Q-TranSim: Batch Quantum Circuit Simulation using Tensor Transpilation

Shui Jiang*
The Chinese University of Hong Kong
Hong Kong, China
sjiang22@cse.cuhk.edu.hk

Hengrui Chen*
The Chinese University of Hong Kong
Hong Kong, China
hrchen26@cse.cuhk.edu.hk

Tsung-Yi Ho
The Chinese University of Hong Kong
Hong Kong, China
tyho@cse.cuhk.edu.hk

Tsung-Wei Huang
University of Wisconsin-Madison
Madison, USA
tsung-wei.huang@wisc.edu

Abstract

Batch quantum circuit simulation (BQCS) is important for analyzing quantum circuits across batches of input states. However, the state-of-the-art GPU BQCS simulator, BQSim, is memory-inefficient, limiting scalability and performance on large circuits. Addressing these limitations through manual redesign is labor-intensive, as it requires developing and maintaining numerous GPU kernels and incurs substantial turnaround time. While modern machine learning compilers such as the PyTorch compiler can automatically generate optimized GPU kernels from high-level tensor programs, they cannot be directly applied to BQCS because its execution is not directly compatible with the tensor abstractions and memory layouts assumed by these compilers.

To overcome this challenge, we present Q-TranSim, a transpilation algorithm for efficient BQCS execution on GPU. Q-TranSim first generates compressed sparse quantum gates for memory efficiency. Then, it introduces a tensor program generation framework that automatically rewrites BQCS into high-level, PyTorch-based tensor programs, which are compiled to optimized GPU kernels. Evaluated on commonly used circuits from MQT-Bench, Q-TranSim achieves 14.4× lower gate memory usage and delivers an 8.3× speedup over BQSim on average. Our study also establishes BQCS as a valuable non-ML benchmark for advancing tensor optimizations in ML compilers.

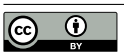
CCS Concepts

• **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Parallel computing methodologies.*

Keywords

Batch quantum circuit simulation, GPU computing, tensor compilation, sparse matrix multiplication

*Equal contribution.



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846839

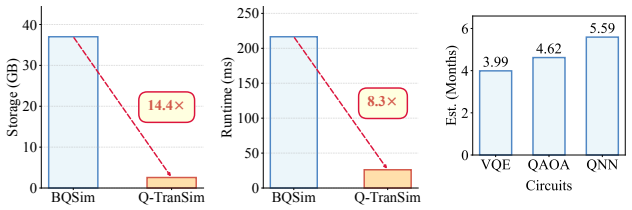
ACM Reference Format:
Shui Jiang, Hengrui Chen, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. **Q-TranSim: Batch Quantum Circuit Simulation using Tensor Transpilation**. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3838684.3846839

1 Introduction

Quantum computing (QC) has the potential to tackle certain problems (e.g., combinatorial optimization [26]) that are classically intractable. Among various QC applications [4, 63, 79, 110, 118, 136], efficient *quantum circuit simulation* (QCS) [25, 46, 64, 137] plays a crucial role, as it allows researchers to simulate a quantum algorithm on a classical computer without the need to access high-cost quantum computers. Many simulation-driven applications, such as verification [4], testing [122, 123], and state analysis [41, 67], require simulating multiple inputs to analyze circuit behavior under different scenarios. In these applications, thousands of input states are processed in batches, a paradigm referred to as *batch quantum circuit simulation* (BQCS) [2, 63].

To enable BQCS, cuQuantum [2] provides the `custatevecApply- MatrixBatched` API [105], which simulates batched state vectors through data parallelism on GPU. However, this API operates at the quantum gate level and lacks circuit-level optimizations such as gate fusion [40]. BQSim [63] further accelerates BQCS by introducing a BQCS-aware gate fusion algorithm together with a GPU-accelerated simulation kernel based on a sparse gate representation. Despite these improvements, BQSim still incurs substantial memory redundancy. Specifically, although each gate is stored in a sparse representation, it is expanded over the full n-qubit index space of the circuit, even when it acts on only a subset of qubits. As a result, this redundancy may lead to frequent CPU-GPU data movement for large circuits, limiting scalability and degrading end-to-end performance.

