GraFeyn: Efficient Parallel Sparse Simulation of Quantum Circuits

Sam Westrick[1], Pengyu Liu[1], Byeongjee Kang[1], Colin McDonald[1], Mike Rainey[1],
Mingkuan Xu[1], Jatin Arora[1], Yongshan Ding[2], Umut A. Acar[1]
[1]Carnegie Mellon University, USA [2]Yale University, USA

***Abstract*—Many circuit simulators use either a Schrödinger-based or Feynman-based approach, which have complementary strengths. Schrödinger-based simulators maintain a state vector by synchronously updating it after each gate (or group of gates), ensuring time efficiency at the cost of exponential space. In contrast, Feynman-based simulators use low space but require high time to compute the sum of exponentially many independent Feynman paths. Because they treat paths as independent, Feynman-based simulators miss opportunities to take advantage of sparsity from destructive interference.**

In this paper, we present a hybrid Schrödinger-Feynman technique which takes advantage of sparsity by selectively synchronizing Feynman paths. Our hybrid technique partitions the circuit into *kernels* (groups of gates) and uses Feynman simulation within each kernel. It then synchronizes across the kernels by using Schrödinger-style simulation. We parallelize our approach by representing the simulation as a graph, leveraging state-of-the-art parallel graph algorithms. By selecting kernels carefully, we show that our approach can simulate hundreds of qubits efficiently (in both time and space) on just a single multicore node. In certain "sparse" circuits, we are able to improve running times by multiple orders of magnitude.

***Index Terms*—quantum circuit simulation, Feynman paths, parallel computing, sparsity.**

I. Introduction

Quantum computing has numerous advantages over classical computing, especially in areas such as cryptography, machine learning, and physical sciences [1, 2, 3, 4, 5, 6, 7, 8]. Although advances in quantum hardware is progressing relatively rapidly with new computers coming online reasonably regularly [9, 10, 11, 12, 13], the hardware remains "noisy" [14, 15].

There is therefore continuing interest simulating quantum circuits on classical hardware and several approaches have been proposed, including Schrödinger- and Feynman-style simulation. Feynman-style simulation proceeds by exploring each and every Feynman path through the circuit, which requires little memory (space) but can require as much time as 4^m for m gates. Schrödinger-style simulation can reduce time requirements to $m2^n$ for n qubits by maintaining the entire quantum state in a state vector of size 2^n and applying each gate of the circuit. Schrödinger-style simulation also exposes a large amount of parallelism that can be exploited by CPUs and GPUs [16, 17, 18, 19, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Because of exponential time and space requirements, quantum circuit simulation remains challenging, and even simulating several dozen qubits can require supercomputers with thousands of CPU or GPU cores.

In this paper, we propose a technique that identifies and exploits sparsity of the state space in quantum circuits. Specifically, we partition a quantum circuit into subcircuits, which we call ***kernels***, each of which is designed to minimize density or equivalently maximize sparsity of the state vector. We then present a hybrid Feynman-Schrödinger simulation algorithm that uses Feynman-style for simulating each kernel and Schrödinger-style simulations between kernels. This algorithm exploits sparsity within each kernel by using Feynman-paths based simulation, where the time complexity positively correlates with sparsity—the sparser the kernel, the lower the simulation time—and space complexity is minimal. By switching to Schrödinger-style simulation between kernels, the algorithm realizes the potential "interference" between Feynman-paths to reify the simulation state as a state vector, which can then be used as a starting point for the next Feynman-style simulation.

Because our hybrid algorithm operates on a sparse state space, and because it simulates Feynman-paths for each basis index, it is not amenable to data parallelism. Data parallelism works well for highly regular computations that can be represented as operations on dense matrices, but struggles with irregular computations, where the amount of work may vary between the different "iterations" to be parallelized. To parallelize our hybrid algorithm, we formulate its challenging part—the simulation of Feynman-paths—by using a parallel graph algorithm [28, 29, 30, 31]. By representing basis states as vertices of a graph and the transitions between them (as induced by the gates) as edges, our approach represents the sparsity as a graph structure. We then use this graph to express the irregular parallelism by using fork-join parallelism techniques. Because of the emphasis on Feynman paths and the connection with graph theoretic techniques, we call our simulator **GraFeyn**.

Our evaluation on a variety of quantum circuits shows that GraFeyn can efficiently simulate circuits containing hundreds of qubits. On these circuits, due to significant sparsity, GraFeyn is multiple orders of magnitude more efficient than Schrödinger-style simulation, in terms of both time and space (memory). We also measure that the performance of GraFeyn is closely tied to the sparsity of the simulation, and this performance degrades gracefully as the average density increases. These results show that sparsity can be exploited for significant performance gains. We believe that, in future work, GraFeyn could be integrated with a highly optimized dense simulator to achieve the best of both techniques.

In summary, this paper makes the following contributions.

- **GraFeyn**, a hybrid Feynman-Schrödinger style algorithm for circuit simulation that exploits sparsity for improved performance.
- A parallel implementation of the algorithm that can use modern multicore computers and can exploit irregular parallelism.
- An empirical evaluation that shows that the approach can simulate sparse circuits with hundreds of qubits on relatively small multicore computers, delivering orders of magnitude improvement over other simulators.

II. Background

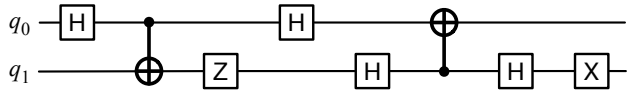
a) Basic definitions and notation: The state of a quantum system with n qubits can be represented as a unit vector in $\mathbb{C}^{2^n}$, called a *state vector*. The basis of this space is indexed by bitstrings of length n, with basis vectors denoted $|\beta\rangle = |q_0 q_1 \dots q_{n-1}\rangle$ where $q_i \in \{0,1\}$ is the state of the i^{th} qubit.[1] We will write $|\beta\rangle$ for a basis vector, and $|\psi\rangle = \sum_i \alpha_i |\beta_i\rangle$ for an arbitrary state, i.e., a linear combination of basis vectors satisfying $\sum_i |\alpha_i|^2 = 1$. Here, $\alpha_i \in \mathbb{C}$ are complex coefficients, called *probability amplitudes*, where $|\alpha_i|^2$ is the probability of measuring $|\beta_i\rangle$ when the system is measured.

b) Circuits and gates: Quantum states can undergo unitary transformations which are commonly represented as *circuits* with primitive operations called *gates*. Each gate typically operates on only one or two qubits at a time, which is understood within a larger system by the decomposition $|\beta\rangle = |q_0 q_1 \dots q_{n-1}\rangle = |q_0\rangle \otimes |q_1\rangle \otimes \dots \otimes |q_{n-1}\rangle$ where $|0\rangle = \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and $|1\rangle = \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ are basis vectors for a single qubit system $\mathbb{C}^2$. A few examples of common gates include the following.

