
Proceedings Paper:
Tarabkhah, M. orcid.org/0000-0002-5070-1898, Delavar, M. orcid.org/0000-0001-6354-330X, Doosti, M. orcid.org/0000-0003-0920-335X et al. (1 more author) (2025) Synthesis of quantum simulators by compilation. In: Doerfert, J., Grosser, T., Leather, H. and Sadayappan, P., (eds.) CGO '25: Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization. CGO '25: 23rd ACM/IEEE International Symposium on Code Generation and Optimization, 01-05 Mar 2025, Las Vegas, Nevada, United States. ACM , pp. 284-298. ISBN 9798400712753

https://doi.org/10.1145/3696443.3708949

Synthesis of Quantum Simulators by Compilation

Meisam Tarabkhah
University of Edinburgh
Edinburgh, United Kingdom
m.tarabkhah@ed.ac.uk

Mahshid Delavar
University of Sheffield
Sheffield, United Kingdom
m.delavar@sheffield.ac.uk

Mina Doosti
University of Edinburgh
Edinburgh, United Kingdom
mdoosti@ed.ac.uk

Amir Shaikhha
University of Edinburgh
Edinburgh, United Kingdom
amir.shaikhha@ed.ac.uk

Abstract

Quantum simulation plays a critical role in advancing our understanding and development of quantum algorithms, quantum computing hardware, and quantum information science. Despite the availability of various quantum circuit simulators, they often face challenges in terms of maintainability and extensibility. In this paper, we introduce Lightweight Functional Quantum Simulator (LFQS), a compilation-based framework that addresses these issues by leveraging an intermediate language to synthesize efficient quantum simulators in C and CUDA. The intermediate language exploits the underlying structured sparsity behind the matrix representation of quantum gates. Our framework generates efficient code for multi-core CPUs and GPUs and outperforms state-of-the-art simulators, such as Qiskit, Catalyst, and QuEST, as well as a representative of dense tensor frameworks, NumPy, in both micro-benchmark and macro-benchmark circuits.

CCS Concepts: • **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Compilers**.

Keywords: Sparse tensor algebra, rewrite systems

ACM Reference Format:
Meisam Tarabkhah, Mahshid Delavar, Mina Doosti, and Amir Shaikhha. 2025. Synthesis of Quantum Simulators by Compilation. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3696443.3708949

1 Introduction

Quantum Computing plays an important role in advancing technology and solving complex problems that classical computers struggle to handle [5, 8, 28, 32]. However, quantum machines are not yet fully developed and available. One of the main tools for studying quantum algorithms is to simulate them on classical computers. These simulations are particularly valuable for testing and validating quantum algorithms before deploying them on actual quantum hardware [31, 32].

Quantum objects can be handled classically as complex vectors and matrices. However, the size of such quantum vector spaces grows exponentially with the number of qubits. Therefore, classical simulation of quantum computing requires huge memory and computational resources. Thus, developing optimized methods for these linear algebraic transformations is crucial. Currently, several state-of-the-art quantum simulators are available, including open-source frameworks such as QuEST [21], Qiskit [33] and Catalyst [19].

Existing quantum simulators are developed using hand-tuned, low-level implementations by computational physicists rather than computer scientists. Despite being very efficient, they have the following issues. First, although their low-level codebases contain various optimizations manually performed by the developers, many optimizations are missing. Second, the large codebase sizes make the maintenance challenging; any changes to the data layout, algorithm improvements, or additional support for new backends require modifications or re-implementations of all related functions.

In recent years, significant efforts have been made to utilize intermediate languages for quantum simulations to address some of these challenges [7, 12, 20, 26, 30]. However, they mostly focus on global circuit optimizations rather than gate-level optimizations; they still rely on low-level gate implementations in their developer-provided runtimes, which misses optimization opportunities (see Section 7).

Inspired by the techniques employed in optimizing compilers, we developed the first compilation stack to synthesize an efficient quantum simulator. As opposed to the previous systems, which rely on developer-provided low-level gate implementations, we employ a compilation-based approach to generate these low-level implementations systematically. Our proposed framework, Lightweight Functional Quantum Simulator (LFQS), outperforms state-of-the-art quantum simulators and offers improved maintainability and extensibility.

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708949

This paper makes the following contributions:

Quantum Simulator Synthesis. We built LFQS, a quantum simulator centered around compilation (Section 3). Starting from the high-level specification of quantum gates and circuits in linear algebra, LFQS employs an intermediate language to capture the structure of the computation and transforms it into a more efficient representation.

Maintainability and Extensibility. Thanks to the framework's modular design, it has better maintainability than the hand-tuned competitors. LFQS captures novel algorithms for quantum simulation as an optimization in the intermediate language (Section 4). The code generator of LFQS translates this specification into efficient parallel low-level C and CUDA code (Section 5.1). However, the modular design of LFQS enables us to extend it with new backends; we only need to provide an alternative code generator for each new backend. Furthermore, we showcase the extensibility by modifying the data layout of vectors and matrices, and by adding support for a GPU backend (Sections 5.3, 6.6, 6.7, and 6.8).

Performance Superiority. We demonstrate that LFQS is competitive with state-of-the-art simulators while being significantly faster than library-based solutions and tensor frameworks in handling the quantum circuits (Section 6). For benchmarking, we investigated the performance of LFQS in circuits with individual gates as well as multi-gate circuits from eight real-world quantum circuits that are highly significant in quantum computing (Table 1).

2 Background and Related Work

In this section, we first introduce the background for quantum computation. Then, we cover the most relevant state-of-the-art on quantum simulation. Finally, we introduce the intermediate language that LFQS uses and extends.

2.1 Basics of Quantum Computation

Qubit. Quantum bit (qubit) [27] is the fundamental unit in quantum computing. A qubit is a two-level physical system with quantum behaviour. Quantum states are unit vectors in a complex-valued vector space with an inner product equipped with a set of d orthonormal vectors called a basis. In the case of a single qubit where $d = 2$, the following set of vectors are a complete basis (a.k.a. computational bases):

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

and any qubit state can be written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ where $|\alpha|^2 + |\beta|^2 = 1$ for some $\alpha, \beta \in \mathbb{C}$. The above form is called a superposition of two quantum states. Quantum states can be composed using the tensor product as

$$|\psi_{AB}\rangle = |\psi_A\rangle \otimes |\psi_B\rangle$$

Here, $\otimes$ denotes the Kronecker product of two vectors or matrices [16]. One can describe an n-qubit quantum state in their composed vector space $\mathcal{H}^2 \otimes \mathcal{H}^2$ (e.g. $|01\rangle = |0\rangle \otimes |1\rangle$).

X (Pauli-X)	Y (Pauli-Y)	Z (Pauli-Z)
$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$	$\begin{bmatrix}0 & -i\\i & 0\end{bmatrix}$	$\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$
H (Hadamard)	**T-Gate**	**$\mathbb{I}$ (Identity)**
$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$	$\begin{bmatrix}1 & 0\\0 & e^{i\pi/4}\end{bmatrix}$	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$
SWAP	**CNOT**	**CU (Controlled Gate)**
$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 0 & 1 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$	$\begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & u_{00} & u_{01}\\0 & 0 & u_{10} & u_{11}\end{bmatrix}$

Figure 1. Matrix representation of common quantum gates.

$$|01\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\begin{bmatrix}0\\1\end{bmatrix}\\0\begin{bmatrix}0\\1\end{bmatrix}\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

Quantum Gates. Transformations between pure quantum states are usually described by unitary matrices; as a result, such transformations are reversible and preserve the inner product. When working with qubits, such unitary matrices are often called *quantum gates*, as they allow the manipulation of quantum information units, similar to classical computation. Thus, single-qubit gates are 2×2 unitary matrices, and n-qubit gates are $2^n \times 2^n$ unitaries.

Universal Gates. Quantum computation can be performed universally with a restricted set of single and two-qubit gates [4, 27]. The most common single-qubit and multi-qubit gates are shown in Figure 1 [25]. The **X**, **Y**, **Z** matrices known as Pauli gates (together with the identity matrix $\mathbb{I}$) are a full basis for the transformation on a qubit space. X gate is equivalent to a classical bit-flip operation, Z gate acts as a phase-flip, and Y gate combines the two with a complex phase. H gate, known as Hadamard, creates a superposition of the states in a computational basis with equal weight and transforms between the two bases X and Z. $\mathbb{I}$ denotes the identity matrix in the qubit space and does not change the state of a qubit. We denote a general n-qubit identity matrix as $\mathbb{I}_n$. The T gate is a specific rotation gate, which is particularly important for the universality of quantum computation. One can also define two-qubit gates as not simply combining two single-qubit gates. The most notable example is CNOT, which is a controlled gate. It applies an X gate on a target state if the *control* qubit is in state $|1\rangle$, and applies nothing, i.e. identity if it is $|0\rangle$. Similarly, one can define a controlled gate for any single-qubit unitary **U** as CU. Finally, SWAP is a two-qubit gate that swaps the order of qubits applied to, i.e., $\textbf{SWAP}\,|ij\rangle = |ji\rangle$ for any choice of basis.

Quantum Circuits. A *quantum circuit* is constructed with a combination of gates over n-qubits, often followed by measurement at the end, aiming at performing a computation and extracting the result through measurement. An overall action of a circuit can be described by a $2^n \times 2^n$ size unitary

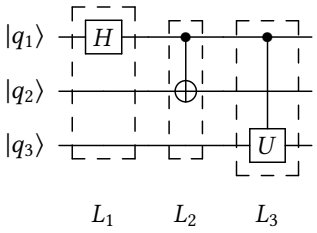

Figure 2. A quantum circuit with 3 gates: a single-qubit gate (L_1: Hadamard gate), an adjacent two-qubit gate (L_2: CNOT gate, not to be confused with direct sum $\oplus$), and a non-adjacent two-qubit gate (L_3: CU gate).

matrix over the space of n-qubits. This matrix can be obtained from the circuit description by matrix multiplication of the gates one after another. The reverse is also possible. General methods exist for decomposing n-qubit unitary matrices into single and two-qubit quantum gates [10, 27]. Figure 2 shows a quantum circuit with single and two-qubit gates.