To address BQSim's limitations, an intuitive approach is to further compress the sparse gate representations by storing only the qubits involved in each gate. However, this approach creates substantial variation in gate structure, which a fixed GPU kernel struggles to handle efficiently, since a static implementation cannot fully utilize GPU resources across different qubit subsets and sparsity patterns. Achieving high performance therefore requires specialized kernels tuned to each gate on each GPU model, involving



(a) Gate memory usage **(b) Runtime** **(c) Kernel dev effort**

Figure 1: Comparison of gate memory usage and runtime between Q-TranSim and BQSim [63], along with the estimated development effort required for Q-TranSim's high-performance kernels across representative circuits. Q-TranSim achieves a 14.4× reduction in gate memory usage and an 8.3× speed-up over BQSim on average, while developing such optimized kernels would otherwise require a large amount of development effort.

repetitive manual engineering: profiling, adjusting thread block configurations, and managing memory hierarchies, etc. This process is time-consuming and imposes a high maintenance burden, particularly when kernels must be rewritten or retuned for different quantum circuits and GPU architectures.

On the other hand, recent success in machine learning (ML) compiler stacks [1, 3, 14, 124] have streamlined GPU kernel development by automatically lowering high-level tensor programs into specialized kernels. For example, the PyTorch compiler can *transpile* (source-to-source compile) user code into optimized Triton kernels on GPUs while applying optimizations such as kernel fusion [121] and autotuning over kernel parameters [114]. This approach substantially reduces the need for manual kernel engineering, so that developers can generate highly optimized programs with quick turnaround time [138]. Inspired by this success, we aim to optimize BQCS by transpiling a quantum circuit into a high-level tensor program, specifically targeting the PyTorch compiler to automatically generate gate-specialized GPU kernels.

However, applying PyTorch to BQCS is not straightforward. First, because BQSim expands each gate over the full n-qubit index space, applying a gate involving only a small subset of qubits introduces substantial memory overhead. While we propose a compressed representation that stores only the qubits involved in each gate, maintaining coalesced memory access still requires a carefully designed memory layout, making its integration into the full BQCS pipeline non-trivial. Second, since each gate operates on a different subset of qubits, the batch tensor must be dynamically permuted to bring the target qubits into contiguous dimensions for efficient kernel execution, incurring non-trivial data-movement overhead. Third, applying compressed sparse gates onto the batch tensor requires sparse matrix–matrix multiplication (SpMM), which cannot be easily optimized by the PyTorch compiler; this is because existing SpMM kernels often rely on indirect indexing, dynamic loops over non-zeros, and unstructured control flow, which trigger graph breaks in PyTorch [124], leading to redundant intermediate tensors and limiting opportunities for effective kernel fusion.

To overcome these challenges, we present Q-TranSim, a transpilation framework that automatically converts BQCS into tensor programs, enabling the PyTorch compiler to generate high-performance GPU kernels. The key novelty lies in bridging the semantic gap between quantum circuit operations and GPU tensor abstractions, eliminating the need for handcrafting difficult GPU kernels for each new gate, circuit topology, or GPU architecture. Our technical contributions are as follows:

- We propose a BQCS-inspired compressed sparse representation (BCSR) built on ELL format [95] to significantly reduce quantum gate storage overhead. Based on this representation, we introduce a Kronecker-product-based algorithm to construct compressed sparse gates.
- We introduce a tensor program generation framework that automatically rewrites BQCS of compressed sparse quantum gates into high-level PyTorch tensor programs.
- Within this framework, we introduce a tensor rearrangement algorithm that restructures the batch state tensor into a memory-efficient layout for SpMM on GPU.
- To further enhance the performance, we introduce an SpMM algorithm that applies compressed sparse gates to the restructured batch tensor, facilitating kernel fusion and reducing data-movement overhead.