- The quantum "not" gate, $\mathsf{X}(i)$, flips the i^{th} qubit. It is defined by the transformation $\left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right]$.
- The controlled-not gate $\mathsf{CX}(i,j)$ flips the j^{th} qubit but only if the i^{th} qubit is set to 1.
- The Hadamard gate $\mathsf{H}(i)$ applies the transformation $\frac{1}{\sqrt{2}}\left[\begin{smallmatrix}1&1\\1&-1\end{smallmatrix}\right]$ to the i^{th} qubit.
- The phase-flip gate $\mathsf{Z}(i)$ which applies the transformation $\left[\begin{smallmatrix}1&0\\0&-1\end{smallmatrix}\right]$ to the i^{th} qubit.

A circuit can be defined simply as a sequence of gates, to be applied in order from left to right, for example as shown below. Any circuit can also be pictorially represented by a circuit diagram, where each horizontal wire is a qubit.

$$\mathsf{H}(0)\ \mathsf{CX}(0,1)\ \mathsf{Z}(1)\ \mathsf{H}(0)\ \mathsf{H}(1)\ \mathsf{CX}(1,0)\ \mathsf{H}(1)\ \mathsf{X}(1)$$



[1] Throughout the paper we will write these bitstrings reading left-to-right, with the 0^{th} qubit on the left.

c) State vector (Schrödinger-based) simulation: Based on the above definitions, it becomes clear that one could simulate the behavior of a quantum circuit by maintaining a state vector S and iteratively update it by applying gates in sequence (essentially, by a large matrix-vector multiplication on every step). This general approach is known as a *Schrödinger-based simulation*. Writing down the full state vector in memory requires storing 2^n complex amplitudes, which is infeasible even for small n. For example, the full state vector of a system of 50 qubits would require more than a petabyte. The time complexity of a Schrödinger-based simulation is $O(m2^n)$ for m gates, as it requires m rounds, each with $O(2^n)$ memory updates.

d) Feynman-based simulation: Alternatively, a quantum circuit can be simulated by summing many independent Feynman paths. Here, a path refers to a sequence of basis indices of length m where m is the number of gates. The idea is to compute the probability amplitude for all possible paths and then perform a weighted summation to determine the output distribution. For example, in the circuit with two qubits and two gates $(\mathsf{X}(0), \mathsf{X}(1))$, there are $4^2 = 16$ possible paths, corresponding to all combinations of $\{|00\rangle, |10\rangle, |01\rangle, |11\rangle\}$ across two time-steps; in this example, exactly one of these paths (namely, $|00\rangle \to |10\rangle \to |11\rangle$, assuming an initial state of $|00\rangle$) has a probability amplitude of 1, while all other paths have probability 0.

For a circuit composed of 1- and 2-qubit gates, the number of paths can be as much as $O(4^m)$, and these paths can be enumerated via a depth-first search through the gates, requiring only $O(m+n)$ memory. In contrast to Schrödinger, a Feynman-based simulation is inefficient in terms of time. Nevertheless, Feynman-based simulation has a number of advantages: (1) it is highly memory efficient, (2) it is embarrassingly parallel, as every path can be traversed independently, and (3) it can easily take advantage of sparsity, by pruning any path of amplitude 0.

III. Motivation

A. Feynman simulation is time-efficient except for interference

Interference happens when more than one path leads to the same final state. Feynman-based simulation will track all these paths, which might explode exponentially. As a result, Feynman-based simulation is inefficient for general cases. On the other hand, Schrödinger-based simulation only needs to track the state, which is more efficient in most cases.

However, for some circuits, interference is low or non-existent. For example, in a circuit containing only "non-branching" gates such as CX and Rz, Feynman-based simulation is just as time-efficient as Schrödinger-based simulation. This motivates us to improve performance by utilizing Feynman-based simulation on sub-circuits with low interference, which we call ***kernels***.

B. Computing interference appears to require synchronization

Computing interference during the simulation helps to reduce the number of paths to track, which is crucial for performance.

However, computing interference requires synchronization as we have to combine different paths to compare the final state and compute the amplitude after interference. This synchronization can harm parallelism, and requires a large amount of memory, to store intermediate amplitudes.

Feynman-based simulation only synchronizes paths at the end of the simulation. In contrast, Schrödinger-based simulation can be viewed as synchronizing every step. This inspires us to combine the two approaches to get the benefits of both worlds: we use Feynman-based simulation when interference is low, and fall back on Schrödinger-based simulation as a synchronization mechanism to reduce the number of Feynman paths to track.

C. Both approaches (Feynman and Schrödinger) can naturally benefit from sparsity

Many quantum gates are sparse, meaning the corresponding matrices are sparse. Some quantum circuits are also sparse, meaning that the state is sparse at some points in the simulation.

Schrödinger-based simulation can benefit from sparsity by using sparse matrix-vector multiplication, while Feynman-based simulation can benefit from sparsity by tracking fewer paths. We can take advantage of sparsity by using a sparse representation of the state and pruning paths with an amplitude of zero. This can reduce the memory usage and improve the simulation's performance.

D. Reordering gates can result in more sparsity

Simulation is performed by applying gates in sequence. During simulation, there are cases where multiple gates can be applied next, because either they commute or are all in the front layer of the circuit. In dense Schrödinger-based simulations (that do not attempt to take advantage of sparsity), the order of gates does not matter. However, in our hybrid approach, the order of gates can affect

- the sparsity of each kernel, and
- the sparsity of the state at each synchronization point.

We therefore strategically reorder gates to maximize the chance of destructive interference and encourage this interference to occur as early as possible, to improve the simulation's performance by exposing additional sparsity.

E. Overview of approach

The developments in this paper follow from the observations above, by combining Schrödinger-based and Feynman-based techniques to simultaneously get the benefits of both worlds and maximize sparsity for improved performance.

At a high level, our approach is two-fold. First, we rewrite the circuit to select kernels with low interference and expose additional sparsity. Second, we simulate the rewritten circuit by independently traversing Feynman paths in parallel, and force these paths to synchronize after each kernel. Each synchronization computes interference to reduce the number of paths, and also prunes entries of amplitude zero and therefore exploits sparsity for improved performance.

Our approach for interference calculation is reminiscent of a sparse matrix-vector multiplication, and therefore is similar to a Schödinger-style simulation. In this way, our simulator is essentially a hybrid Schödinger-Feynman approach.

IV. Kernelization

Before simulating a circuit, we first ***kernelize*** it, i.e., rewrite the circuit into a sequence of *kernels*, where each kernel is itself a smaller sequence of gates. We note that kernelization is (in principle) similar to gate fusion [21, 25, 27, 32]: in both cases, the goal is to improve performance by grouping together the execution of certain gates.

In our setting, the purpose of kernelization is to delimit the boundary between Feynman and Schrödinger-style simulation. Within each kernel, we use a Feynman-style simulation, which (as discussed in Section III) is time-efficient as long as there is little inteference. To avoid interference, we group together gates that cumulatively have a low ***branching factor***, which takes into account sparsity to give an upper bound on the number of distinct Feynman paths.