Note that in the figure, whenever there is a straight line with no gate, one can assume an identity gate is applied. Thus where a single-qubit gate $\mathbf{M}$ is present on qubit i, can be written as $\mathbb{I}_{(1)} \otimes \mathbb{I}_{(2)} \otimes \cdots \otimes \mathbb{I}_{(i-1)} \otimes \mathbf{M}_{(i)} \otimes \cdots \otimes \mathbb{I}_{(n)}$.

We simplify this notation and introduce the following notation for the overall transformation matrix $\mathbf{G}$ of a gate:

$$\mathbf{G} = \mathbb{I}_L \otimes \mathbf{M} \otimes \mathbb{I}_R \tag{1}$$

where $\mathbf{M}$ is a single or two-qubit gate (adjacent), on a position where there are L qubits (with identity applied to them) before and R qubits after. This notation allows us to describe our simulation techniques clearly in what follows. Also note that for the two-qubit gates, this notation is only applicable for adjacent qubits acted by the gate. If the gate acts on two non-adjacent qubits, one can use a SWAP gate to move them to the adjacent position, apply the two-qubit gate, and then move them back (see Section 4.1). The outcome of the circuit, after application of the gate, on an input state vector $|\psi_{in}\rangle$, would be another state vector $|\psi_{out}\rangle = \mathbf{G}\,|\psi_{in}\rangle$:

$$S_{out} \;=\; \mathbf{G} \cdot S_{in} \tag{2}$$

The full circuit matrix, represented by D gates, is:

$$\mathbf{G}_D \mathbf{G}_{D-1} \ldots \mathbf{G}_1 \tag{3}$$

2.2 Quantum Simulators

Considering the properties of the vector spaces and the transformation rules dictated by quantum mechanics, several general methods for quantum simulation[1] have been developed, namely Schrödinger-based, Feynman path integrals, Heisenberg-based, and hybrid approaches [53]. Most simulators, including QuEST [21] and Qiskit [33], fall into the Schrödinger-based category, which simulates quantum states and unitary transformations using vectors and matrices [53].

QuEST [21] is a high-performance simulator for quantum circuits and many-body quantum systems. It is designed to be highly efficient and platform-independent, supporting various backends such as CPUs and GPUs. QuEST includes a comprehensive set of universal and specialised gates, each with multiple variants to optimise performance across different hardware architectures. Figure 3a is an example of the function implemented by QuEST to simulate Hadamard (a single-qubit) gate. However, this approach results in a complex and extensive codebase, making it challenging to maintain and extend the simulator.

Qiskit [33] is an open-source software development kit for working with quantum computers at the level of pulses, circuits, and algorithms. Qiskit's simulator component is based on the Schrödinger method and supports a wide range of quantum operations. It offers a user-friendly interface and a rich ecosystem of tools for quantum algorithm development. However, Qiskit's performance is inferior to hand-tuned low-level quantum simulators (e.g., QuEST).

The Catalyst compiler [19] is implemented using MLIR by adding a quantum-based intermediate representation, referred to as the QIR dialect in MLIR, to handle quantum instructions. This enables a high-level intermediate representation that covers both quantum and classical components of the program, and results in more effective optimization. After optimization, the representation lowers to LLVM + QIR to generate an executable binary file. However, as mentioned earlier, the optimizations are mostly at the circuit level, and the gate implementations are provided in the runtime library.

LFQS is a state-vector simulator that produces the intermediate states of quantum circuits after each gate. This contrasts with tensor network contraction methods, which are often more efficient when only the final measurement outcomes are required. Consequently, our simulator does not employ optimization techniques like gate fusion or other methods typically used in tensor networks at the global level [34]. While the tensor network approach can achieve higher performance for certain types of circuits [53], state-vector simulation is essential for specific quantum algorithms where access to intermediate state vectors or mid-circuit measurements is necessary. Additionally, state-vector simulation is a valuable debugging tool for quantum programmers.

LFQS combines the advantages of the current approaches in quantum simulation; as demonstrated in Section 6, it significantly outperforms library-based solutions (e.g., Qiskit) and resolves the maintainability challenges available in hand-tuned low-level quantum simulators (e.g., QuEST). To achieve this, LFQS uses a compilation-based approach with an intermediate representation to automate the optimization process. However, unlike the existing approaches in this category, LFQS focuses on gate-level optimizations by translating the high-level specification of gates into low-level code.

[1]We use the term quantum simulation to refer to the classical simulation of quantum computing. This term is also used for simulating the behaviour of complex many-body physical systems and predicting their behaviour using quantum computation, which is not the focus of this work.

```
for (thisTask=0; thisTask<numTasks; thisTask++) {
    thisBlock = thisTask / sizeHalfBlock;
    indexUp = thisBlock*sizeBlock + thisTask%sizeHalfBlock;
    indexLo = indexUp + sizeHalfBlock;
    stateRealUp=stateVecReal[indexUp]; stateImagUp=stateVecImag[indexUp];
    stateRealLo=stateVecReal[indexLo]; stateImagLo=stateVecImag[indexLo];
    stateVecReal[indexUp] = recRoot2*(stateRealUp + stateRealLo);
    stateVecImag[indexUp] = recRoot2*(stateImagUp+stateImagLo);
    stateVecReal[indexLo] = recRoot2*(stateRealUp – stateRealLo);
    stateVecImag[indexLo] = recRoot2*(stateImagUp–stateImagLo);
}
```

(a) The implementation of Hadamard gate in QuEST.

```
for (rL= 0; rL<Ls; rL++) {
  for (rR= 0; rR<Rs; rR++) {
    id1 = rR + (rL * 2) * Rs;
    id2 = id1 + Rs;
    sid1real = S[id1].real; sid1imag = S[id1].imag;
    sid2real = S[id2].real; sid2imag = S[id2].imag;
    Out[id1].real = sq2r * (sid1real + sid2real);
    Out[id1].imag = sq2r * (sid1imag + sid2imag);
    Out[id2].real = sq2r * (sid1real – sid2real);
    Out[id2].imag = sq2r * (sid1imag – sid2imag);
}}
```

(b) Generated code for the Hadamard gate in LFQS.

Figure 3. The correspondence between the sequential implementation of the Hadamard gate in QuEST and LFQS.

2.3 SDQL

We use SDQL [41, 42] as the intermediate language of our compiler. SDQL is a high-level language that handles various workloads, including relational query processing [38, 45] and sparse tensor algebra with built-in optimization capabilities.

A key aspect of SDQL's ability to optimize and efficiently translate expressions is its use of semi-ring structures. A semi-ring is an algebraic structure that consists of a set equipped with two binary operations (typically called addition and multiplication) that satisfy certain properties such as associativity, distributivity, and identity elements. These structures are useful in various computational contexts, including formal languages, automata theory, and optimization problems [44]. SDQL supports the following data types:

Scalars. In boolean semi-ring, disjunction ($\vee$) and conjunction ($\wedge$) serve as binary operators, while false and true act as identity elements. Scalars can also be values of type int and real, forming the Integer ($\mathbb{Z}$) and Real ($\mathbb{R}$) semi-ring.

Dictionaries. A dictionary with keys of type K, and values of type V, is represented by the data type `{ K -> V }`. If the value elements with type V form a semi-ring structure, the dictionary also forms a semi-ring structure known as a semi-ring dictionary (SD). In this semi-ring, addition is performed point-wise, meaning the values of elements with the same key are added. Additionally, elements in an SD with 0_V as values can be removed from the dictionary. This makes them appropriate for relations [39, 43] and sparse tensors [36, 40].

Vectors and Matrices. In SDQL, vectors are represented as dictionaries that map indices to element values. Therefore, vectors with elements of type S correspond to SDQL expressions of type `{ int -> S }`. Similarly, matrices can be viewed as dictionaries that map row and column indices to element values, denoted as `{ <int, int> -> S }`. Alternatively, matrices can be considered as dictionaries that map row indices to row values, where each row value is another dictionary (vector) of type S, denoted as `{ int -> { int -> S } }`. We will use the latter form to represent matrices.[2]

[2]The semi-ring properties, such as the associativity and distributivity laws, are only used for scalar operations, not at the matrix multiplication level.

Example 1. Consider the vector V and the matrix **M** defined and presented in SDQL as follows:

V	$\begin{bmatrix} v_0 & v_1 & 0 & v_2 \end{bmatrix}$	`{0 -> v0, 1 -> v1, 3 -> v2}`
M	$\begin{bmatrix} m_0 & 0 & 0 & m_1 \\ 0 & m_2 & 0 & m_3 \end{bmatrix}$	`{0 -> {0 -> m0, 3 -> m1},` `1 -> {1 -> m2, 3 -> m3}}`

The expression $\mathbf{M} \cdot V$ is evaluated to the following expression:

`{ 0 ->` $m_0 * v_0 + m_1 * v_2$`, 1 ->` $m_2 * v_1 + m_3 * v_2$ `}`

This expression is the dictionary representation of the following vector, which is the result of the matrix-vector multiplication: $\begin{bmatrix} m_0 * v_0 + m_1 * v_2 & m_2 * v_1 + m_3 * v_2 \end{bmatrix}$.

