Improving large-Scale Simulation efficiency of Shor's Quantum Factoring Algorithm

DISSERTATION SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Cryptology & Security

by

Pushpendra Pal

[Roll No: CRS2209]

under the guidance of

Mr. Prashant Verma (Primary supervisor)
Scientist 'E', Scientific Analysis Group (SAG)
Defence Research & Development Organization (DRDO)

Dr. Goutam Kumar Paul (Secondary Supervisor)
Associate Professor
Cryptology & Security Research Unit

Indian Statistical Institute, Kolkata
Kolkata-700108, India

July 2024

Acknowledgement

I would like to express my deepest gratitude to my supervisors, Mr. Prashant Verma from the Scientific Analysis Group (SAG) at the Defence Research and Development Organisation (DRDO), and Dr. Goutam Kumar Paul from the Indian Statistical Institute, Kolkata, for their continuous guidance and unwavering support throughout my thesis. Their expertise, patience, and encouragement have been invaluable, providing me with countless opportunities to learn and grow both personally and professionally. Their feedback and suggestions have not only refined my research skills but also deepened my appreciation for high-quality work.

I extend my heartfelt thanks to the faculties of the Indian Statistical Institute for their support and assistance. Their guidance and readiness to help with any queries have been crucial in navigating the challenges of my research. The knowledge they have imparted has significantly contributed to my academic development.

Finally, I acknowledge the collaborative environment, the CSSC, of the Indian Statistical Institute, which has greatly enriched my research experience. The resources, seminars, and workshops provided a platform to learn, present, and discuss my work with peers and experts in the field.

Once again, I am deeply thankful to everyone who has supported me throughout this journey. Their collective efforts have made this thesis possible and enriched my academic experience.

Pushpendra Pal
Roll No. CRS2209
Indian Statistical Institute, Kolkata
Kolkata - 700108, India

Certificate

This is to certify that the dissertation titled “Improving large-Scale Simulation efficiency of Shor’s Quantum Factoring Algorithm” submitted by Pushpendra Pal to Indian Statistical Institute, Kolkata, in fulfilment for the award of the degree of Master of Technology in Cryptology & Security is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the rules and regulations of this institute and, according to me, has reached the standard needed for submission.

Goutam Paul

Associate Professor

Cryptology and Security Research Unit (CSRU)

Indian Statistical Institute, Kolkata

Abstract

Shor’s factoring algorithm is one of the most eagerly awaited applications of quantum computing, promising to revolutionise fields such as cryptography by efficiently factoring large integers, a task that is computationally intensive for classical computers. Currently, the limited capabilities of today's quantum computers allow for testing Shor’s algorithm only on very small numbers, as they do not yet possess the qubit count or error rates necessary to handle larger, more complex computations.

In this study, we demonstrate how large GPU-based supercomputers can be leveraged to evaluate the performance of Shor’s algorithm on numbers that are beyond the reach of current and near-term quantum hardware. By simulating the algorithm on powerful classical machines, we can gain insights into its behaviour and performance under various conditions.

Initially, we examine Shor’s original factoring algorithm and find that, despite theoretical success probabilities being quite low, the average success rates are significantly higher due to frequent "lucky" cases. These are instances where factorization succeeds even when the sufficient conditions of the algorithm are not met. This phenomenon suggests that practical implementations of Shor’s algorithm may benefit from a higher-than-expected probability of success.

We then explore a robust post-processing method designed to increase the success probability of Shor’s algorithm to nearly one with just a single execution. This method involves additional classical computational steps that refine the output of the quantum algorithm, making the overall factorization process much more reliable.

Finally, we analyse the effectiveness of this post-processing method in the presence of typical quantum processing hardware errors, such as decoherence and gate errors. Our findings show that the quantum factoring algorithm displays a unique form of universality and resilience against various errors. This resilience suggests that Shor’s algorithm, when combined with efficient post-processing techniques, can remain viable even in less-than-ideal quantum hardware environments.

The largest semiprime we have factored using Shor’s algorithm on a GPU-based supercomputer, without prior knowledge of the factors, represents a significant milestone. It

poses a formidable challenge for the quantum computing community to exceed without resorting to oversimplifications on any quantum computing device. This achievement underscores the potential of classical-quantum hybrid approaches in advancing our understanding and implementation of quantum algorithms.