At the end of each kernel, we "synchronize" the current set of Feynman paths by aggregating these into a state vector, similar to a Schrödinger-style simulation. This computes interference and prevents a combinatorial explosion on the number of Feynman paths in later kernels. Additionally, this step presents an opportunity to expose sparsity from destructive interference, by pruning away elements of the vector with amplitude zero.

A. Sparse gate application and branching factor

The key primitive in our simulator is a ***sparse gate application***, denoted $\textsc{ApplyGate}(g, \beta)$, which takes a single basis vector $\beta \in \{0,1\}^n$ and applies a gate to it, producing a sparse output, where only the terms with non-zero coefficients are kept. The output is represented as a mapping from basis vector to probability amplitude.

$$\textsc{ApplyGate}(g,\beta) \triangleq \left\{\beta' \mapsto w \;\middle|\; \langle\beta'|g|\beta\rangle = w, w \neq 0\right\} \quad (\text{where } \beta,\beta' \in \{0,1\}^n)$$

By keeping only the non-zero outputs, we can distinguish between different types of gates based on how many non-zeros these gates produce. For example, $\textsc{ApplyGate}(\mathsf{X}(i),\ldots)$ is always a singleton; in contrast, $\textsc{ApplyGate}(\mathsf{H}(i),\ldots)$ always produces two elements.

In general, we can define the ***branching factor*** of a gate, denoted $f(g)$, as the maximum number of non-zero outputs:

$$f(g) \triangleq \max_{\beta\in\{0,1\}^n} |R| \quad \text{where} \quad R = \textsc{ApplyGate}(g,\beta).$$

In Figure 1 we classify gates according to their branching factors.

For efficient simulation, an especially important class of gates are the ***non-branching*** gates, i.e., gates with a branching factor of 1. Many common gates—such as X, CX, etc.—are non-branching. These gates are important because any kernel of non-branching gates can be simulated via Feynman paths efficiently.

Gate	Description	Branching Factor
X	NOT	1 (*non-branching*)
CX	controlled NOT	1 (*non-branching*)
S	phase	1 (*non-branching*)
Z	phase flip	1 (*non-branching*)
T	$\pi/8$ gate	1 (*non-branching*)
SWAP	qubit swap	1 (*non-branching*)
CCX	Toffoli	1 (*non-branching*)
H	Hadamard	2

Fig. 1: Branching factors of common gates

More generally, we have the following theorem, which states that the efficiency of a Feynman-based simulation can be bounded by the branching factors of individual gates.

Theorem 1. *For any sequence of gates $(g_0, \ldots, g_{m-1})$ and any basis vector $|\beta\rangle$ as input, the number of sparse Feynman paths is exactly $\prod_i f(g_i)$.*

Proof. We can organize the sparse Feynman paths into a rooted tree, where each path is identified by a leaf of the tree. We have $|\beta\rangle$ at the root, and each non-leaf node of the tree corresponds to a sparse gate application. The gates are applied in order (i.e., each node of the tree at depth i corresponds to an application of gate g_i), and therefore each node at depth i has exactly $f(g_i)$ branches. The number of leaves therefore is $\prod_i f(g_i)$. □

Corollary 1. *For any sequence of gates $(g_0, \ldots, g_{m-1})$ and any initial state $|\psi\rangle$, the number of sparse Feynman paths is at most $nz(|\psi\rangle) \prod_i f(g_i)$ where $nz(|\psi\rangle)$ is the number of basis vectors with non-zero amplitude in $|\psi\rangle$.*

B. Kernelization algorithm

We propose a greedy kernelization strategy that produces kernels with low branching factor. The idea is simply to select the longest run of gates such that $\prod_i f(g_i) \leq f_{\max}$, where $f_{\max}$ is a constant threshold called the ***maximum branching factor***. This guarantees that any Feynman-style inefficiency, due to interference, is bounded. In our experiments, we use $f_{\max} = 4$.

V. GraFeyn: Parallel Sparse Simulation

In this section we present our simulation algorithm, called GraFeyn. The algorithm bears close resemblence to parallel graph traversals that have been developed by the parallel algorithms community (e.g., [28, 29, 30, 31]), and in Section V-E, we describe this connection in more detail.

GraFeyn optimizes for sparsity using dynamically resizing hash tables. We use one hash table for each sparse state during execution, i.e., one hash table per synchronization point (which occurs between kernels). These hash tables are used to accumulate the results of the Feynman paths of a kernel. The keys of these hash tables are basis vectors $\beta \in \{0,1\}^n$, and the associated values are complex amplitudes. We refer to these amplitudes as ***weights***, denoted with variables named w, w', etc.

```
procedure GraFeyn(C, |ψ⟩)
    S := {β ↦ w | ⟨β|ψ⟩ = w, w ≠ 0}   ▷ initialize sparse state
    while kernels remain in C do
        K := choose next kernel from C
        T := {}                          ▷ fresh empty sparse state
        if |S| < dense threshold then
            for all (β ↦ w) ∈ S do in parallel
                ForwardFeynmanKernel(T, β, w, K, 0)
            end for
        else
            for all β ∈ {0,1}^n do in parallel
                w := BackwardFeynmanKernel(S, β, K, |K|)
                insert (β ↦ w) into T
            end for
        end if
        S := {(β ↦ w) ∈ T | w ≠ 0}       ▷ prune zeros
    end while
    return S
end procedure

procedure ForwardFeynmanKernel(T, β, w, K, i)
    if i < |K| then      ▷ still more gates to execute in this kernel
        g := i^th gate in kernel K
        R := ApplyGate(g, β)   ▷ sparse application; see §IV-A
        for all (β' ↦ w') ∈ R do in parallel
            ForwardFeynmanKernel(T, β', w * w', K, i + 1)
        end for
    else                                 ▷ done executing the kernel
        atomically insert (β ↦ w) into T          ▷ see §V-D
        (accumulate weights if β is already present)
    end if
end procedure

procedure BackwardFeynmanKernel(S, β, K, i)
    if i > 0 then        ▷ still more gates to execute in this kernel
        g := (i − 1)^th gate in kernel K
        R := ApplyGateBackward(g, β)
        w :=          ▷ parallel reduction with addition, output to w
            for all (β' ↦ w') ∈ R reduce(+) in parallel
                w'*BackwardFeynmanKernel(S, β', K, i − 1)
            end for
        return w
    else                                 ▷ done executing the kernel
        return S[β]
    end if
end procedure
```

Fig. 2: GraFeyn simulation algorithm.

An essential component of our algorithm is parallelism, which is especially important for dense executions, but is relevant even in highly sparse executions. For example, for $n = 30$ qubits, even at 99.9% sparsity, the size of a sparse state vector will be $0.001 \cdot 2^{30} \approx 10^6$ basis indices, and in principle, all of these elements can be processed in parallel. To support parallelism, our implementation uses a lock-free hash table to ensure efficient atomic updates, and relies on a custom resizing strategy, which we describe in more detail in Section V-D.