Looping. The expression `sum(<k, v> in d)e` represents iteration over the elements of dictionary d. Each key-value pair is bound to k and v, respectively. This iteration starts from an appropriate additive identity element and computes the summation of the result of expression e using an appropriate addition operator; the summation uses the scalar addition operator if e has a scalar type and the SD addition is used for an SD expression.

Example 2. Consider the expression `sum(<k,v> in d)v` where `d` is a dictionary with the values `{ "a"-> 3, "b"-> 5, "c"-> 1 }`. This expression sums the values (3 + 5 + 1) in the dictionary `d`, evaluating to 9. Now, consider the expression `sum(<k,v> in d){k->v+3}`, using the same dictionary `d`. This expression evaluates to `{ "a"->6, "b"->8, "c"->4 }`, which is the result of adding 3 to each value in the dictionary: `{ "a"-> 3+3 }`, `{ "b"-> 5+3 }`, and `{ "c"-> 1+3 }`.

The `sum` in the general form of `sum(<k,v> in d)e` iterates over the given `d`, and uses the addition operation of `e`.

Example 3. Consider the expression `sum(<k,v> in V){k -> {k -> v}}` where the body of the summation is a nested dictionary. This expression converts the vector V as shown in Example 1, into the expression `{0 -> {0 ->` v_0`}} + {1 ->` `{1 ->` v_1`}} + {3 -> {3 ->` v_2`}}`. Evaluating this expression results in the following diagonal matrix: `{0 -> {0 ->` v_0`} , 1 -> {1 ->` v_1`} , 3 -> {3 ->` v_2`}}`.

LFQS Extensions. This paper extends SDQL [42] by:

- Introducing a Complex ($\mathbb{C}$) type for complex numbers.
- Adding nRows to the vector type.
- Adding nRows and nCols to the matrix type.

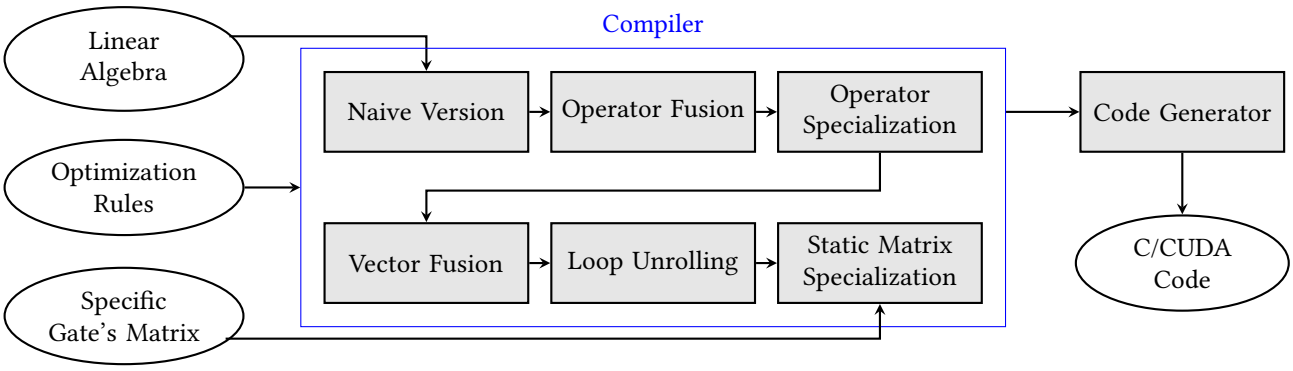

Figure 4. The high-level architecture of LFQS.

- Adding macro definitions denoted by `macro(params)` to make the code more concise (cf. Figure 5).
- Using the following syntactic sugar for the iteration over a nested dictionary to avoid nested `sum`s:

```
sum(<r,VM> in M)                    sum(<<r,c>, eM> in M)
 sum(<c,eM> in VM) →                    f(r, c, eM)
   f(r,c,eM)
```

- Using the tupled let-binding, for convenience, as a syntactic sugar for multiple let bindings as follows:

```
let v1=e1, v2=e2 →    let v1=e1 in let v2=e2
in ...                in ...
```

3 LFQS Architecture

Figure 4 provides the high-level architecture of the Lightweight Functional Quantum Simulator (LFQS). LFQS takes as input the linear algebra specification of a quantum circuit and represents them in SDQL. Our compiler then transforms the initial specifications into more optimized expressions. Subsequently, the code generator translates these expressions from SDQL into efficient parallel low-level C and CUDA code.

3.1 Representing Quantum Circuits

We implement each gate of the target quantum circuit independently, starting with its unoptimized linear algebra representation. To achieve this, we extend the linear algebra operations originally supported by SDQL with additional definitions, as shown in Figure 5. We use Equation (1) to implement a single-qubit gate and an adjacent two-qubit gate. Figure 6 presents the LFQS representation of a circuit with multiple gates, where U represents 2×2 and 4×4 unitary matrices for single-qubit and two-qubit gates, respectively. However, this formulation cannot be applied to non-adjacent qubits; we use the SWAP method (cf. Section 2.1) instead.

3.2 Efficient Simulation of Quantum Circuits

Although the expression in Figure 6 appears straightforward and correctly represents single-qubit and adjacent two-qubit gates, it is inefficient in terms of both memory usage and simulation time. This approach fails to leverage several optimizations, such as exploiting the sparsity, specific shapes and values of the matrices involved and merging operations to minimize redundant memory and time overhead.

Expression	SDQL representation
`eye(I_size)`	`sum(<rI, _> in 0:I_size)` `{ rI -> { rI -> 1 } }`
`mvdot(M,V)`	`sum(<<rM, cM>, eM> in M)` `{ rM -> eM * V(cM) }`

`kron(A,B)`	`sum(<<rA, cA>, eA> in A)`
	`sum(<<rB, cB>, eB> in B)`
	`{ rB+rA*B.nRows ->`
	`{ cB+cA*B.nCols -> eA*eB } }`

`slice(U,i,j)`	`sum(<r, _> in 0:2) sum(<c, _> in 0:2)` `{ r -> { c -> U(r+i*2)(c+j*2) } }`
`onehot(i,j)`	`{ i -> { j -> 1 } }`

Figure 5. Required Expressions to implement a Quantum gate, with their representation in SDQL.

In the next section, we first introduce a vectorized formulation for simulating non-adjacent two-qubit gates (See Section 4.1) that can replace the inefficient SWAP method. Then, we show how we can compile a naive expression like Figure 6 to an efficient expression which can be used to generate a C code similar to Figure 14b (See Section 4.2-4.6).

4 Optimizations

We define several stages of optimization to transform the naive linear algebra into a more optimized expression. At each stage, we utilize an optimization technique and apply some transformation rules to optimize and simplify our expression compared to the previous stage.

Running Example. We use the Hadamard gate as an example of a single-qubit gate to demonstrate the whole transformation procedure of a naive expression like Figure 9a to an optimized expression like Figure 12c. Figure 9 summarizes the transformation procedures for the Hadamard gate. The procedures for other single-qubit gates and even two-qubit ones follow a similar process.

```
let S_1 = /* H Gate elided for brevity */ in
let U = cnotgate() in
let I_L = eye(0), I_R = eye(1) in
let T1 = kron(I_L, U), T2 = kron(T1, I_R) in
let S_2 = mvdot(T2, S_1) in
let S_3 = /* CU Gate elided for brevity */ in
S_3
```

Figure 6. The LFQS representation of the quantum circuit in Figure 2. We only show the code for the CNOT Gate.

(a) Non-adjacent 2-Qubit gate. **(b)** Using SWAP method.

Figure 7. 2-Qubit gate on non-adjacent qubits.

4.1 Vectorized Non-adjacent Two-Qubit Gates

While two-qubit gates over adjacent qubits are easy to formulate, the same notation cannot be applied to the case of non-adjacent qubits. The common approach to address this is to use the swapping method mentioned earlier (See Section 2.1) using several SWAP gates. However, this is a highly inefficient method for simulation as it requires many unnecessary matrix products. Let us consider a 3-qubit circuit as an example to clarify this issue:

Example 3. Consider a 3-qubit circuit with one two-qubit gate (CU) applied to qubits 0 and 2, as shown in Figure 7(a). This circuit is equivalent to Figure 7(b), which uses the SWAP gate. Based on Equation (3), we simulate this by generating an 8×8 matrix using the following transformation:

$$\mathbf{G_{0,2}} = (\mathbb{I} \otimes \mathbf{SWAP_{1,2}})(\mathbf{CU} \otimes \mathbb{I})(\mathbb{I} \otimes \mathbf{SWAP_{1,2}}) \tag{4}$$

As an alternative, we propose a novel method for simulating a non-adjacent two-qubit gate, denoted as below:

$$\mathbf{G_{q_1 q_2}} = \sum_{i=0}^{1}\sum_{j=0}^{1} \mathbf{X_{ij}} \otimes \mathbb{I}_N \otimes \mathbf{U_{ij}} \tag{5}$$

where CU is a two-qubit gate with the following matrix:

$$\mathbf{CU} = \begin{bmatrix} u_{00} & u_{01} & u_{02} & u_{03} \\ u_{10} & u_{11} & u_{12} & u_{13} \\ u_{20} & u_{21} & u_{22} & u_{23} \\ u_{30} & u_{31} & u_{32} & u_{33} \end{bmatrix}$$

applied on qubits q_1 and q_2, N is the number of qubits between the q_1 and q_2 (we denote that as *the gap*), and the matrices $\mathbf{X_{ij}}$ and $\mathbf{U_{ij}}$ are as follows:

$$\mathbf{X_{00}} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \mathbf{U_{00}} = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \quad \mathbf{X_{01}} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad \mathbf{U_{01}} = \begin{bmatrix} u_{02} & u_{03} \\ u_{12} & u_{13} \end{bmatrix}$$

$$\mathbf{X_{10}} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{U_{10}} = \begin{bmatrix} u_{20} & u_{21} \\ u_{30} & u_{31} \end{bmatrix} \quad \mathbf{X_{11}} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{U_{11}} = \begin{bmatrix} u_{22} & u_{23} \\ u_{32} & u_{33} \end{bmatrix}$$