Contents

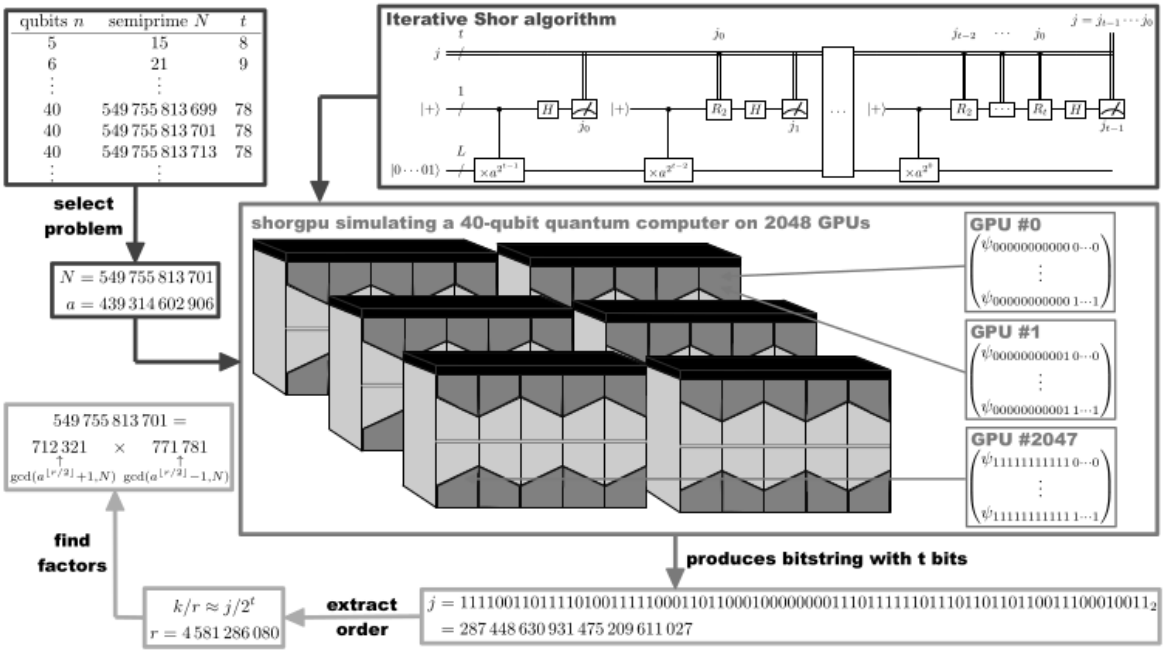

Fig. 1. Scheme to test Shor's algorithm. After selecting an L-bit semiprime N = p × q to factor and a random integer 1 < a < N coprime to N (blue), a quantum computer or quantum computer simulator with n = L + 1 qubits runs the iterative Shor algorithm (red) and produces several bitstrings j with t bits (green). Here, "iterative" means that one qubit is measured and reused t times to produce the t classical bits of each j. Every bitstring j is analysed using the Shor post-processing method (yellow), independent of whether certain algorithmic requirements on j are satisfied or not. Here, k (r) denotes the numerator (denominator) obtained from a continued fraction expansion of j/2t. Note that the expression for the factors in the yellow section is specific to Shor's post-processing; We remark that conceptually, it does not matter whether the green section is performed by a quantum computer simulator or a real quantum computing device. Reprinted from "Large-Scale Simulation of Shor's Quantum Factoring Algorithm," by Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, Kristel Michielsen, *arXiv*, arXiv:2308.05047.

Simulation

The text outlines the process and tools used for simulating the iterative Shor algorithm for factoring large numbers. It describes the use of a specialised tool to handle the simulation by processing a quantum state vector through a quantum circuit designed for factoring. Each step in the circuit corresponds to an operation on the quantum state vector, which consists of numerous complex numbers distributed across the memory of multiple GPUs. Communication between these GPUs is managed through a specific interface. For verification purposes, a separate universal quantum computer simulator, previously used for simulating conventional quantum algorithms with a significant number of qubits, is employed. Enhancements were made to this simulator to accommodate the iterative version of the algorithm. However, the universal simulator becomes less efficient as the problem size increases due to its poor distribution capabilities over many processing units. In contrast, the

specialised algorithm in the dedicated tool performs the same simulation significantly faster using the same number of GPUs. For the largest simulated problem, the dedicated tool completes the simulation much quicker than the universal simulator, even when using a large number of GPUs. The results produced by the iterative Shor algorithm using the specialised tool match those of the universal simulator for the sizes that can be handled by the latter. This summary highlights the advancements in quantum algorithm simulation, showcasing the use of GPU-based tools to tackle the challenges of memory and computation in large-scale simulations, and underscores the efficiency and scalability improvements achieved with the dedicated tool.