A. Algorithm description

Figure 2 shows pseudocode for GraFeyn. The algorithm takes a circuit C (represented as a sequence of kernels) and an

initial state $|\psi\rangle$ as input, and outputs a sparse state vector S representing the final state of the simulation. It then proceeds in rounds, where on each round a new sparse state T is constructed from the previous round by walking the Feynman paths of a kernel.

Similar to the so-called "direction optimization" of Beamer et al. [29], we use two strategies for executing a kernel: a forward version, and a backward version. The forward version (described below, in Section V-B) is used whenever the number of non-zero elements of the current state vector is small. Otherwise, when the state is large (i.e., dense), the backward version is used (this is described in Section V-C). In both cases, the algorithm parallelizes across basis vectors β.

At the end of each round, the algorithm updates S by pruning away weights of value zero in T, and continues to the next round. This proceeds until all kernels in the circuit have been executed.

a) Note on notation: In pseodocode, we use $\beta \in \{0,1\}^n$ consistently to refer to basis vectors, and variables w to refer to weights (i.e., complex amplitudes). For hash tables, we use a set-like notation. The syntax $(\beta \mapsto w)$ denotes a single key-value pair, and we write $T[\beta]$ for a lookup of a key β.

B. Forward kernel execution

The procedure ForwardFeynmanKernel(T, β, w, K, i) executes a kernel by walking Feynman paths starting at $|\beta\rangle$ with weight w, writing results into T. The kernel K is a sequence of gates, and the variable i (indexed from 0) is the current position within the kernel, i.e., the index of the next gate to execute.

When $i < |K|$, the procedure applies a gate from the kernel and continues recursively in a depth-first manner. The gate is applied sparsely, as described in Section IV-A, and therefore the number of recursive calls to ForwardFeynmanKernel will be determined by the branching factor of the gate. For example, if the gate is X which is non-branching, then there will only be a single recursive call. All recursive calls can proceed in parallel.

When $i = |K|$, the whole kernel has been executed, and T is updated by inserting the key-value pair $\beta \mapsto w$. If some other $\beta \mapsto w'$ is already present in the table (due to interference), then the update will aggregate the results, resulting in $\beta \mapsto w+w'$. This is one of the places where GraFeyn "synchronizes" Feynman paths and computes interference.

Note that, to be safe for parallel execution, all updates into the hash table must be performed atomically. We describe atomic updates and other hash table details in Section V-D.

C. Backward kernel execution

When the current state S is sufficiently dense, the overhead of hashing and concurrent aggregation for a sparse state vector can outweigh the benefits of the sparse representation. We therefore switch to a different strategy, where each element out of the output is generated in parallel by "pulling" contributions from the previous state. To implement this, we traverse Feynman paths backwards.

The key primitive in this case is similar to the ApplyGate primitive of the forward direction, but with input and output basis vectors of the gate application reversed. This primitive, called ApplyGateBackward, is defined as follows.

$$\begin{aligned}&\textsc{ApplyGateBackward}(g, \beta)\\ &\triangleq \left\{\beta' \mapsto w \;\middle|\; \langle\beta|g|\beta'\rangle = w, w \neq 0\right\}\\ &\quad (\text{where } \beta, \beta' \in \{0,1\}^n)\end{aligned}$$

Note that backward application in this manner respects all of the same sparsity properties of forward application, i.e., the branching factor of a backwards gate is the same as the gate in the forwards direction. Therefore, we can traverse Feynman paths in exactly the same manner, but in the reverse direction.

The implementation of BackwardFeynmanKernel is otherwise very similar to ForwardFeynmanKernel, with a few small differences. First, the kernel is executed in reverse, with $i = |K|$ initially and counting down to 0. Second, we perform a parallel reduction over the recursive calls, which computes the incoming interference and accumulates this into a single output amplitude. Third and finally, when we have completed traversing the kernel in reverse (i.e., when $i = 0$), then we simply perform a lookup of the appropriate basis vector in S, which is the sparse state vector of the previous round. The structure of the recursive calls then propagates these results forwards.

D. Parallel hashing and dynamic resizing

To support atomic updates and value aggregation, we use lock-free hash tables based on open addressing. Updates are performed with a two-step procedure. First, an appropriate hash table slot is located (and claimed using a compare-and-swap if the key is not already present). Then, after the slot has been identified, we use atomic fetch-and-add primitives to update the two components (a, b) of the complex value $w = a + bi$. This approach is correct as long as no lookups are performed concurrently while an update is still in flight, which GraFeyn naturally guarantees.

When performing an update, it may be the case that the hash table is full and needs to be resized. To perform resizing, we implement a custom procedure as follows. As soon as any one thread detects that the hash table is full, a shared boolean (visible to all threads) is set, and each thread periodically polls this boolean to check if the table is full. All threads then participate in resizing the hash table, replacing it with a table that is a constant fraction larger. After the resizing is complete, each thread resumes its normal execution.

Resizing occurs only in forward kernel executions. In the backward (dense) kernel executions, the initial table size can be chosen to be exactly 2^n; in fact, our implementation avoids using a hash table entirely in the dense case, and instead simply fills an array in parallel.

E. Interpreting GraFeyn as a parallel graph traversal

The implementation of GraFeyn is inspired by parallel graph traversal algorithms (e.g., [28, 29, 30, 31]) which utilize fine-grained irregular parallelism to efficiently process sparse graphs.

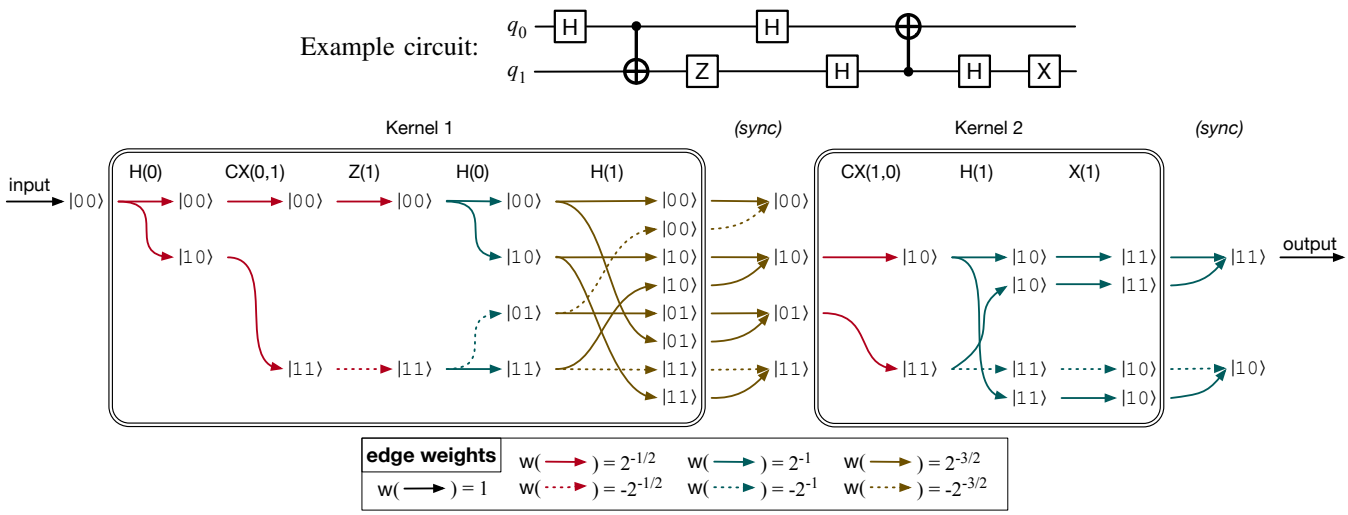


Fig. 3: Illustrating the execution of GraFeyn as a graph traversal, with vertices corresponding to basis vectors and edges corresponding to non-zero (complex) weights. We show a simulation of an example circuit on two qubits assuming an initial state $|\psi\rangle = |00\rangle$. The simulation consists of two kernels (forward execution only), and two corresponding synchronization points.