Note that $\mathbf{X_{ij}}$s are constructed from the outer product of different computational bases, i.e., $\mathbf{X_{ij}} = |i\rangle\langle j|$. The idea

```
let I_L=eye(Ls), I_G=eye(Gs), I_R=eye(Rs) in
let T1 = sum(<i,_> in 0:2) sum(<j,_> in 0:2)
  let Xij = onehot(i, j), Uij = slice(U, i, j) in
  let Tij = kron(Xij, I_G) in
let T2 = kron(I_L, T1), T3 = kron(T2, I_R) in
let Out = mvdot(T3, S) in Out
```

Figure 8. Vectorized 2-Qubit gate SDQL expression.

behind this formulation is that a two-qubit gate can be decomposed into 4 blocks, which can be treated as elements of a single-qubit gate when the composition happens through tensor product. Thus, one can write the matrix $\mathbf{G_{0,1}}$ when there are no qubits in between, as follows:

$$|0\rangle\langle 0| \otimes \mathbf{U_{00}} + |0\rangle\langle 1| \otimes \mathbf{U_{01}} + |1\rangle\langle 0| \otimes \mathbf{U_{10}} + |1\rangle\langle 1| \otimes \mathbf{U_{11}} = \sum_{i=0}^{1}\sum_{j=0}^{1} \mathbf{X_{ij}} \otimes \mathbf{U_{ij}}$$

If there is a qubit in between to which the gate does not apply, the resulting matrix should act trivially on that subspace. This means one can expand the representation by applying the identity matrix to the space between the qubits through the tensor product. This is similar to how one formulates the two-qubit controlled gates, where conditioning on the control qubit is in the zero state, the gate applied to the subspace is identity and the full matrix can be represented as the linear combination of the two tensor products.

Following above, we rewrite the previous example as Equation 5, which leads to the following form for $\mathbf{G_{0,2}}$:

$$\mathbf{G_{0,2}} = \begin{bmatrix} u_{00} & u_{01} & 0 & 0 & u_{02} & u_{03} & 0 & 0 \\ u_{10} & u_{11} & 0 & 0 & u_{12} & u_{13} & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} & 0 & 0 & u_{02} & u_{03} \\ 0 & 0 & u_{10} & u_{11} & 0 & 0 & u_{12} & u_{13} \\ u_{20} & u_{21} & 0 & 0 & u_{22} & u_{23} & 0 & 0 \\ u_{30} & u_{31} & 0 & 0 & u_{32} & u_{33} & 0 & 0 \\ 0 & 0 & u_{20} & u_{21} & 0 & 0 & u_{22} & u_{23} \\ 0 & 0 & u_{30} & u_{31} & 0 & 0 & u_{32} & u_{33} \end{bmatrix}$$

We note that this reformulation is equivalent to using $2 \times N$ SWAP gates, where N is the number of qubits between the two target qubits, to bring the qubits to the neighbouring position, apply the adjacent two-qubit gate, and then swap them back to the original position. Figure 8 shows the vectorized representation of a two-qubit circuit.

For the rest of this section, we focus on the optimizations applicable to single- and two-qubit gates. However, for the sake of brevity, we show the application to the single-qubit.

4.2 Operator Fusion

At this stage, we employ the operator fusion technique to merge the required operations, producing the overall transformation matrix. The Kronecker product of three or more matrices requires calculating and storing several intermediate matrices. LFQS merges these steps into a single operation.

Figure 10 shows how we deploy the `kron(A,B)` ($\mathbf{A} \otimes \mathbf{B}$) expression from Figure 5 to produce the Kronecker product

```
let U = hadmat() in
let I_L = eye(Ls) in
let I_R = eye(Rs) in
let T1 = kron(I_L, U) in
let T2 = kron(T1, I_R) in
let Out = mvdot(T2, S) in Out
```

(a) Unoptimized SDQL representation.

```
let U = hadmat() in
let I_L = eye(Ls) in
let I_R = eye(Rs) in
let T2 = kron3(I_L,U,I_R) in
let Out=mvdot(T2,S) in Out
```

(b) After operator fusion.

```
let U = hadmat() in
let T2 = kroneye3(Ls, U, Rs) in
let Out = mvdot(T2, S) in Out
```

(c) After operator specialization.

```
let U = hadmat() in
let Out =
  kroneye3dot(Ls,U,Rs,S) in
Out
```

(d) After vector fusion.

```
let U = hadmat() in
let Out =
 single_qbit(Ls,U,Rs,S) in
Out
```

(e) After loop unrolling.

```
let U = hadmat() in
let Out =
  hadgate(Ls, Rs, S) in
Out
```

(f) After static matrix specialization.

Figure 9. Optimization steps applied to the Hadamard gate.

```
let D = // kron(A,B)
 sum(<<rA,cA>,eA> in A)
  sum(<<rB,cB>,eB> in B)
  {rB+rA*B.nRows ->
   {cB+cA*B.nCols->eA*eB
  } } in // kron(D,C)
sum(<<rD,cD>,eD> in D)
 sum(<<rC,cC>,eC> in C)
 {rC+rD*C.nRows ->
  {cC+cD*C.nCols->eD*eC}}
```

```
//kron3(A,B,C)
sum(<<rA,cA>, eA> in A)
 sum(<<rB,cB>, eB> in B)
  sum(<<rC,cC>, eC> in C)
   let rD=rB+rA*B.nRows,
       cD=cB+cA*B.nCols,
       eD=eA*eB in
   { rC+rD*C.nRows ->
    { cC+cD*C.nCols ->
      eD*eC } }
```

Figure 10. Operator fusion translates the Kronecker product of three matrices expressed by a chain of `kron` into `kron3`.

of 3 matrices ($\mathbf{A}\otimes\mathbf{B}\otimes\mathbf{C}$), named `kron3(A,B,C)`. Thus, instead of calculating $\mathbf{A}\otimes\mathbf{B}\otimes\mathbf{C}$ in two steps, i.e., (1) $\mathbf{D}=\mathbf{A}\otimes\mathbf{B}$ and (2) $\mathbf{D}\otimes\mathbf{C}$, the whole process can be done in one step, saving the time and memory required for calculating $\mathbf{D}$.

Employing this technique to the naïve simulation shown in Figure 6 will merge the two steps of calculations of T1 and T2 into one, eliminating the overhead associated with computing T1 separately.

4.3 Operator Specialization

Generated matrices are very sparse and if not considered, will result in unnecessary calculations over zero-value elements. Since identity matrices significantly influence the shape and sparsity of the resulting transformation matrix, we merge the identity matrix in the Kronecker product calculation to create a more optimized version of the overall transformation matrix. This also improves memory consumption as identity matrices are no longer required to produce and store.

Figure 11b demonstrates how the `eye(n)` ($\mathbb{I}_n$) expression from Figure 5 is used to transform the Kronecker product of 3 arbitrary matrices `kron3(A,B,C)` ($\mathbf{A}\otimes\mathbf{B}\otimes\mathbf{C}$) into `kroneye3(Ls,U,Rs)`. This method allows us to bypass the creation of $\mathbb{I}_L$ and $\mathbb{I}_R$ and the subsequent calculation of $\mathbb{I}_L\otimes\mathbf{U}\otimes\mathbb{I}_R$, which involves numerous unnecessary computations with zero. Instead, we perform only the required calculations for non-zero elements, significantly reducing the overhead.

4.4 Vector Fusion

Typically, updating the state vector is a two-step procedure. The first step involves computing a very large transformation matrix ($\mathbb{I}_L\otimes\mathbf{U}\otimes\mathbb{I}_R$). The second step immediately multiplies this large matrix by the state vector to obtain the new state vector. Given the size of the state vector (2^n) and the transformation matrix ($2^n\times 2^n$), this approach requires a significant amount of memory to store the intermediate matrix and a long computation time.

At this stage, we apply the vector fusion technique to merge the overall transformation matrix calculation with the matrix-to-vector multiplication, creating a single expression to compute the new state vector in one step. This technique significantly reduces the required time and memory by eliminating the overhead of separately computing the overall transformation matrix.

Figure 11c illustrates how we deploy the `mvdot(M,V)` ($\mathbf{M}\cdot V$) expression from Figure 5 together with `kroneye3(Ls,U,Rs)` to produce `kroneye3dot(Ls,U,Rs,S)`. This allows us to calculate the new state vector in a single step instead of two.

