} + \frac{2n}{\log k}F$.

5) Comparing the approaches: While the number of registers is easy to compare, we need to know the relative costs of F, S, and $\hat{S}$ to obtain a relative cost per bit and compare the approaches. Ignoring insertions, we have 4 main cases:

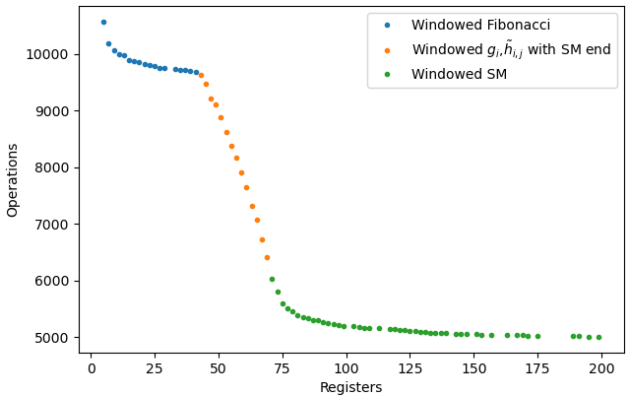


Fig. 5. Tradeoff for an N of 2048 bits, $w \leq 50$, with explicit inverses and F costs 3 S.

- Square-multiply, costs 1 S/bit,
- Level-1 tradeoff, $\sqrt{n}$-memory square-multiply, costs 2 S/bit,
- Fibonacci, costs $\log_\phi(2) \simeq 1.44$ F/bit,
- Our generalization, costs $\frac{1}{c}$ F/bit and 2 $\hat{S}$/bit.

Thus, if F costs roughly the same as S, Fibonacci is optimal, except compared with a memory-inefficient square-multiply with $\mathcal{O}(n)$ registers. In this case the only way to meaningfully reduce the time complexity is to add windowing.

We can also remark that while the sequence $\psi_c^{\frac{1}{c+1}}$ converges towards 2, the first values are decreasing: $\phi = \psi_0 > \sqrt{\psi_1}$, and $\phi = \sqrt[3]{\psi_2}$. As such, for a very limited number of registers Fibonacci will stay more competitive.

B. Costs for a fixed size

We have done extensive benchmarks for an N of 2048 bits, which are presented in Figures 5, 6 and 7. The Python-code in the supplementary material [1] should allow to reproduce them.

1) Finite field case: In this case we estimate that F costs 3 times S, and we can interpolate, as shown in Figure 5. We can see a steep phase transition from Fibonacci to SM. This is due to the fact that Fibonacci and our generalization need to use registers with the inverse, which doubles its memory footprint, while the squaring chain in SM does not.

2) Elliptic curve case: When computing the inverse is free, the behaviour is quite different if F has a cost similar to S, or if we are in a situation where F is more expensive than S. This could for instance be the case if we have to implement F as a complete addition law that is robust to the two input points being equal.

If $F \simeq 2S$, as shown in Figure 6, windowed Fibonacci is quickly outperformed by our generalization, which interpolates smoothly with SM.

If F and S have a similar cost, as shown in Figure 7, using uncomputed squares ends up being more expensive than F, and as such there is no interpolation between windowed Fibonacci and slightly uncomputed square-and-multiply. In

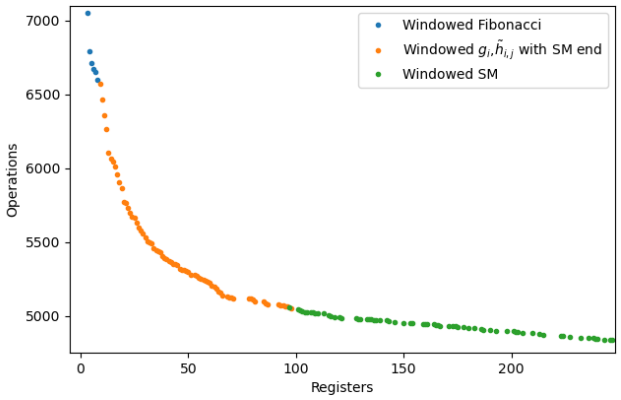


Fig. 6. Tradeoff for an N of 2048 bits, $w \leq 50$, with a free inverse and $F = 2S$.

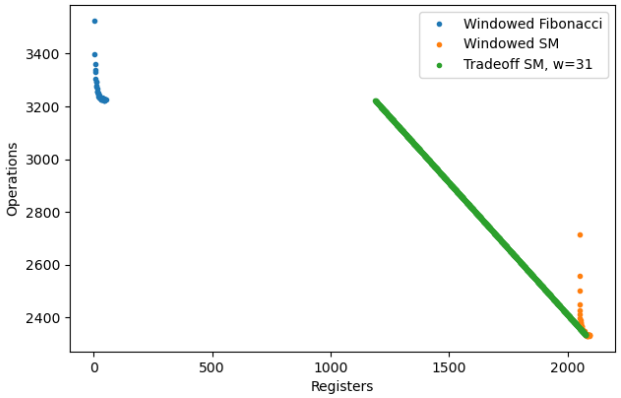


Fig. 7. Tradeoff for an N of 2048 bits, $w \leq 50$, with a free inverse and comparable costs for S and F.

this example adding windowing reduces the cost by $\sim 10\%$, and switching to SM allows to gain another $\sim 30\%$, and overall one can divide the time by less than 2 at the expense of completely blowing up the memory.