To make this connection more clear, we can interpret the execution of GraFeyn as a graph traversal of a particular graph which is generated "on-the-fly."

We present an example in Figure 3, which illustrates the execution of GraFeyn on an example circuit of two qubits, assuming an input of $|00\rangle$. Here, we consider only the "forward" kernel executions of GraFeyn. In the figure, each vertex corresponds to either a sparse gate application or a synchronization point, and each edge corresponds to a non-zero weight. The edges are drawn with different colors and dashing according to the associated weights.

In the figure, we label each vertex with a basis vector and organize these vertices into columns, where each column can be understood as a state vector. GraFeyn only computes state vectors at explicit synchronization points, immediately after each kernel. The example shows two kernels, and two corresponding synchronization points. At each synchronization, GraFeyn computes interference, deduplicates basis vectors, and prunes weights of value zero. Within each kernel, sparse Feynman paths are traversed independently.

Based on this graph-based interpretation of GraFeyn, we note that the interference calculations (i.e., synchronization points) of GraFeyn are inspired by—and similar in implementation to—the EdgeMap primitives of Shun and Blelloch [28] and Dhulipala et al. [31]. These primitives utilize Beamer et al. [29]'s "direction optimization", which is similar to our distinction between forwards and backwards kernels.

F. Gate scheduling

As stated thus far, our approach takes a sequence of gates, kernelizes this sequence (Section IV), and then applies the GraFeyn algorithm to the resulting sequence of kernels. An immediate question, however, is whether or not the initial ordering of gates (before kernelization) affects performance. We have found that this input ordering can have a significant impact on sparsity, and to control for this, we propose a greedy gate scheduler which reorders the initial gate sequence to encourage additional sparsity.

The high-level idea of our gate scheduler is to pick a qubit, execute all gates on that qubit, and then pick another qubit and repeat until all gates have been executed. To execute a gate, however, all of its dependencies must first be executed. Therefore, at each step, our scheduler first executes the fewest number of other gates as possible to make progress towards picking the next desired gate on the chosen qubit.

VI. Implementation and Evaluation

We implemented our techniques using the MPL [33] programming language. All code and experiment scripts are available open-source at https://github.com/CMU-TOP/grafeyn. Our implementation is highly parallel and closely follows the algorithm as described in Section V, but is otherwise largely unoptimized. Nevertheless, in this section, we show that our approach is able to outperform existing simulators, including Qiskit [21] and QSim [34], in some cases by multiple orders of magnitude. We also show that our techniques are capable of simulating certain "sparse" circuits on hundreds of qubits.

a) Experimental setup and benchmarks: We run our experiments on a 64-core AWS `r6i.32xlarge` instance, which has 2×2.9GHz Intel Xeon (32-core) 8375C CPUs and 1TB of memory. All reported numbers use 128 threads (hyper-threaded). We compile our implementation with MPL version 0.4, and compare against Qiskit version 0.39.5 (with `qiskit-terra` version 0.22.4 and `qiskit-aer` version 0.11.2) and QSim version 0.16.3. We use a variety of benchmarks from QASMBench [35] and MQTBench [36], selected to cover multiple families and sizes of quantum algorithms. We run each benchmark at least 10 times, and report averages for

Benchmark			Relative Density		Non-zeros	GraFeyn Absolute		Qiskit Absolute		Qiskit Relative		QSim Absolute		QSim Relative	
	Qubits	Gates	Avg	Max	Max	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
adder	28	88	<0.1%	<0.1%	1	0.0001	0.0195	1.91	4.30	19100	221	1.15	4.35	11500	223
	433	1393	<0.1%	<0.1%	1	0.0087	0.0341	–	–	–	–	–	–	–	–
multiplier	45	689	<0.1%	<0.1%	1	0.0011	0.0266	–	–	–	–	–	–	–	–
	400	57237	<0.1%	<0.1%	1	0.401	0.617	–	–	–	–	–	–	–	–
bv	30	78	<0.1%	<0.1%	4	0.0010	0.0195	2.79	16.9	2790	867	4.56	16.9	4560	867
	70	176	<0.1%	<0.1%	4	0.0163	0.0223	–	–	–	–	–	–	–	–
	280	712	<0.1%	<0.1%	4	0.181	0.0327	–	–	–	–	–	–	–	–
cat	35	35	<0.1%	<0.1%	2	0.0001	0.0191	–	–	–	–	–	–	–	–
	260	260	<0.1%	<0.1%	2	0.0012	0.0222	–	–	–	–	–	–	–	–
ghz	40	40	<0.1%	<0.1%	2	0.0001	0.0191	–	–	–	–	–	–	–	–
	255	255	<0.1%	<0.1%	2	0.0013	0.0221	–	–	–	–	–	–	–	–
dj_indep	30	88	<0.1%	<0.1%	60	0.0030	0.0206	4.20	16.9	1400	820	5.18	16.9	1727	820
	62	184	<0.1%	<0.1%	128	0.0099	0.0243	–	–	–	–	–	–	–	–
	130	388	<0.1%	<0.1%	240	2.20	0.106	–	–	–	–	–	–	–	–
qram	20	27	<0.1%	<0.1%	1	0.0001	0.0190	0.0087	0.107	87.0	5.63	0.0102	0.168	102	8.84
knn	31	47	3.3%	36.7%	787991531	9.63	54.0	10.3	33.6	1.07	0.62	7.45	33.7	0.77	0.62
qft	29	2059	10.4%	100.0%	536870912	4.28	16.6	9.78	8.50	2.29	0.51	15.2	8.55	3.55	0.52
ising	26	280	14.6%	100.0%	67108864	2.30	3.35	0.249	1.16	0.11	0.35	0.274	1.19	0.12	0.36
	34	368	11.3%	100.0%	17179869184	638	652	76.7	268	0.12	0.41	–	–	–	–
dnn	33	874	13.2%	100.0%	8589331981	824	261	61.1	134	0.07	0.51	–	–	–	–

TABLE I: Experimental data of our GraFeyn simulator, and comparison with Qiskit and QSim. Absolute memory is measured in GB and time in seconds. Relative memory and times are computed with respect to GraFeyn; higher is better for GraFeyn. The relative density is the number of non-zero entries relative to 2^n. Cells marked "–" indicate either a crash or out-of-memory.

both time and space measurements. Space (peak memory usage) is measured as the maximum resident set size, as reported by Linux.