4.5 Loop Unrolling

The application of the vector fusing technique resulted in the removal of one loop from the expression by fusing the outer loop for creating the intermediate matrix with the one for multiplying the vector. The final expression in Section 4.4 features two loops with a fixed number of iterations over the rows and columns of the gate matrix. These iterations are independent of the type of gate, the number of qubits in the circuit, or the index of the target qubit. Here, we use the loop unrolling technique (similar to [17]) to turn loops with fixed iteration sizes into a sequence of statements, enhancing performance by eliminating the time overhead associated with the nested sum required to loop over the gate matrix.

```
let L = // eye(Ls)
  sum(<rI,_> in 0:Ls) {rI->{rI->1}} in
let R = // eye(Rs)
  sum(<rI,_> in 0:Rs) {rI->{rI->1}} in
let M = // kron3(L,U,R)
  sum(<<rL,cL>, eL> in L)
   sum(<<rU,cU>, eU> in U)
    sum(<<rR,cR>, eR> in R)
     let rD=rU+rL*U.nRows in
     let cD=cU+cL*U.nCols, eD=eL*eU in
     {rR+rD*R.nRows->{cR+cD*R.nCols ->
        eD*eR } } in // mvdot(M,S)
sum(<<rM, cM>, eM> in M)
   { rM -> eM * S(cM) }
```

(a) Before applying operator specialization.

```
let M =
//kroneye3(Ls,U,Rs)
 sum(<rL,_> in 0:Ls)
  sum(<<rU,cU>, eU> in U)
   sum(<rR,_> in 0:Rs)
    let rD=rU+rL*U.nRows in
    let cD=cU+rL*U.nCols in
    {rR+rD*Rs->{rR+cD*Rs ->
     eU}} in // mvdot(M,S)
sum(<<rM, cM>, eM> in M)
   { rM -> eM * S(cM) }
```

(b) After operator specialization.

```
//kroneye3dot(Ls,U,Rs,S)
sum(<rL,_> in 0:Ls)
 sum(<<rU,cU>, eU> in U)
  sum(<rR,_> in 0:Rs)
   let rD=
     rU+rL*U.nRows in
   let cD=
     cU+rL*U.nCols in
   let rM=rR+rD*Rs in
   let cM=rR+cD*Rs in
   { rM -> eU*S(cM) }
```

(c) After applying vector fusion.

Figure 11. Applying operator specialization to translate the Kronecker product of three arbitrary matrices expressed by **kron3** into **kroneye3** and vector fusion to translate the matrix to the vector product expressed by **kroneye3** and **mvdot** into **kroneye3dot**.

Figure 12b demonstrates the transformation of **kroneye3dot** into **single_qbit**, focusing on a single-qubit gate matrix **U**, which transforms expressions in form of

```
let U = {0->{0->u00,1->u01},1->{0->u10,1->u11}} in
sum(<<rU,cU>, eU> in U) f(rU, cU, eU)
```

into a sequence of statements in the form of

```
f(0,0,u00)+ f(0,1,u01)+ f(1,0,u10)+ f(1,1,u11)
```

4.6 Static Matrix Specialization

At this stage, we further simplify the gate expressions and specialize expressions for each gate type (e.g., Hadamard, X, T-gate, etc.) utilizing the known matrices values for these gates (refer to Figure 1). These specific expressions are then incorporated into the general expressions for one and two-qubit gates. Depending on the shape or sparsity of the matrix, this approach can capture the additional sparsity in the system (e.g., CNOT and SWAP gates) and eliminate time overhead associated with unnecessary operations such as duplicated calculations (e.g., Hadamard and Pauli gates).

Figure 12c shows how we convert the generic **single_qbit** expression into the gate-specific expression **hadgate**.

5 Implementation

For the quantum simulator, we utilize Python, C, and CUDA; Python is employed for developing the naïve version, while C and CUDA are utilized for the optimized version. Additionally, Python executes the Qiskit and Catalyst simulators.

5.1 Compiler

We use Scala to implement our compiler, which serves optimization tasks and includes a code generator translating SDQL expressions into C and CUDA code.

Frontend. Our compiler takes inputs as SDQL source code. We encode different gates using their naïve representation in SDQL. Then, the source code for these expressions is passed to a parser that creates the SDQL IR.

Transformation. At each stage of optimization, we inline the definition of macros to update the required symbols with new expressions, then main rewrite rules, including the loop transformations (cf. Figure 13) are applied until a fix-point is reached. Finally, generic optimizations (e.g., CSE and algebraic simplifications) are applied.

Code Generator. The code generator translates SDQL expressions into C and CUDA code, enabling us to produce the quantum simulator with any desired level of optimization.

Generating code for SDQL is generally simple due to the first-order nature of most of its constructs; SDQL's design avoids the complexities of compiling polymorphic higher-order functional languages. The main challenge is the summation construct, which we convert into for-loops. We only need to define the backend-specific code generation rules to generate code for other backends, as demonstrated next.

5.2 Parallelization

The code generation process for the parallel version is very similar to that of the sequential version. In the parallel version, the code generator adds the necessary commands for multi-threaded execution of the loops. Figure 14 shows the optimized expression (Figure 12c partially translated to A-Normal Form [11]) for the Hadamard gate alongside the equivalent generated parallelized C and CUDA code. As shown in Figure 14b, the code generator creates a list of shared and private variables used inside the loops (lines 3,4). The number of nested loops is also counted to generate the collapse clause which specifies how many loops should be merged into a single loop (line 5). This maximizes workload distribution and improves the efficiency of parallel execution.

```
let sq2r = 0.7071,
 u00=sq2r,u01=sq2r,u10=sq2r,u11=-sq2r,
 U = {0->{0->u00,1->u01},
      1->{0->u10,1->u11}} in
//kroneye3dot(Ls,U,Rs,S)
sum(<rL,_> in 0:Ls)
 sum(<<rU,cU>, eU> in U)
  sum(<rR,_> in 0:Rs)
   let rD=rU+rL*U.nRows in
   let cD=cU+rL*U.nCols in
   { rR+rD*Rs -> eU*S(rR+cD*Rs) }
```

(a) Before applying loop unrolling.

```
let sq2r = 0.7071,
  u00=sq2r, u01=sq2r,
  u10=sq2r, u11=-sq2r in
//single_qbit(Ls,U,Rs,S)
sum(<rL,_> in 0:Ls)
 sum(<rR,_> in 0:Rs)
  let id1=rR+(rL*2)*Rs in
  let id2=id1 + Rs in
  {id1->u00*S(id1)+u01*S(id2)}
  +
  {id2->u10*S(id1)+u11*S(id2)}
```

(b) After loop unrolling.

```
let sq2r = 0.7071 in


//hadgate(Ls,Rs,S)
sum(<rL,_> in 0:Ls)
 sum(<rR,_> in 0:Rs)
  let id1=rR+(rL*2)*Rs in
  let id2=id1 + Rs in
  {id1->sq2r*(S(id1)+S(id2))}
  +
  {id2->sq2r*(S(id1)-S(id2))}
```

(c) Most optimized version.

Figure 12. Applying loop unrolling to translate into `single_qbit`, and static matrix specialization to translate into `hadgate`.

`sum(<k, v> in {r1->e1}) e2`	`let k=r1, v=e1 in e2`
`sum(<k1,v1> in sum(<k2,v2> in e2) e1) e3`	`sum(<k2,v2> in e2) sum(<k1,v1> in e1) e3`
`sum(<k,v> in e1) e2 + sum(<k,v> in e1) e3`	`sum(<k,v> in e1) e2+e3`
`sum(<k,v> in {r1->e1,r2->e2}) e3`	`let k=r1, v=e1 in e3 + let k=r2, v=e2 in e3`

Figure 13. Selected loop transformations in our optimizer.

Additionally, as the input state vector (S) is not needed later, we can use an in-place update to avoid creating a new state vector (Out) by reusing the existing state vector (S). This approach improves efficiency by reducing the memory overhead of managing an additional state vector. Also, it improves memory locality, helping to keep data close to the processing cores. In-place updates allow the state vector to remain in the cache, reducing the need for costly memory transfers between different levels of the memory hierarchy (e.g., from RAM to cache). To enable this, a list of temporary variables is generated to store the necessary elements of the input vector (lines 10,11), and these variables are then used to update the input vector directly with the output of the calculations (lines 12-15). Finally, the in-place update can also eliminate unnecessary calculations when the new values are the same as the old ones (e.g., in CNOT or SWAP gates).

GPU code generation follows a similar approach. The key difference here is that a separate kernel function is used to run across many threads in parallel instead of loops in the main function. Figure 14c shows the generated kernel function; the main function assigns values (lines 1,2) and invokes the kernel function (omitted here). If there are multiple nested loops, we apply the loop-flattening technique and merge them into a single loop, where the total number of iterations is calculated by multiplying the iterations of each individual loop (line 2). A combined iterator is used for the flattened loop (line 4) and the loop's termination condition is replaced by an `if` statement (line 5). Each thread computes the original loop counters based on the value of the combined iterator (lines 6,7).

5.3 Runtime

The runtime environment for LFQS is lightweight. The key challenge is to handle vectors and matrices of complex values, for which we consider two different data layouts:

Array of Structs (AOS). An array of complex numbers represents the state vector, and a 2-dimensional array of structs (array of arrays of complex numbers) represents quantum gate matrices.

Struct of Arrays (SOA). A struct of two arrays (of float numbers) represents the real and imaginary parts of the state vector and two 2-dimensional arrays (arrays of arrays of float numbers) represent quantum gate matrices.

6 Experiments

We evaluate the performance of our generated quantum simulator against state-of-the-art quantum simulators on eight real-world quantum circuits (Section 6.2). Then, we study the impact of the number of qubits using a GSC circuit. The unoptimized version is compared with a Python implementation using NumPy functions for the same linear algebra operations, while the LFQS (optimized version) is compared with Qiskit, Catalyst, and QuEST (Section 6.3). We compare the scalability of LFQS with QuEST, using the RQC circuit (Section 6.4). We examine the impact of each optimization level on a circuit with a single H or CNOT gate (Section 6.5). Additionally, we study the performance of each data layout using RQC circuit (Section 6.6). Finally, we evaluate the performance of LFQS against QuEST on eight real-world quantum circuits over GPU (Section 6.7). Furthermore, we showcase the maintainability of LFQS compared to QuEST in three scenarios: adding a new gate, modifying the data layout, and adding a GPU backend (Section 6.8). We use the Array of Structs data layout in all experiments, apart from Section 6.6, where we study the impact of data layouts.