We finally remark that for Figures 5 and 6, we have cut the x-axis to zoom on the interesting part, but the picture would continue to the right, up to a point where they reach essentially the same point as the bottom-right point of Figure 7, where the cost is dominated by nS in all the cases.

VII. Conclusion and Future Work

We've shown how to achieve low-memory tradeoffs for quantum exponentiation with a classical exponent. We manage to fully interpolate between Fibonacci and no-uncomputing windowed-SM, except in the case where in-place multiplication has a cost similar to a squaring, where Fibonacci exponentiation, maybe with windowing, is likely to be the more attractive tradeoff, except is one dismisses completely memory considerations.

While this is not the focus of our work, our techniques would be applicable in a context where both the exponent and the value are in quantum superposition. The main difference is that the decomposition of the exponent needs to be done quantumly, and that to leverage a lightweight decomposition, the indices of insertion have to be the same for all exponents.

Another extension would be to leverage intermediate measurements in the algorithm. This is no longer reversible computing, but it allows better generic time-memory tradeoffs. The general framework is the *spooky pebbling games*, and has been studied in [8].

References

[1] —, "Auxiliary material." https://inria.hal.science/hal-05601790v1/file/auxiliary_material.zip

[2] H. Alpert, "Differences of multiple Fibonacci numbers," *Integers*, vol. 9, pp. 745–749, 2009.

[3] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM J. Comput.*, vol. 18, no. 4, pp. 766–776, 1989.

[4] D. J. Bernstein, T. Lange, C. Martindale, and L. Panny, "Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies," in *EUROCRYPT 2019*.

[5] X. Bonnetain and A. Schrottenloher, "Quantum security analysis of CSIDH," in *EUROCRYPT 2020*.

[6] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes, "CSIDH: an efficient post-quantum commutative group action," in *ASIACRYPT 2018*.

[7] T. Häner, S. Jaques, M. Naehrig, M. Roetteler, and M. Soeken, "Improved quantum circuits for elliptic curve discrete logarithms," in *PQCrypto 2020*.

[8] G. D. Kahanamoku-Meyer, S. Ragavan, and K. V. Kirk, "Parallel spooky pebbling makes Regev factoring more practical," *IACR Cryptology ePrint Archive*, 2025. https://eprint.iacr.org/2025/1887

[9] B. S. Kaliski Jr, "Targeted Fibonacci exponentiation," *arXiv preprint*, 2017. https://arxiv.org/abs/1711.02491

[10] P. Kaye, R. Laflamme, and M. Mosca, *An Introduction to Quantum Computing*. Oxford University Press, 2006.

[11] E. W. Knudsen, "Elliptic scalar multiplication using point halving," in *ASIACRYPT'99*. Springer, 1999.

[12] G. Kuperberg, "A subexponential-time quantum algorithm for the dihedral hidden subgroup problem," *SIAM Journal on Computing*, vol. 35, no. 1, pp. 170–188, 2005.

[13] H. T. Larasati, D. S. C. Putranto, R. W. Wardhani, and H. Kim, "Reducing the depth of quantum FLT-based inversion circuit," *IACR Cryptology ePrint Archive*, 2022. https://eprint.iacr.org/2022/463

[14] N. Méloni, "New point addition formulae for ECC applications," in *Arithmetic of Finite Fields, First International Workshop, WAIFI*, vol. 4547, 2007, pp. 189–201.

[15] S. J. Miller and Y. Wang, "From Fibonacci numbers to central limit type theorems," *Journal of Combinatorial Theory, Series A*, vol. 119, no. 7, pp. 1398–1413, 2012.

[16] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information (10th Anniversary edition)*. Cambridge University Press, 2016.

[17] D. S. C. Putranto, R. W. Wardhani, H. T. Larasati, and H. Kim, "Space and time-efficient quantum multiplier in post quantum cryptography era," *IEEE Access*, vol. 11, pp. 21 848–21 862, 2023.

[18] S. Ragavan, "Regev factoring beyond Fibonacci: Optimizing prefactors," *IACR Cryptology ePrint Archive*, 2024. https://eprint.iacr.org/2024/636

[19] S. Ragavan and V. Vaikuntanathan, "Space-efficient and noise-robust quantum factoring," in *CRYPTO 2024*.

[20] O. Regev, "An efficient quantum factoring algorithm," *Journal of the ACM*, vol. 72, no. 1, pp. 1–13, 2025.

[21] M. Roetteler, M. Naehrig, K. M. Svore, and K. E. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," in *ASIACRYPT 2017*.

[22] P. W. Shor, "Polynominal time algorithms for discrete logarithms and factoring on a quantum computer," in *Algorithmic Number Theory*, 1994.

[23] S. Wang, X. Li, W. J. B. Lee, S. Deb, E. Lim, and A. Chattopadhyay, "A comprehensive study of quantum arithmetic circuits," *Philosophical Transactions A*, vol. 383, no. 2288, p. 20230392, 2025.

[24] S. Wang, A. Mondal, and A. Chattopadhyay, "Optimal Toffoli-depth quantum adder," *ACM Trans. Quantum Comput.*, vol. 6, pp. 25:1–25:16, 2025.