A. Results

The results of our evaluation are presented in Figure I. Each row corresponds to a benchmark instance, and the columns include some properties of the benchmark, the results of our GraFeyn simulator, and the results of Qiskit and QSim. The key metrics are the time taken to simulate the circuit and the peak memory usage, which we report both the absolute and relative values compared to GraFeyn. Absolute memory is measured in GB and time in seconds. For the relative value, higher is better for GraFeyn. We also report the state vector's average and maximum relative density, which is the ratio of the number of non-zero amplitudes to the total number of amplitudes. Cells marked "–" indicate that the simulator consistently ran out of memory or crashed.

a) Many quantum circuits have significant sparsity: We observe that across many different families of quantum circuits, there is significant sparsity that can be exploited for improved performance. In particular, our results show that the following benchmark families are almost entirely sparse: adder, multiplier, bv, cat, ghz, and qram. In these benchmark families, on up to hundreds of qubits, GraFeyn requires only a handful of non-zero amplitudes and completes in a short amount of time. Running times for these families are less than a second across the board, and less than $1/10^{\text{th}}$ of a second in 10 out of 12 instances. Other benchmark families are also highly sparse, such as dj_indep, where the number of non-zero amplitudes appears to scale only linearly with the number of qubits (as opposed to exponentially).

For other benchmark families, like qft, ising, and dnn, the maximum relative density is 100%, meaning that we still need $O(2^n)$ memory to store the state vector. However, since the relative average density is significantly less than 100%, sparse simulation can save computation and time.

b) GraFeyn scales to hundreds of qubits on sparse circuits: We observe that GraFeyn efficiently handles hundreds of qubits for circuits that are highly sparse. On the adder benchmark in particular, GraFeyn simulates 433 qubits and 1393 gates in less than $1/10^{\text{th}}$ of a second and only 34MB of space. Similarly, the bv benchmark with 280 qubits and 712 gates completes in less than 0.2 seconds using only 33MB. In contrast, Qiskit and QSim run out of memory for these benchmarks.

In instances of sparse circuits where the number of qubits is small enough to simulate with QSim and Qiskit, we observe that GraFeyn is multiple orders of magnitude more efficient than both QSim and Qiskit in terms of space and time. For example, on the bv circuit with 30 qubits, GraFeyn is approximately $2800\times$ faster than Qiskit and uses $870\times$ less memory; the results in comparison to QSim are similar. On the adder circuit with 28 qubits, GraFeyn is nearly $20000\times$ faster than Qiskit.

c) GraFeyn's performance improves with sparsity: We observe that GraFeyn's performance improves with the amount of sparsity in the circuit simulation, and this improvement scales linearly with the amount of sparsity. In Figure 4, we compare the average time per gate vs the average number of non-zero amplitudes for a few benchmark instances, and see a linear relationship between them. This is because the actual computation required is proportional to the number of non-zero amplitudes, which determines the time per gate. Although there are only a few data points, we compute a linear regression and see a slope of 9.00×10^{-10}, which means that our simulator can simulate 1.1×10^9 amplitude-gate pairs per second on average.

For circuits that are sufficiently dense, GraFeyn is outperformed by both Qiskit and QSim. For example, GraFeyn is approximately 10x slower than Qiskit on both ising circuits, and uses approximately 3x as much memory. The reason is that

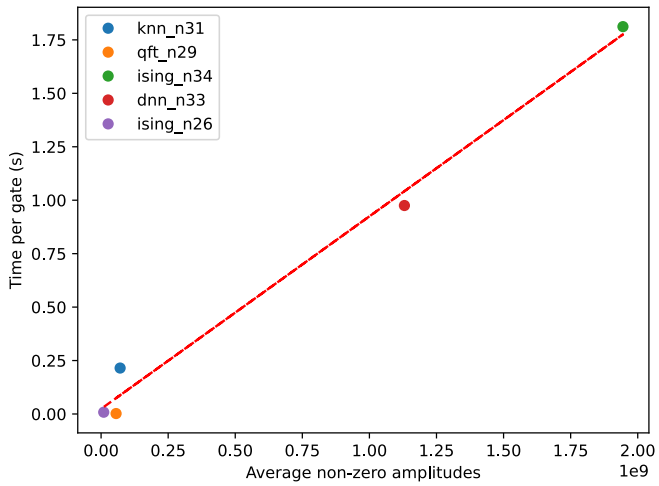


Fig. 4: Average non-zero amplitudes vs. time per gate for Grafeyn. The red line is the linear regression.

GraFeyn currently does not execute dense kernels efficiently. We believe that, in future work, the dense rounds of GraFeyn could instead use an efficient Schrödinger-based technique, and this would improve the performance of GraFeyn on dense circuits.

VII. Related Work

a) Parallel quantum circuit simulation: Several recent works have tried to accelerate Schrödinger-based quantum circuit simulation with data-parallelism [19, 26, 27, 37, 38, 39, 40, 41, 42, 43, 44]. Previous work shows that efficient parallel simulation using a sparse representation of the quantum state is challenging because the hash map used for the state is not thread-safe for concurrent update operations [45]. In our work, we leverage an efficient lock-free concurrent hash table to perform fine-grained parallel updates to sparse states.

b) Quantum gate fusion and scheduling: Prior work has used gate fusion techniques to accelerate quantum circuit simulation and reduce overhead [21, 25, 27, 32, 44]. In our work, we notice that it is possible to kernelize for sparsity and exploit the sparsity to dramatically improve the efficiency of simulation.

For Schrödinger-based simulators that do not exploit sparsity, the simulation ordering of gates has no significant impact on performance. But for sparse simulations, previous work has observed that the order of simulation has a significant impact on performance [45]. Our work proposes a greedy scheduling algorithm to optimize the simulation order for sparse simulation.

c) Hybrid Schrödinger-Feynman simulation: There have been existing works that use ideas from both Schrödinger-based and Feynman-based approaches [46, 47], but they scale up to only 45 qubits (or 56 qubits for weak simulation). Our approach is more systematic and is able to simulate hundreds of qubits efficiently.

d) Structured quantum circuit simulators: It is widely believed that quantum circuits are difficult to simulate in general [48]. However, it is still possible to exploit special structures of quantum circuits to realize efficient simulation. Our work exploits the sparsity of quantum circuits while there are other works that focus on different structures.