6.1 Experimental Setup

Our experiments are conducted on a system featuring an Intel Xeon Silver 12-Core Processor operating at 2.2GHz, with 250GB of DDR3 RAM, running Ubuntu 22.04.4 OS. We

```
let sq2r = 0.7071 in


let Out =
 sum(<rL,_> in 0:Ls)
  sum(<rR,_> in 0:Rs)
   let id1=rR+(rL*2)*Rs in
   let id2=id1 + Rs in
   let s1 = S(id1) in
   let s2 = S(id2) in
   { id1 -> sq2r*(s1 + s2) }
   +
   { id2 -> sq2r*(s1 - s2) }
   in
Out
```

(a) Optimized SDQL expression.

```
sq2r = 0.7071;
#pragma omp parallel default (none) \
  shared (Rs, Ls, sq2r, S) \
  private (rL, rR, s1r, s1i, s2r, s2i, id2, id1) {
#pragma omp for schedule(static) collapse(2)
for (rL= 0; rL<Ls; rL++) {
  for (rR= 0; rR<Rs; rR++) {
    id1 = rR + (rL * 2) * Rs;
    id2 = id1 + Rs;
    s1r = S[id1].real; s1i = S[id1].imag;
    s2r = S[id2].real; s2i = S[id2].imag;
    S[id1].real = sq2r * (s1r + s2r);
    S[id1].imag = sq2r * (s1i + s2i);
    S[id2].real = sq2r * (s1r - s2r);
    S[id2].imag = sq2r * (s1i - s2i);
}}}
```

(b) Generated C code.

```
sq2r = 0.7071; //Main Function
iters = Ls*Rs; //Main Function
//CUDA Kernel Function
rI = blockIdx.x*blockDim.x + threadIdx.x;
if (rI < iters) {
    rL = rI / Rs;
    rR = rI % Rs;
    id1 = rR + (rL * 2) * Rs;
    id2 = id1 + Rs;
    s1r = S[id1].real; s1i = S[id1].imag;
    s2r = S[id2].real; s2i = S[id2].imag;
    S[id1].real = sq2r * (s1r + s2r);
    S[id1].imag = sq2r * (s1i + s2i);
    S[id2].real = sq2r * (s1r - s2r);
    S[id2].imag = sq2r * (s1i - s2i);
}
```

(c) Generated CUDA code.

Figure 14. Translation of the most optimized expression into parallelized C and CUDA code.

Table 1. Quantum circuits used in our evaluation with the number of qubits in each section.

Circuit	Description	Section	Qubits
Google's Sycamore quantum supremacy Circuit (GSC) [3]	A class of random quantum circuits used to demonstrate quantum advantage for a sampling problem.	6.2, 6.7	26
		6.3	2-14
Random Quantum Circuit (RQC) [9] (a.k.a. Brickwork circuits)	Specified by a fixed repeatable structure that is used to generate random and pseudorandom quantum objects.	6.2, 6.4, 6.7	26
		6.6	2-20
Quantum Phase Estimation (QPE) [22]	Used both independently and as a subroutine in several quantum algorithms including estimating eigenvalues of matrices or period finding.	6.2, 6.7	26
Quantum Fourier Transform (QFT) [35]	A linear transformation on quantum bits, which is an important subroutine for many quantum algorithms, including the QPE and Shor's algorithm.	6.2, 6.7	26
Variational Quantum Eigensolver algorithm (VQE) [50]	A hybrid quantum-classical algorithm commonly used for solving quantum chemistry problems and finding ground state energies of molecules.	6.2, 6.7	26
Greenberger-Horne-Zeilinger states (GHZ) [27]	Used to create multi-qubit quantum entangled states involving three or more particles, where the particles are maximally entangled.	6.2, 6.7	26
Grover's algorithm [15] (a.k.a. quantum search algorithm)	Used for unstructured search to find the input to a black box function with a particular output value.	6.2, 6.7	26
Shor's algorithm [46]	Used for finding the prime factors of an integer.	6.2, 6.7	26

compile the generated C code using gcc 11.4.0 with the O2 optimization flag. For GPU experiments, we use an NVIDIA GeForce RTX 3070 and compile the code using nvcc 12.0. Our competitors include QuEST, which utilizes the same system and compiling flags, as well as Catalyst v0.8, and Qiskit, which operate on Python 3.10.12 with NumPy 1.26.4. We observed similar behaviour with other dense tensor frameworks, thus, we only include the results for NumPy as a representative of them. In Section 6.7, we present the results of experiments conducted on a GPU, while all other experiments are executed on the CPU. Section 6.4 uses varying numbers of threads, Section 6.5 uses a single thread, and all other experiments use 48 threads.

In all cases, we measure the execution time for the circuit simulation and report the average of five runs. We limit the number of qubits for the naïve versions to restrict the memory usage and execution time. We consider two workloads:

Real-World Circuits. We evaluate the performance of our simulator on eight real-world quantum circuits with varying numbers of qubits. Table 1 shows the quantum circuits used in our evaluation with the number of qubits used for them in each section.

Individual Gates. We generate a circuit with a single H gate on the 2nd qubit, and the other one with a CNOT gate where the 1st qubit is the control and the 2nd qubit is the target. We investigate the performance of our simulator by varying numbers of qubits.

6.2 End to End Performance Evaluation

We compare the performance of the LFQS with QuEST, Catalyst, and Qiskit on different quantum circuits. We have set the number of qubits to 26 in this study. We have set a maximum of 2 hours simulation limit, which has terminated the simulation of the Grover and Shor circuits in Qiskit.

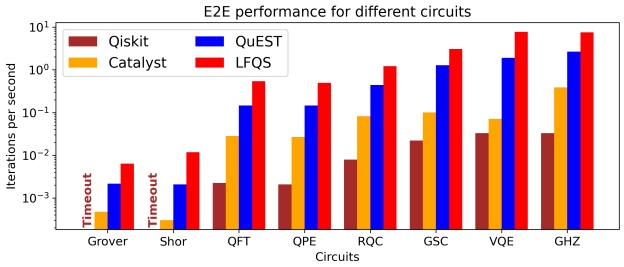

Figure 15. Performance of LFQS for multi-core CPU.

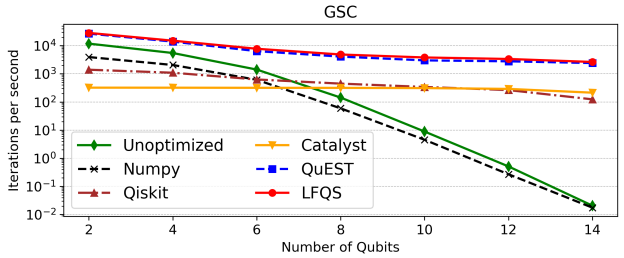

Figure 16. Impact of the number of qubits on performance.

Figure 15 presents the results of this study. As shown, on average LFQS performs 3.45× faster than QuEST, 32.71× faster than Catalyst, and 205.56× faster than Qiskit.

6.3 Impact of Number of Qubits

Figure 16 shows the performance of LFQS on GSC circuits with varying numbers of qubits. As the number of qubits increases, the performance of the simulator decreases. The unoptimized (C version) outperforms the NumPy version for circuits with fewer qubits. This performance difference diminishes as the number of qubits increases, becoming minimal beyond eight qubits.

Similarly, Qiskit and Catalyst exhibit lower performance for circuits with fewer qubits. Their performance are less than that of the naive C and NumPy versions for circuits with less than 6 qubits. However, beyond this point, they perform better than both, though they still lag behind QuEST. The optimized LFQS consistently outperforms QuEST, being 1.16× faster on average.

6.4 Scalability Experiment

We have used an RQC circuit with 26 qubits to study the scalability performance of the LFQS using different numbers of threads for execution. As shown in Figure 17 both systems have similar trends while LFQS performs 5.81× faster than QuEST on average.

6.5 Impact of Individual Optimization

As shown in Figure 18, each optimization stage slightly improves performance. Table 2 shows the average speed-up values for each optimization (compared to the previous stage). The most substantial improvement occurs with the application of Vector Fusion, which significantly enhances performance. This indicates that merging the calculation of the

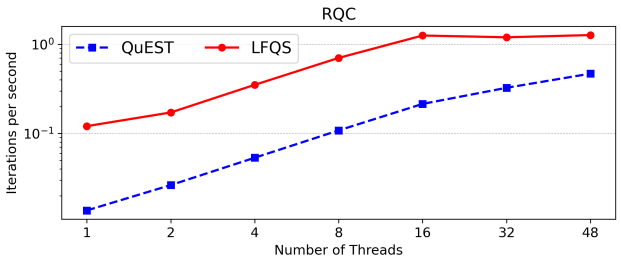

Figure 17. Scalability experiment using RQC circuit.

Table 2. Average speed-up values for each individual optimization, compared with the previous stage (cf. Figure 18).