Initialization

To simulate the iterative Shor algorithm for factoring an L-bit semiprime N , shorgpu simulates the full quantum circuit with n = L+1 qubits shown. This is done by computing all complex coefficients of the state vector

$$|\psi\rangle = \sum_{k_L \cdots k_0 = 0,1} \psi_{k_L \cdots k_0} \, |k_L \cdots k_0\rangle = \begin{pmatrix} \psi_{0\cdots 00} \\ \psi_{0\cdots 01} \\ \vdots \\ \psi_{1\cdots 11} \end{pmatrix}.$$

The distributed memory communication between the GPUs uses CUDA-aware MPI.These 2^{L+1} complex double-precision numbers ψkL ⋯k0 are distributed over $N_{GPU} \in \{2, 4, 8, \ldots, 2048\}$ GPUs.

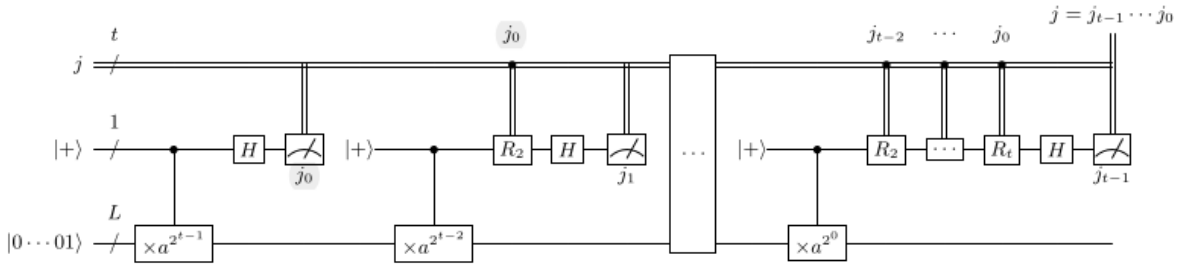

FIG. 2. **Quantum circuit of the iterative Shor algorithm.** The circuit consists of $L+1$ qubits that undergo t separate stages $\texttt{cbit} = 0, \ldots, t-1$, in which the classical bit $j_{\texttt{cbit}}$ is measured. Each stage starts with the first qubit in the initial state $|+\rangle$ and ends with this qubit being measured (middle row). Between initialization and measurement, each stage consists of a controlled modular multiplication (bottom row) with some power of a (see Eq. (6)), then a rotation gate controlled by all previously measured classical bits (see Eq. (8)), and finally a Hadamard gate. The resulting bit $j_{\texttt{cbit}}$ is used to assemble the classical bitstring $j = j_{t-1} \cdots j_0$ (top row).

Fig. 2. Iterative Shor algorithm circuit. Reprinted from "Large-Scale Simulation of Shor's Quantum Factoring Algorithm," by Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, Kristel Michielsen, *arXiv*, arXiv:2308.05047.

Controlled Modular Exponentiation

The above figure shows shor's algorithm, which apparently looks simple if you do not account for the modular exponentiation which has been pressed to a controlled operation box. But this is the bottleneck of the algorithm with respect to time as well as space. If we look at the original P.W. Shor's paper, the exponentiation is done using the Schönhage–Strassen algorithm [1971]. This gives a gate array for integer multiplication that uses O(l log l log log l) gates to multiply two l-bit numbers.

Thus, asymptotically, modular exponentiation requires $O(l^2 \log l \log \log l)$ time. This gives a gate array for integer multiplication that uses O(l log l log log l) gates to multiply two l-bit numbers.

Thus, asymptotically, modular exponentiation requires O(l2 log l log log l) time.

$$CU_{\mathtt{a}} \left|x\right\rangle \left|y\right\rangle = \begin{cases} \left|0\right\rangle \left|y\right\rangle & (x = 0) \\ \left|1\right\rangle \left|\mathtt{a}y \bmod N\right\rangle & (x = 1 \text{ and } 0 \leq y < N) \\ \left|1\right\rangle \left|y\right\rangle & (x = 1 \text{ and } N \leq y) \end{cases}, \tag{6}$$

Rotation Gate

After the oracle gate, each stage (except the first stage) of the quantum circuit in Fig. 2 contains a sequence of rotation gates defined by

$$R_l = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^l} \end{pmatrix}.$$

These are controlled by bits resulting from previous measurements. Specifically, at the stage cbit = 0, . . . , t − 1, in which the classical bit jcbit is being measured, the sequence of these controlled rotation gates reads

$$\prod_{l=2}^{1+\mathtt{cbit}} CR_l = \prod_{l=2}^{1+\mathtt{cbit}} \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i j_{1+\mathtt{cbit}-l}/2^l} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\varphi_{\mathtt{cbit}}} \end{pmatrix},$$

where the phase φcbit at stage cbit amounts to

$$\varphi_{\texttt{cbit}} = 2\pi \sum_{l=2}^{1+\texttt{cbit}} \frac{j_{1+\texttt{cbit}-l}}{2^l} = \frac{\pi j^{(\texttt{cbit})}}{2^{\texttt{cbit}}},$$

and j (cbit) = jcbit−1 jcbit−2 · · · j1 j0 is the integer assembled from all classical bits measured up to this point.