Circuits with low entanglement can be efficiently simulated by matrix product states and tensor networks [21, 49, 50, 51]. Clifford circuits with no or few T gates can also be efficiently simulated [52]. There are also efficient simulators for Clifford circuits with errors [53, 54]. Quantum circuits that result in compressible states can be accelerated by decision diagrams [55, 56] or classical compression techniques [39, 57]. There are also works that focus on the simulation of specific classes of quantum circuits such as quantum Fourier transformation, special implementations of Shor's algorithm, or many-body quantum systems [45, 58, 59].

e) Parallel graph algorithms: In this paper, we use irregular parallel graph algorithms [30, 31] to parallelize Feynman-paths based simulation of quantum circuits. These algorithms differ from those used in state-vector simulations because they can parallelize sparse state vectors and the resulting irregularity of parallel computations. The algorithms express parallelism by using fork-join parallelism (e.g., parallel for-loops and reductions which can be recursively nested), with no limitations on the irregularity between iterations. This approach to irregular parallelism has proved effective over the past decade in a variety of languages and systems, e.g., C [60], Java [61], X10 [62], and Parallel ML [63, 64, 65, 66].

VIII. Conclusion

The goal of this paper is not to develop the fastest simulator, but rather to demonstrate that Feynman paths can be used to efficiently execute "sparse" circuits, with multiple order-of-magnitude improvements in time and space consumption. To this end, we develop GraFeyn, a hybrid Feynman-Schrödinger quantum simulator which takes advantage of sparsity by incorporating a parallel Feynman-style simulation technique. Our experiments show that (1) sparsity is common across a range of quantum circuits, and (2) if the execution of a circuit is sufficiently sparse, then the number of qubits in the circuit is largely inconsequential. In particular, we show that GraFeyn can simulate certain "sparse" circuits with hundreds of qubits in less than a second.

Looking forward, we believe it is possible to integrate GraFeyn with existing state-of-the-art dense simulation techniques, to take advantage of sparsity when it is present but otherwise fall back on a more efficient approach whenever density is sufficiently high. We plan to explore this approach in future work.

Acknowledgments

This research was supported by the NSF under the grants CCF-1901381, CCF-2115104, CCF-2119352, and CCF-2107241.

References

[1] M. A. Nielsen and I. L. Chuang, "Quantum information and quantum computation," *Cambridge: Cambridge University Press*, vol. 2, no. 8, p. 23, 2000.

[2] C. H. Bennett and G. Brassard, "Quantum cryptography: Public key distribution and coin tossing," *arXiv preprint arXiv:2003.06557*, 2020.

[3] M. Schuld and N. Killoran, "Quantum machine learning in feature hilbert spaces," *Physical review letters*, vol. 122, no. 4, p. 040504, 2019.

[4] R. Orús, S. Mugel, and E. Lizaso, "Quantum computing for finance: Overview and prospects," *Reviews in Physics*, vol. 4, p. 100028, 2019.

[5] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning," *Nature*, vol. 549, no. 7671, pp. 195–202, 2017.

[6] R. Babbush, J. McClean, D. Wecker, A. Aspuru-Guzik, and N. Wiebe, "Chemical basis of trotter-suzuki errors in quantum chemistry simulation," *Physical Review A*, vol. 91, no. 2, p. 022311, 2015.

[7] S. Lloyd, M. Mohseni, and P. Rebentrost, "Quantum algorithms for supervised and unsupervised machine learning," *arXiv preprint arXiv:1307.0411*, 2013.

[8] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, and M. Head-Gordon, "Simulated quantum computation of molecular energies," *Science*, vol. 309, no. 5741, pp. 1704–1707, 2005.

[9] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[10] G. Q. A. lab, "A preview of bristlecone, google's new quantum processor," https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html, 2018.

[11] H.-S. Zhong, H. Wang, Y.-H. Deng, M.-C. Chen, L.-C. Peng, Y.-H. Luo, J. Qin, D. Wu, X. Ding, Y. Hu *et al.*, "Quantum computational advantage using photons," *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020.

[12] C. Q. Choi, "Ibm unveils 433-qubit osprey chip," Mar 2023. [Online]. Available: https://spectrum.ieee.org/ibm-quantum-computer-osprey

[13] D. Castelvecchi, "Ibm releases first-ever 1,000-qubit quantum chip," Dec 2023. [Online]. Available: https://www.nature.com/articles/d41586-023-03854-1

[14] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, aug 2018. [Online]. Available: https://doi.org/10.22331%2Fq-2018-08-06-79

[15] A. D. Corcoles, A. Kandala, A. Javadi-Abhari, D. T. McClure, A. W. Cross, K. Temme, P. D. Nation, M. Steffen, and J. M. Gambetta, "Challenges and opportunities of near-term quantum computing systems," *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1338–1352, aug 2020. [Online]. Available: https://doi.org/10.1109%2Fjproc.2019.2954005

[16] A. Amariutei and S. Caraiman, "Parallel quantum computer simulation on the gpu," in *15th International Conference on System Theory, Control and Computing*. IEEE, 2011, pp. 1–6.

[17] A. Avila, A. Maron, R. Reiser, M. Pilla, and A. Yamin, "Gpu-aware distributed quantum simulation," in *Proceedings of the 29th Annual ACM symposium on applied computing*, 2014, pp. 860–865.

[18] A. Avila, R. H. Reiser, M. L. Pilla, and A. C. Yamin, "Optimizing d-gm quantum computing by exploring parallel and distributed quantum simulations under gpus arquitecture," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 5146–5153.

[19] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik, "qhipster: The quantum high performance software testing environment," *arXiv preprint arXiv:1601.07195*, 2016.

[20] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3126908.3126947

[21] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen *et al.*, "Qiskit: An open-source framework for quantum computing," *Accessed on: Mar*, vol. 16, 2019.

[22] E. Gutiérrez, S. Romero, M. A. Trenas, and E. L. Zapata, "Quantum computer simulation using the cuda programming model," *Computer Physics Communications*, vol. 181, no. 2, pp. 283–300, 2010.

[23] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "Quest and high performance simulation of quantum computers," *Scientific reports*, vol. 9, no. 1, pp. 1–11, 2019.

[24] P. Zhang, J. Yuan, and X. Lu, "Quantum computer simulation on multi-gpu incorporating data locality," in *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part I 15*. Springer, 2015, pp. 241–256.

[25] Y. Suzuki, Y. Kawase, Y. Masumura, Y. Hiraga, M. Nakadai, J. Chen, K. M. Nakanishi, K. Mitarai, R. Imai, S. Tamiya *et al.*, "Qulacs: a fast and versatile quantum circuit simulator for research purpose," *Quantum*, vol. 5, p. 559, 2021.

[26] D. Park, H. Kim, J. Kim, T. Kim, and J. Lee, "SnuQS: scaling quantum circuit simulation using storage devices," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3524059.3532375

[27] C. Zhang, Z. Song, H. Wang, K. Rong, and J. Zhai, "Hyquas: Hybrid partitioner based quantum circuit simulation system on gpu," in *Proceedings of the ACM*

International Conference on Supercomputing, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 443–454. [Online]. Available: https://doi.org/10.1145/3447818.3460357

[28] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *PPOPP '13*. New York, NY, USA: ACM, 2013, pp. 135–146.

[29] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12*, 2012, pp. 12:1–12:10.