Optimization	H gate	CNOT gate
Operator Fusion	1.2	1.48
Operator Specialization	2.09	2.49
Vector Fusion	1557.45	684.13
Loop Unrolling	2.62	2.65
Static Matrix Specialization	2.09	6.02

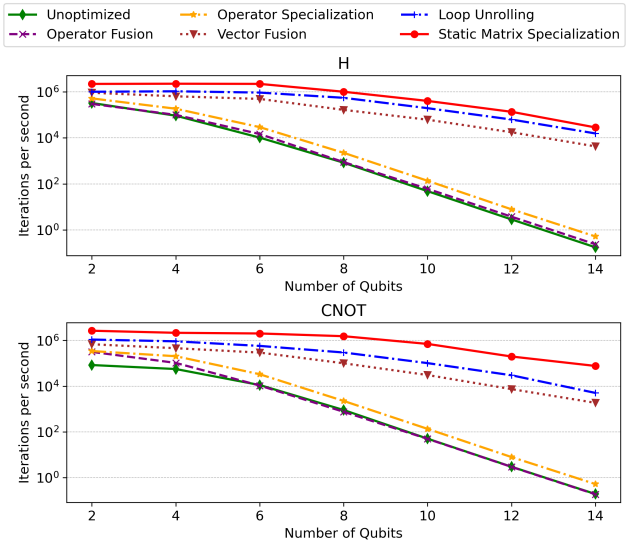

Figure 18. Impact of each optimization on performance.

overall transformation matrix with the matrix-to-vector multiplication into a single step has the most significant impact. This optimization not only improves execution time but also drastically reduces memory consumption by eliminating the need to create temporary complex matrices of size $2^n \times 2^n$.

6.6 Impact of Data Layout

We have generated two different versions of C code using different data layouts to store the state vectors and matrices, both derived from the same optimized SDQL expression. Both data layouts exhibit reasonable performance on RQC

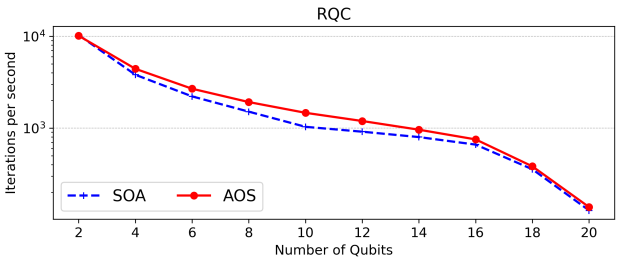

Figure 19. Impact of the data layout on performance.

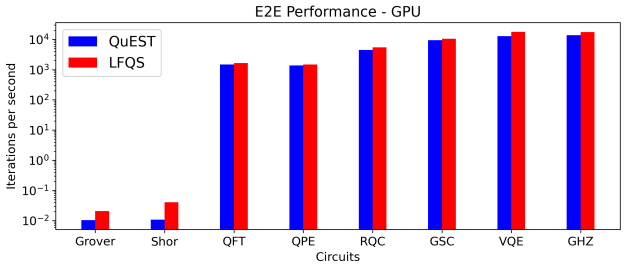

Figure 20. GPU performance for LFQS and QuEST.

circuits. However, as shown in Figure 19, AOS performs on average 1.19× faster than SOA.

6.7 GPU Performance Evaluation

We generated a CUDA version from optimized SDQL expressions and compared the performance of LFQS with QuEST on various quantum circuits running on the GPU. For this study, we set the number of qubits to 26. The results, shown in Figure 20, indicate that LFQS performs, on average, 1.6× faster than QuEST.

6.8 Maintainability

We investigate how the LFQS architecture influences the maintainability of the quantum simulator. Table 3 shows the number of lines of code (LoC) required to change in order to add a new gate or to change the data layout for a multi-core CPU backend, as well as supporting a new backend in LFQS compared to the same scenario in QuEST. We report the average and total number of LoC for QuEST based on its source code.

Adding a new gate. On average, QuEST requires manually implementing 46 LoC to add a new single-qubit gate and 48 LoC for a new two-qubit gate. This can be generated in LFQS without any changes in the code and only by using the gate matrix for the desired gate during "Static Matrix Specialization" with a single LoC.

Changing data layout. QuEST requires manually hand-tuning an average of 10 LoC per function to change the data layout. Assuming the available native/general single-qubit and two-qubit gates and just in the CPU version, 128 LoC must be modified. This can be easily addressed in LFQS by only modifying 5 LoC in our code generator; we need to change the code generator rules for accessing the element of a vector/matrix.

Supporting a new backend. Adding a new backend in QuEST requires manually re-implementing the code for all gates. This results in 542 LoC for the available native/general single-qubit and two-qubit gates to support the GPU. This can be handled in the LFQS by modification of 90 LoC in the code generator.

Table 3. Required LoC changes in LFQS and QuEST.

Scenario	LFQS	QuEST
Adding a new single-qubit gate (CPU)	1	46
Adding a new two-qubit gate (CPU)	1	48
Changing data layout (CPU)	5	128
Supporting a new backend (GPU)	90	542

7 More Related Work

This section presents the related work from two dimensions. First, we cover different approaches for quantum simulation. Then, we cover different tensor compilers.

7.1 Quantum Simulators

There are several approaches for simulating quantum circuits on classical computers, each with its own advantages and limitations. Here, we present the main approaches; however, we refer to the comprehensive surveys [53] for more information.

Stabilizer Circuits. For quantum circuits consisting only of Clifford gates (which map Pauli operators to Pauli operators under conjugation) and computational basis state preparations/measurements, there exist efficient classical simulation algorithms based on the stabilizer formalism. These algorithms have polynomial complexity in the number of qubits and gates, enabling efficient simulation even for large circuits within this restricted gate set. Simulators like Qiskit Aer [33] and CHP [1] implement this approach.

Tensor Network Contraction. This approach represents the quantum state as a tensor network and simulates the circuit by contracting the network. It can achieve significant computational speedups over naive state vector simulations in certain cases, especially for circuits with limited entanglement [37, 52]. The key idea is to exploit the low-rank structure of the quantum state by approximating it with a tree tensor network. This reduces the memory requirements and computational cost compared to storing the full state vector. Examples of simulators using this approach include Qiskit Aer [33], Perceval [18], and Yao [24].

General Purpose Simulators. These simulators can simulate arbitrary quantum circuits, albeit with varying performance and scalability. Examples include Intel Quantum Simulator, Qrack, and QuEST. LFQS falls into the last category, it is designed for general-purpose simulation, handling a broad range of quantum gates, thus providing greater versatility.

LFQS does not utilize tensor network contraction as it aims to provide a general-purpose simulation approach that can handle a wide variety of circuits without relying on specific low-rank approximations. Furthermore, tensor network contraction does not expose intermediate states which is essential for debugging in quantum simulation. LFQS provides a robust approach for efficiently simulating quantum circuits. The framework's modular design provides better maintainability for LFQS compared to its competitors like QuEST and allows us to capture new algorithms and extend to new backends much more easily than the other competitors.

MLIR-QIR Based Simulations. In recent years, significant efforts have been made to utilize compilation frameworks, in particular LLVM and MLIR, for quantum computing IRs. The Quantum Intermediate Representation (QIR) [12] provides a language- and hardware-agnostic IR for integrating classical and quantum programs, utilizing LLVM's compiler infrastructure. QIRO [20] and QSSA [30] have expanded on this by defining specialized dialects within MLIR for quantum-classical co-optimization and SSA-based quantum IRs, respectively. McCaskey and Nguyen [26] also introduced a new MLIR dialect that translates quantum assembly languages to executable binaries using QIR.

These studies mainly focus on compiling high-level quantum programs into the LLVM-based QIR and applying global optimizations at the circuit level. The gate-level optimizations are handled manually in the runtime implementation of the gates, which need to be redone for different choices of data layout or new gates. Instead, LFQS concentrates on individual gates and automates gate-level optimization through compiler optimizations, reducing the engineering effort needed for building the runtime implementation of gates. Global circuit optimizations are left as future work.

7.2 Tensor Compilers

Dense Tensor Frameworks. Inspired by the recent advances in AI and ML, there has been significant progress in tensor compilers. Tensor frameworks such as TensorFlow [2], PyTorch [29], and TVM [6] optimize computations for dense data without considering any inherent structure, leading to unnecessary computations for structured data. Similarly, polyhedral-based frameworks [51] cannot fully leverage such inherent structures.

Sparse Tensor Compilers. Tensor frameworks such as TACO [23], SPF [49], and SPLATT [47] handle sparse tensors by performing computations only over non-zero elements. However, these frameworks manage sparsity at runtime, resulting in irregular memory access patterns that are not cache-friendly and are hard to optimize.

Structured Tensor Compilers. LGen [48] captures specific structural patterns like symmetry or fixed-size constraints at compile time. However, it is limited to small-scale matrices and does not support the flexible, variable-sized structures often encountered in real-world applications. StructTensor [14] and DASTAC [13] aim to resolve these limitations. However, they fail to capture the static structures with irregular shapes, e.g., SWAP gate (cf. Figure 1).

Quantum simulation involves sparse matrices with repeating structures, exemplified by the identity matrix in the Kronecker product, which introduces a structured sparsity. Existing frameworks fail to efficiently handle this scenario because they (1) overlook the structural redundancy (dense/sparse frameworks), (2) are constrained to fixed sizes (LGen [48]), or (3) cannot capture the quantum-specific structure (StructTensor [14]). LFQS addresses these limitations by employing an algebraic intermediate language that exploits the quantum-specific structure in compilation time.

8 Conclusion and Outlook

This paper presented LFQS, a new approach for efficiently simulating quantum circuits. Our compilation stack starts with unoptimized, high-level linear algebra descriptions of the quantum gates and circuits. It then applies a series of optimization techniques including operator fusion, operator specialization, vector fusion, loop unrolling, and static matrix specialization, using an intermediate language to produce an efficient representation. At the end, this representation is translated into low-level C and CUDA code using code generation. Through several experiments on various types of circuits, we demonstrated that LFQS consistently outperforms existing simulators like QuEST, Catalyst, and Qiskit, particularly in circuits involving a higher number of qubits.