Hadamard Gate

The implementation of the Hadamard gate on the first qubit transforms the statevector coefficients as For every GPU in the mpi x = 0 group, this requires two sided MPI communication with exactly one GPU in the mpi x = 1 group.

$$\begin{aligned}
&\psi_{0\,\mathrm{bin}(\texttt{mpi_xrank})*\cdots*} \\
&\quad \leftarrow \frac{\psi_{0\,\mathrm{bin}(\texttt{mpi_xrank})*\cdots*} + \psi_{1\,\mathrm{bin}(\texttt{mpi_xrank})*\cdots*}}{\sqrt{2}}, \\
&\psi_{1\,\mathrm{bin}(\texttt{mpi_xrank})*\cdots*} \\
&\quad \leftarrow \frac{\psi_{0\,\mathrm{bin}(\texttt{mpi_xrank})*\cdots*} - \psi_{1\,\mathrm{bin}(\texttt{mpi_xrank})*\cdots*}}{\sqrt{2}}.
\end{aligned}$$

Measurement operation

At the end of each stage in Iterative shor's, the classical bit jcbit is measured, where cbit = 0, . . . , t − 1 enumerates the stage. This amounts to adding up the probabilities

$$p_1 = \sum_{k_{L-1}\cdots k_0=0,1} |\psi_{1\,k_{L-1}\cdots k_0}|^2,$$

which is an MPI reduction over all GPUs belonging to the mpi_x = 1 group.

Reset operation

This operation requires an MPI transfer of all coefficients from the GPUs in the group mpi x = jcbit to the GPUs in the group mpi x = 1 − jcbit. It performs the reinitialization of the first qubit

in |+⟩ at the same time. If the result of the measurement is given by jcbit = 0, 1, this operation is done by transforming all coefficients according to

$$\begin{pmatrix} \psi_{00\cdots0} \\ \vdots \\ \psi_{01\cdots1} \\ \psi_{10\cdots0} \\ \vdots \\ \psi_{11\cdots1} \end{pmatrix} \leftarrow \begin{pmatrix} \psi_{j_{\mathrm{cbit}}0\cdots0}/\sqrt{2p_{j_{\mathrm{cbit}}}} \\ \vdots \\ \psi_{j_{\mathrm{cbit}}1\cdots1}/\sqrt{2p_{j_{\mathrm{cbit}}}} \\ \psi_{j_{\mathrm{cbit}}0\cdots0}/\sqrt{2p_{j_{\mathrm{cbit}}}} \\ \vdots \\ \psi_{j_{\mathrm{cbit}}1\cdots1}/\sqrt{2p_{j_{\mathrm{cbit}}}} \end{pmatrix}$$

Results

In this section, we describe and interpret the results obtained from simulating Shor's algorithm according to Fig. 1. For our analysis, we generated 61362 factoring problems (N, a), 52077 of which were chosen to have uniformly distributed prime factors to ensure unbiased results, and the rest comprise individual factoring problems for large semiprimes. We consider Shor's original post-processing.