[30] U. A. Acar and G. E. Blelloch, *Algorithms: Parallel and Sequential*, 2022, http:www.algorithms-book.com.

[31] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM Trans. Parallel Comput.*, vol. 8, no. 1, pp. 4:1–4:70, 2021. [Online]. Available: https://doi.org/10.1145/3434393

[32] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. García-Sáez, J. I. Latorre, and S. Carrazza, "Qibo: a framework for quantum simulation with hardware acceleration," *CoRR*, vol. abs/2009.01845, 2020. [Online]. Available: https://arxiv.org/abs/2009.01845

[33] U. A. Acar, J. Arora, M. Fluet, R. Raghunathan, S. Westrick, and R. Yadav, "Mpl: A high-performance compiler for parallel ml," 2020, https://github.com/MPLLang/mpl.

[34] Q. A. team and collaborators, "qsim," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4023103

[35] A. Li, S. Stein, S. Krishnamoorthy, and J. Ang, "Qasmbench: A low-level qasm benchmark suite for nisq evaluation and simulation," *arXiv preprint arXiv:2005.13018*, 2021.

[36] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking software and design automation tools for quantum computing," 2022, MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[37] A. Zulehner and R. Wille, "Advanced simulation of quantum computations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 848–859, 2018.

[38] Z.-Y. Chen, Q. Zhou, C. Xue, X. Yang, G.-C. Guo, and G.-P. Guo, "64-qubit quantum circuit simulation," *Science Bulletin*, vol. 63, no. 15, pp. 964–971, 2018.

[39] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–24.

[40] J. Doi, H. Takahashi, R. Raymond, T. Imamichi, and H. Horii, "Quantum computing simulator on a heterogenous HPC system," in *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*, F. Palumbo, M. Becchi, M. Schulz, and K. Sato, Eds. ACM, 2019, pp. 85–93. [Online]. Available: https://doi.org/10.1145/3310273.3323053

[41] A. Fatima and I. L. Markov, "Faster schrödinger-style simulation of quantum circuits," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 194–207. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00026

[42] Y. Zhao, Y. Chen, H. Li, Y. Wang, K. Chang, B. Wang, B. Li, and Y. Han, "Full state quantum circuit simulation beyond memory limit," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

[43] C. Zhang, H. Wang, Z. Ma, L. Xie, Z. Song, and J. Zhai, "UniQ: A unified programming model for efficient quantum circuit simulation," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–16, ISSN: 2167-4337.

[44] M. Xu, S. Cao, X. Miao, U. A. Acar, and Z. Jia, "Atlas: Hierarchical partitioning for quantum circuit simulation on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2024.

[45] S. Jaques and T. Häner, "Leveraging state sparsity for more efficient quantum simulations," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, jun 2022. [Online]. Available: https://doi.org/10.1145/3491248

[46] I. L. Markov, A. Fatima, S. V. Isakov, and S. Boixo, "Massively parallel approximate simulation of hard quantum circuits," in *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/DAC18072.2020.9218591

[47] L. Burgholzer, H. Bauer, and R. Wille, "Hybrid schrödinger-feynman simulation of quantum circuits with decision diagrams," in *IEEE International Conference on Quantum Computing and Engineering, QCE 2021, Broomfield, CO, USA, October 17-22, 2021*, H. A. Müller, G. Byrd, C. Culhane, and T. S. Humble, Eds. IEEE, 2021, pp. 199–206. [Online]. Available: https://doi.org/10.1109/QCE52317.2021.00037

[48] R. Movassagh, "The hardness of random quantum circuits," *Nature Physics*, vol. 19, no. 11, pp. 1719–1724, 2023.

[49] S.-X. Zhang, J. Allcock, Z.-Q. Wan, S. Liu, J. Sun, H. Yu, X.-H. Yang, J. Qiu, Z. Ye, Y.-Q. Chen, C.-K. Lee, Y.-C. Zheng, S.-K. Jian, H. Yao, C.-Y. Hsieh, and S. Zhang, "TensorCircuit: a Quantum Software Framework for the NISQ Era," *Quantum*, vol. 7, p. 912, Feb. 2023. [Online]. Available: https://doi.org/10.22331/q-2023-02-02-912

[50] R. Jozsa and N. Linden, "On the role of entanglement in quantum-computational speed-up," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 459, no. 2036, pp. 2011–2032, aug 2003.

[51] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM Journal on*

Computing, vol. 38, no. 3, pp. 963–981, 2008.

[52] K. N. Smith, M. A. Perlin, P. Gokhale, P. Frederick, D. Owusu-Antwi, R. Rines, V. Omole, and F. Chong, "Clifford-based circuit cutting for quantum simulation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589352

[53] C. Gidney, "Stim: a fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, Jul. 2021. [Online]. Available: https://doi.org/10.22331/q-2021-07-06-497

[54] R. S. Bennink, E. M. Ferragut, T. S. Humble, J. A. Laska, J. J. Nutaro, M. G. Pleszkoch, and R. C. Pooser, "Unbiased simulation of near-clifford quantum circuits," *Phys. Rev. A*, vol. 95, p. 062337, Jun 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.95.062337

[55] S. Li, Y. Kimura, H. Sato, and M. Fujita, "Parallelizing quantum simulation with decision diagrams," *IEEE Transactions on Quantum Engineering*, vol. 5, no. 01, pp. 1–12, jan 2024.

[56] A. Zulehner, S. Hillmich, and R. Wille, "How to efficiently handle complex values? implementing decision diagrams for quantum computing," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–7.

[57] Y. Zhao, Y. Guo, Y. Yao, A. Dumi, D. M. Mulvey, S. Upadhyay, Y. Zhang, K. D. Jordan, J. Yang, and X. Tang, "Q-gpu: A recipe of optimizations for quantum circuit simulation using gpus," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 726–740.

[58] D. Aharonov, Z. Landau, and J. Makowsky, "The quantum fft can be classically simulated," *arXiv preprint quant-ph/0611156*, 2006.

[59] J. Richter, "Simulating the dynamics of large many-body quantum systems with schrödinger-feynman techniques," *arXiv preprint arXiv:2403.19864*, 2024.

[60] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1995, pp. 207–216.

[61] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00, 2000, pp. 36–43.

[62] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. ACM, 2005, pp. 519–538.

[63] M. Fluet, M. Rainey, J. H. Reppy, and A. Shaw, "Implicitly-threaded parallelism in Manticore," in *ICFP*, 2008, pp. 119–130.

[64] M. Fluet, M. Rainey, J. Reppy, and A. Shaw, "Implicitly threaded parallelism in Manticore," *Journal of Functional Programming*, vol. 20, no. 5-6, pp. 1–40, 2011.

[65] J. Arora, S. Westrick, and U. A. Acar, "Provably space efficient parallel functional programming," in *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[66] ——, "Efficient parallel functional programming with effects," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, pp. 1558–1583, 2023. [Online]. Available: https://doi.org/10.1145/3591284