The framework's modular design provides better maintainability for LFQS compared to its competitors. For example, we easily changed the data layout in our simulator, where the AOS layout slightly outperforms the SOA layout. This modularity allows us to capture new algorithms and extend to new backends more easily than competitors. In the future, we plan to explore alternative backends, including QPUs. Furthermore, we plan to capture the SIMD vectorization as a transformation on the intermediate language.

Acknowledgement

This work was partly funded by a gift from RelationalAI and the Engineering and Physical Sciences Research Council (EPSRC) on Robust and Reliable Quantum Computing (RoaRQ) grant EP/W032635/1. Doosti is supported by the EPSRC on Quantum Advantage Pathfinder (QAP) with grant reference EP/X026167/1, the Physical Sciences Research Council (EPSRC). Delavar acknowledges EPSRC grant EP/T014784/1.

References

[1] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Physical Review A* 70, 5 (2004), 052328.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[3] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[4] Sergey Bravyi and Alexei Kitaev. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A* 71, 2 (2005), 022316.

[5] Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. 2019. Quantum chemistry in the age of quantum computing. *Chemical reviews* 119, 19 (2019), 10856–10915.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[7] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, et al. 2022. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–50.

[8] Alexander M Dalzell, Sam McArdle, Mario Berta, Przemyslaw Bienias, Chi-Fang Chen, András Gilyén, Connor T Hann, Michael J Kastoryano, Emil T Khabiboulline, Aleksander Kubica, et al. 2023. Quantum algorithms: A survey of applications and end-to-end complexities. *arXiv preprint arXiv:2310.03011* (2023).

[9] Christoph Dankert, Richard Cleve, Joseph Emerson, and Etera Livine. 2009. Exact and approximate unitary 2-designs and their application to fidelity estimation. *Physical Review A* 80, 1 (2009), 012304.

[10] David P DiVincenzo. 1995. Two-bit gates are universal for quantum computation. *Physical Review A* 51, 2 (1995), 1015.

[11] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

[12] Alan Geller. 2020. Introducing quantum intermediate representation (QIR). *Q# Blog. Sept* (2020).

[13] Mahdi Ghorbani, Emilien Bauer, Tobias Grosser, and Amir Shaikhha. 2024. Compressing Structured Tensor Algebra. *CoRR* abs/2407.13726 (2024). https://doi.org/10.48550/ARXIV.2407.13726 arXiv:2407.13726

[14] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. 2023. Compiling Structured Tensor Algebra. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 229 (oct 2023), 30 pages. https://doi.org/10.1145/3622804

[15] Pulak Ranjan Giri and Vladimir E Korepin. 2017. A review on quantum search algorithms. *Quantum Information Processing* 16 (2017), 1–36.

[16] Alexander Graham. 2018. *Kronecker products and matrix calculus with applications.* Courier Dover Publications.

[17] Philipp Herholz, Xuan Tang, Teseo Schneider, Shoaib Kamil, Daniele Panozzo, and Olga Sorkine-Hornung. 2022. Sparsity-Specific Code Optimization using Expression Trees. *ACM Trans. Graph.* 41, 5 (2022), 175:1–175:19. https://doi.org/10.1145/3520484

[18] Nicolas Heurtel, Andreas Fyrillas, Grégoire de Gliniasty, Raphaël Le Bihan, Sébastien Malherbe, Marceau Pailhas, Eric Bertasi, Boris Bourdoncle, Pierre-Emmanuel Emeriau, Rawad Mezher, Luka Music, Nadia Belabas, Benoît Valiron, Pascale Senellart, Shane Mansfield, and Jean Senellart. 2023. Perceval: A Software Platform for Discrete Variable Photonic Quantum Computing. *Quantum* 7 (Feb. 2023), 931. https://doi.org/10.22331/q-2023-02-21-931

[19] David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng, and Josh A. Izaac. 2024. Catalyst: a Python JIT compiler for auto-differentiable hybrid quantum programs. *J. Open Source Softw.* 9, 99 (2024), 6720. https://doi.org/10.21105/JOSS.06720

[20] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A static single assignment-based quantum program representation for optimization. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–32.

[21] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific reports* 9, 1 (2019), 10736.

[22] Alexei Yu Kitaev, Alexander Shen, and Mikhail N Vyalyi. 2002. *Classical and quantum computation.* Number 47. American Mathematical Soc.

[23] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[24] Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, and Lei Wang. 2020. Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design. *Quantum* 4 (2020), 341. https://doi.org/10.22331/Q-2020-10-11-341

[25] Dan C Marinescu. 2011. *Classical and quantum information.* Academic Press.

[26] Alexander J. McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2021, Broomfield, CO, USA, October 17-22, 2021*, Hausi A. Müller, Greg Byrd, Candace Culhane, and Travis S. Humble (Eds.). IEEE, 255–264. https://doi.org/10.1109/QCE52317.2021.00043

[27] Michael A Nielsen and Isaac L Chuang. 2010. *Quantum Computation and Quantum Information* (10th ed.). Cambridge University Press.

[28] Carlos Outeiral, Martin Strahm, Jiye Shi, Garrett M Morris, Simon C Benjamin, and Charlotte M Deane. 2021. The prospects of quantum computing in computational molecular biology. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 11, 1 (2021), e1481.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[30] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. 2022. QSSA: an SSA-based IR for Quantum computing. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 2–14. https://doi.org/10.1145/3497776.3517772

[31] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[32] John Preskill. 2023. Quantum computing 40 years later. In *Feynman Lectures on Computation*. CRC Press, 193–244.

[33] Qiskit. [n. d.]. Qiskit. QasmSimulator. ([n. d.]). https://qiskit.org/ecosystem/aer/stubs/qiskit_aer.QasmSimulator.html

[34] Shi-Ju Ran, Emanuele Tirrito, Cheng Peng, Xi Chen, Luca Tagliacozzo, Gang Su, and Maciej Lewenstein. 2020. *Tensor network contractions: methods and applications to quantum many-body systems*. Springer Nature.

[35] Lidia Ruiz-Perez and Juan Carlos Garcia-Escartin. 2017. Quantum arithmetic with the quantum Fourier transform. *Quantum Information Processing* 16 (2017), 1–14.

[36] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1 (2023), 37:1–37:27. https://doi.org/10.1145/3588717

[37] Philipp Seitz, Ismael Medina, Esther Cruz, Qunsheng Huang, and Christian B Mendl. 2023. Simulating quantum circuits using tree tensor networks. *Quantum* 7 (2023), 964.

[38] Hesam Shahrokhi and Amir Shaikhha. 2023. Building a Compiled Query Engine in Python. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, Clark Verbrugge, Ondrej Lhoták, and Xipeng Shen (Eds.). ACM, 180–190. https://doi.org/10.1145/3578360.3580264

[39] Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi. 2023. Hinted Dictionaries: Efficient Functional Ordered Sets and Maps. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:30. https://doi.org/10.4230/LIPICS.ECOOP.2023.28

[40] Amir Shaikhha, Mathieu Huot, and Shideh Hashemian. 2024. A Tensor Algebra Compiler for Sparse Differentiation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Tobias Grosser, Christophe Dubach, Michel Steuwer, Jingling Xue, Guilherme Ottoni, and ernando Magno Quintão Pereira (Eds.). IEEE, 1–12. https://doi.org/10.1109/CGO57630.2024.10444787

[41] Amir Shaikhha, Mathieu Huot, Shideh Hashemian, Amirali Kaboli, Alex Mascolo, Milos Nikolic, Jaclyn Smith, and Dan Olteanu. 2024. *A semi-ring dictionary query language for data science*. Technical Report. School of Informatics, University of Edinburgh.

[42] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

[43] Amir Shaikhha, Marios Kelepeshis, and Mahdi Ghorbani. 2023. Fine-Tuning Data Structures for Query Processing. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2023, Montréal, QC, Canada, 25 February 2023- 1 March 2023*, Christophe Dubach, Derek Bruening, and Ben Hardekopf (Eds.). ACM, 149–161. https://doi.org/10.1145/3579990.3580016

[44] Amir Shaikhha and Lionel Parreaux. 2019. Finally, a Polymorphic Linear Algebra Language (Pearl). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom (LIPIcs, Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:29. https://doi.org/10.4230/LIPICS.ECOOP.2019.25

[45] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Q. Ngo. 2024. Optimizing Nested Recursive Queries. *Proc. ACM Manag. Data* 2, 1 (2024), 16:1–16:27. https://doi.org/10.1145/3639271

[46] Peter W Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 124–134.

[47] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 61–70. https://doi.org/10.1109/IPDPS.2015.27

[48] Daniele G. Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 117–127. https://doi.org/10.1145/2854038.2854060

[49] Michelle Mills Strout, Mary W. Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934. https://doi.org/10.1109/JPROC.2018.2857721

[50] Jules Tilly, Hongxiang Chen, Shuxiang Cao, Dario Picozzi, Kanav Setia, Ying Li, Edward Grant, Leonard Wossnig, Ivan Rungger, George H Booth, et al. 2022. The variational quantum eigensolver: a review of methods and best practices. *Physics Reports* 986 (2022), 1–128.

[51] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[52] Thorsten B. Wahl and Sergii Strelchuk. 2023. Simulating Quantum Circuits Using Efficient Tensor Network Contraction Algorithms with Subexponential Upper Bound. *Phys. Rev. Lett.* 131 (Oct 2023), 180601. Issue 18. https://doi.org/10.1103/PhysRevLett.131.180601

[53] Kieran Young, Marcus Scese, and Ali Ebnenasir. 2023. Simulating Quantum Computations on Classical Machines: A Survey. *arXiv preprint arXiv:2311.16505* (2023).

Received 2024-09-12; accepted 2024-11-04