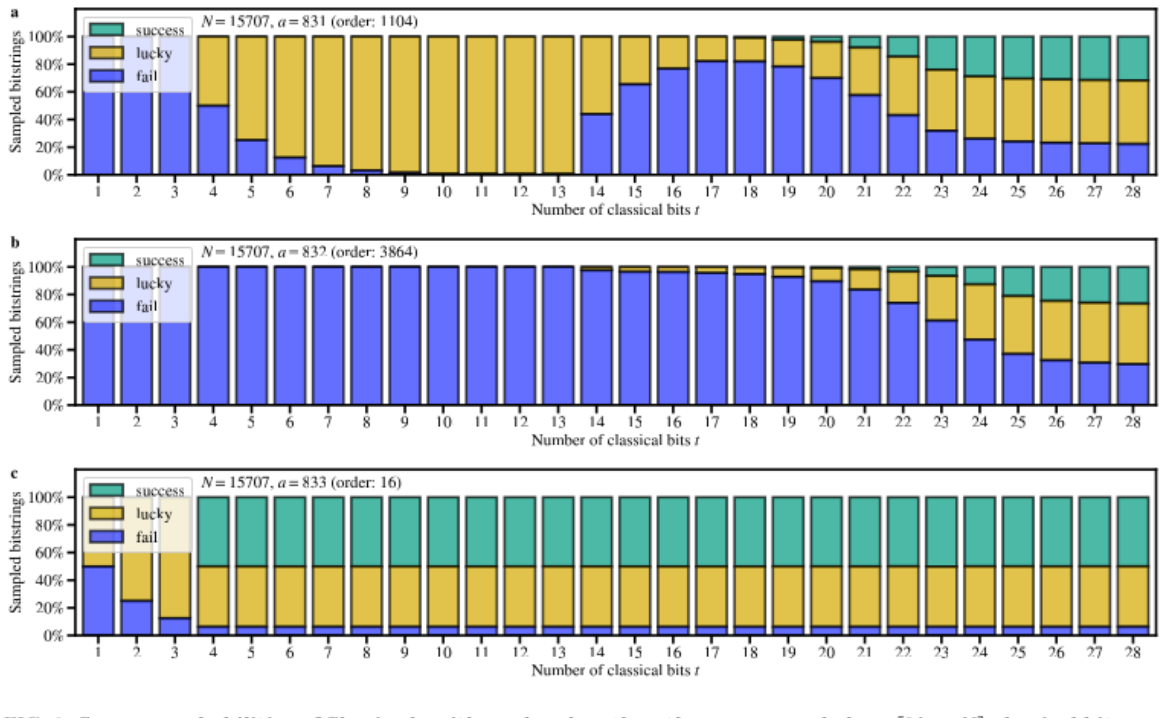

Fig. 3. Success probabilities of Shor's algorithm when less than the recommended t = ⌈2 log2 N ⌉ classical bits are extracted from the QFT. Every bar represents the fraction of 1 million sampled bitstrings classified as

“success” (green), “lucky” (yellow), and “fail” (blue). The factored semiprime is N = 15707 (such that t = 28) with i) a = 831, ii) a = 832, and iii) a = 833. The corresponding orders of a modulo N are indicated at the top. Increasing t beyond 28 does not further improve the success probabilities. Reprinted from "Large-Scale Simulation of Shor’s Quantum Factoring Algorithm," by Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, Kristel Michielsen, *arXiv*, arXiv:2308.05047.

Using Shor’s Post-Processing

A given factoring problem for Shor’s algorithm consists of a semiprime N and a random integer 1 < a < N coprime to N . For each such factoring problem (N, a), Shor’s algorithm produces a sample of M bitstrings (we typically consider M = 1024 samples). Each bitstring j is analyzed using the so-called standard procedure. If all checks on j from the standard procedure pass, the algorithm was successful and we count the bitstring j as “success”. However, if certain checks on j fail, we still evaluate j and test if it yields a factor of N . If it does, we count this factoring attempt as “Lucky”.

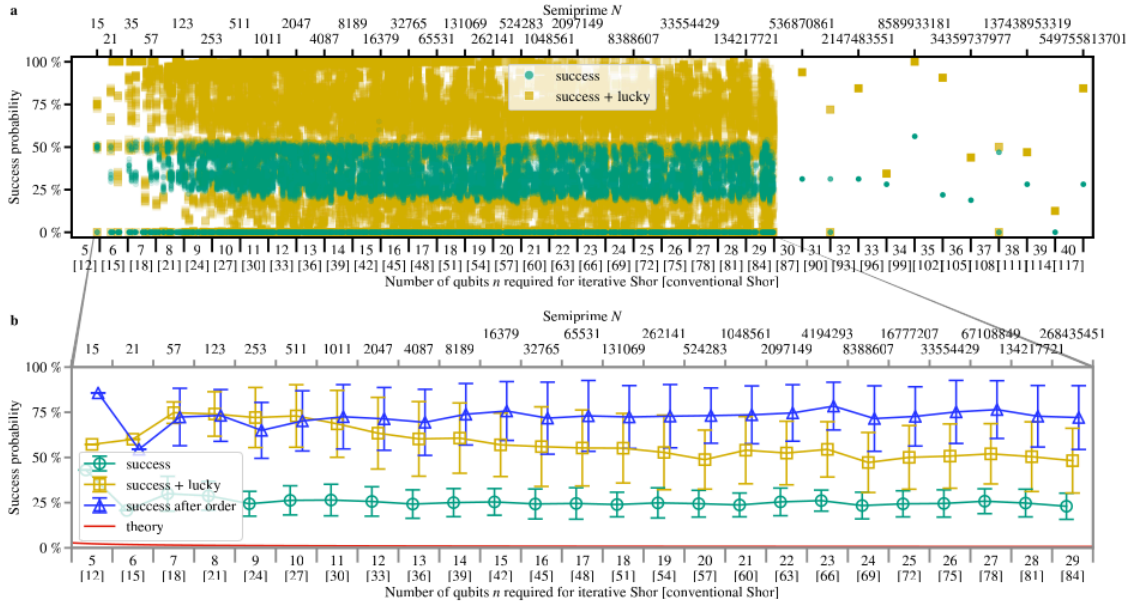

FIG. 4. Success probabilities for Shor’s factoring algorithm. Reprinted from "Large-Scale Simulation of Shor’s Quantum Factoring Algorithm," by Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, Kristel Michielsen, *arXiv*, arXiv:2308.05047.

Figure 4a shows a scatter plot of all “success” and “success+lucky” probabilities for all uniformly distributed problems (n < 30 qubits) and the individual large problems (30 ≤ n ≤ 40

qubits). Surprisingly, “lucky” occurs much more often than expected. In Fig. 4b, we see that on average only 25 % of all bitstrings yield “success”. Including the “lucky” cases,
however, a factor of the semiprime N can be extracted from over 50 % of all bitstrings on average. Additionally, all average success probabilities are significantly larger
than the theoretical bound of 3–4 %. We conjecture that asymptotically, the average success probability for “success+lucky” approaches 50 % from above below, where we give a classification of the different “lucky” scenarios and show that the main contribution saturates around 25 %). This observation is remarkable, as it shows that factoring a semiprime with Shor’s algorithm is often successful, even though the order-finding procedure actually fails. Simulating Shor’s algorithm for semiprimes N between 536870861 and 549755813701 requires substantial computational resources. Therefore, only individual cases are shown in Fig. 4a. These cases correspond to the largest “interesting” semiprimes for a given number of qubits n = 30, . . . , 40. A noteworthy case is the factoring problem for (N, a) = (8589933181, 3974323683) (n = 34 qubits). Here, the “lucky” cases raise the success probability from 56.25 % to 100 % (yellow square) among all M bitstrings. Furthermore, the factoring problem for (N, a) = (274877906893, 226009433972) (n = 39 qubits,
(second from the right) has a success probability of 0% when the sufficient conditions for Shor’s algorithm are presupposed (green circle). However, when ignoring the violations of these conditions, we find that Shor’s algorithm can indeed factor NNN with a “lucky” success probability of 12.5% (yellow squares). The unexpectedly large success probabilities when the lucky cases are included prompt the question: “How many bitstrings do we need to sample until a factor is found?”

Classification of the “Lucky” Scenarios

If a sampled bitstring j does not pass the standard tests required by Shor’s algorithm, but still produces a factor with the procedure shown in Fig. 1, we call this a “lucky” case.

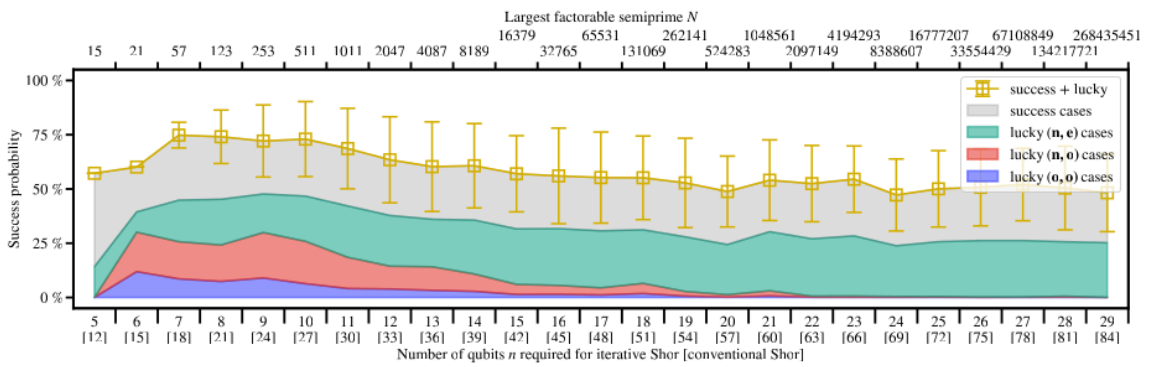

Fig 5: Breakdown of the average success probabilities for successful factoring scenarios. The contributions to the average “success+lucky” probability (yellow squares) from Fig. 4b are divided into the “success” cases (gray), the lucky (n,e) cases (green), the lucky (n,o) cases (red), and the lucky (o,o) cases (blue). Note that an L-bit semiprime requires n = L + 1 qubits for the iterative Shor algorithm and roughly 3L qubits using the conventional Shor algorithm, as indicated on the bottom axis. The largest odd semiprime that can be factored with a given number of qubits is shown on the top axis. Lines are guides to the eye. Reprinted from "Large-Scale Simulation of Shor’s Quantum Factoring Algorithm," by Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, Kristel Michielsen, *arXiv*, arXiv:2308.05047.

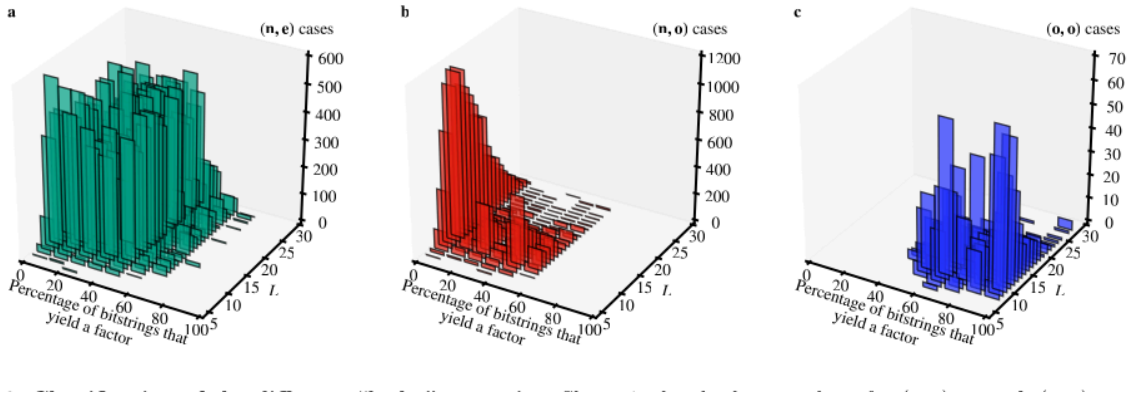

Fig 6: Classification of the different “lucky” scenarios. Shown is the absolute number of a (n,e) cases, b (n,o) cases, and c (o,o) cases, which yield a factor even though the sufficient conditions for Shor’s algorithm are not met. For each bit length L = 9, . . . , 29, the total number of cases is given by 2500 uniformly distributed factoring problems (for smaller L, the total number is smaller than 2500 because all possibilities for factoring problems (N, a) are exhausted. Every bar represents a 10 %-wide half-open interval (P − 10 %, P] with P = 10 %, 20 %, . . . , 100 % for the percentage of all M = 1024 bitstrings that yield a factor in this case. For instance, the large leftmost red bar at L = 11 and P = 10 % in panel b means that in 1197 out of 2500 cases, up to 10 % of all sampled bitstrings yield a factor even though they represent an (n,o) case, i.e., they produce an odd number r that is not the order. Similarly, the large rightmost blue bar at L = 9 in panel c means that in 68 out of 2500 cases, more than 90 % of all sampled bitstrings yield a factor even though the correctly extracted order r = ˆr is odd. Reprinted from "Large-Scale Simulation of Shor’s Quantum Factoring Algorithm," by Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, Kristel Michielsen, *arXiv*, arXiv:2308.05047.

For a given factoring problem with an L-bit semiprime N and an integer a coprime to N , let r̂ be the multiplicative order of a modulo N . Furthermore, let r be the denominator and k be the numerator extracted from the convergent k/r to $j/2^t$ using the continued fractions algorithm from the standard procedure (i.e., the largest r < N such that k/r is a convergent to $j/2^t$ with $|k/r - j/2^t| \leq 1/2r^2$). We distinguish between three scenarios in which the standard checks for Shor’s algorithm fail:

- (n,e) r /= r̂ is not the order but r is even, $j \in \{\text{round}(\hat{k} \times 2^t/\hat{r}) : \hat{k} = 0, \ldots, \hat{r} - 1\}$,
- (n,o) r /= r̂ is not the order and r is odd,
- (o,o) r = r̂ is the order but the order is odd.

In Fig. 6, we show a breakdown of the average success probability for these scenarios. We see that the (n,e) scenario makes up a large fraction of the successful factorizations and its relevance grows for larger integers. In contrast, the (n,o) and (o,o) scenarios, where the ex-

tracted r is odd, only matter for smaller integers. We explain the reasons for this in the discussions of each individual scenario below. Fig 7 shows the number of cases for each scenario among the 52077 uniformly drawn factoring problems. We see that the cases where bitstrings yield a factor in the (n,e) scenario are responsible for a significant fraction of all successful factorizations. Indeed, as Fig. 6 suggests, the relevance of this scenario also grows on average and tends to saturate above 25 % for larger L. We expect that this contribution persists for even larger semiprimes. Contributions from the (n,o) and (o,o) scenarios seem to be responsible only for a small number of successful factorizations, and mostly only for small semiprimes upto L = 10. It is remarkable, however, that for many factoring problems that can be factored in the (o,o) scenario, 50–100% of all bitstrings yield a factor. To understand these effects, we discuss the different scenarios individually. The goal is to obtain an understanding for why the different scenarios occur. The number-theoretic ideas are similar to the algorithm used in [24], which goes back to Miller's algorithm.

Contributions

The implementation of this is already present although has some configuration issues which made compiling the program impossible to compile. Here is an excerpt of the original makefile

```
MPICPP ?= mpic++
NVCCFLAGS ?= -g -lineinfo
NVHPCFLAGS ?= -std=c++17 -acc -fast -gopt -gpu=lineinfo -Minfo=accel -cuda
NVHPCDEFS ?=
NVHPCINCS ?=
NVHPCLIBS ?=
EXE ?= shorgpu

all: $(EXE:%=build/%)

build/%: obj/%.o
	$(MPICPP) $(NVHPCFLAGS) $(NVHPCINCS) $^ -o $@ $(NVHPCLIBS)

obj/%.o: src/%.cpp
	$(MPICPP) $(NVHPCFLAGS) $(NVHPCINCS) $(NVHPCDEFS) -c $< -o $@

clean:
	$(RM) $(EXE:%=build/%) $(EXE:%=obj/%.o)

rebuild:
	$(MAKE) clean
	$(MAKE)

.PHONY: clean rebuild
.PRECIOUS: $(wildcard src/*)
.SECONDARY:
```

And, here is the modified makefile, which could compile the program.

```
# Compiler settings
CUDA_INCLUDE = /usr/local/cuda-12.2/include
CUDA_LIB = /usr/local/cuda-12.2/lib64

MPICPP ?= mpic++
NVCCFLAGS ?= -g -lineinfo
NVHPCFLAGS ?= -std=c++17 -O3
NVHPCDEFS ?=
NVHPCINCS ?= -I$(CUDA_INCLUDE)
NVHPCLIBS ?= -fopenacc -lcudart
EXE ?= shorgpu

# MPI libraries
MPI_LIBS = -lmpi

all: $(EXE:%=build/%)

build/%: obj/%.o
	$(MPICPP) $(NVHPCFLAGS) $(NVHPCINCS) -L$(CUDA_LIB) $^ -o $@ $(NVHPCLIBS) $(MPI_LIBS)

obj/%.o: src/%.cpp
	$(MPICPP) $(NVHPCFLAGS) $(NVHPCINCS) $(NVHPCDEFS) -c $< -o $@

clean:
	$(RM) $(EXE:%=build/%) $(EXE:%=obj/%.o)

rebuild:
	$(MAKE) clean
	$(MAKE)

.PHONY: clean rebuild
.PRECIOUS: $(wildcard src/*)
.SECONDARY:
```

Here are the changes:

- Compiler Settings:
  - Added `CUDA_INCLUDE`:

        #makefile
        CUDA_INCLUDE = /usr/local/cuda-12.2/include

  - Added `CUDA_LIB`:

        #makefile
        CUDA_LIB = /usr/local/cuda-12.2/lib64

  - Modified `NVHPCINCS` to include `CUDA_INCLUDE`:

        #makefile
        NVHPCINCS ?= -I$(CUDA_INCLUDE)

- Linker Flags:
  - Added `-fopenacc -lcudart` to `NVHPCLIBS`:
    #makefile
    NVHPCLIBS ?= -fopenacc -lcudart

- MPI Libraries:
  - Introduced `MPI_LIBS` variable:
    #makefile
    MPI_LIBS = -lmpi

- Linker Command in `build/%` Rule:
  - Updated to include `-L$(CUDA_LIB)` and `$(MPI_LIBS)`:
    #makefile
    $(MPICPP) $(NVHPCFLAGS) $(NVHPCINCS) -L$(CUDA_LIB) $^ -o $@ $(NVHPCLIBS) $(MPI_LIBS)

These changes enhance the Makefile to support CUDA-related paths and libraries and integrate MPI libraries, improving its functionality for building CUDA applications.

References

1. Willsch, D., Willsch, M., Jin, F., De Raedt, H., & Michielsen, K. (2023). Large-scale simulation of Shor's quantum factoring algorithm. arXiv, https://arxiv.org/pdf/2308.05047
2. Shor, P. W. (1996). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer (Version 2). arXiv. https://arxiv.org/pdf/quant-ph/9508027
3. QIP (2023). 'shorgpu'. GitLab. Retrieved July 3,2024, from https://jugit.fz-juelich.de/qip/shorgpu
4. Google Colab TPU, Google, 10 July 2024. [Online]. Available: https://colab.research.google.com/
5. Kitaev, A. Y. (1995). Quantum measurements and the Abelian Stabilizer Problem. arXiv. https://arxiv.org/abs/quant-ph/9511026
6. Griffiths, R. B., & Niu, C.-S. (1996). Semiclassical Fourier transform for quantum computation. Physical Review Letters, 76(17), 3228-3231. https://doi.org/10.1103/PhysRevLett.76.3228
7. Pal, P. (2024, July). Implementing large-scale simulation of Shor's quantum factoring algorithm. Presented at the ASU Seminar Room, Indian Statistical Institute, Kolkata.