We evaluate the performance of Q-TranSim on a set of commonly used quantum circuits selected from MQT-Bench [109]. Compared to BQSim, Q-TranSim achieves 14.4× lower average gate memory usage (Figure 1a) and is 8.3× faster on average (Figure 1b). Additionally, Q-TranSim can automatically generate these high-performance kernels for representative circuits such as VQE, QAOA, and QNN, which would otherwise require several months of manual development effort (Figure 1c). To further highlight the benefits of our transpilation, we compare Q-TranSim against (i) a hybrid baseline that selects state-of-the-art kernels for both tensor rearrangement and SpMM atop our BCSR, and (ii) a direct transpilation (D-Tran) baseline without optimizations. Q-TranSim outperforms these baselines by 1.97× and 7.02× on average, respectively. We plan to open the source of Q-TranSim to facilitate interdisciplinary resaerch between ML compiler and quantum computing.

2 Background

In this section, we provide an overview of QCS. We then summarize key concepts of modern ML compilers and their applications.

2.1 Quantum Circuit Simulation

The goal of QCS is to compute the final state obtained by applying all quantum gates in a given quantum circuit to an initial state [39, 64, 136]. An n-qubit state is represented as a complex vector $\mathbf{a} = (a_i)^T$ of length 2^n, where each entry a_i corresponds to the amplitude of a basis state. Each quantum gate is represented as a unitary matrix applying to one or more qubits. Applying a gate is equivalent to multiplying the state vector by the gate matrix.

For example, a single-qubit gate $U = (u_{ij})$ on qubit k updates amplitude pairs differing only at bit k in state vector $\mathbf{a}$:

$$\begin{pmatrix} a'_{*\cdots*0_k*\cdots*} \\ a'_{*\cdots*1_k*\cdots*} \end{pmatrix} = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix} \cdot \begin{pmatrix} a_{*\cdots*0_k*\cdots*} \\ a_{*\cdots*1_k*\cdots*} \end{pmatrix} \tag{1}$$

Similarly, for a two-qubit controlled gate $V = (v_{ij})$ with control qubit c and target qubit t, the update occurs only on the subspace where the control bit is 1:

$$\begin{pmatrix} a'_{*1_c*\cdots*0_t*\cdots*} \\ a'_{*1_c*\cdots*1_t*\cdots*} \end{pmatrix} = \begin{pmatrix} v_{00} & v_{01} \\ v_{10} & v_{11} \end{pmatrix} \cdot \begin{pmatrix} a_{*1_c*\cdots*0_t*\cdots*} \\ a_{*1_c*\cdots*1_t*\cdots*} \end{pmatrix}. \quad (2)$$

To enhance QCS performance, prior work has explored optimizations targeting both memory footprint and runtime. Previous studies [129, 134] explore data compression to simulate larger circuits with more qubits, though this often introduces approximation errors resulting in fidelity loss. Prior work [140] employs decision diagrams to leverage sparsity and structural regularity; however, this approach is effective only for regular circuits and can incur exponential overhead for irregular ones. QTask [46] utilizes task-parallel decomposition to expose inter- and intra-gate parallelism, but caching intermediate results introduces significant memory overhead. While prior GPU-accelerated simulators achieve high performance through architecture-specific optimizations [68, 130, 136, 137], they often require hardware-specific calibration. Another common optimization is gate fusion, where consecutive gates are merged to reduce passes over the 2^n-sized state vector [40, 64, 136]. However, dense fusion scales poorly: an m-qubit fused gate requires a $(2^m, 2^m)$ matrix, yielding diminishing returns once $m \approx 10$ [40, 136]. Moreover, applying fused gates often requires tensor rearrangement of the state vector, incurring non-trivial overhead [136].

Various QCS paradigms have also been developed, including unitary simulation [140], state-vector simulation [136], tensor-network simulation [127], and density-matrix simulation [78]. While effective, these methods are largely limited to the strong scaling within a *single input*. To address workflows requiring multiple inputs, such as verification [4], testing [122, 123], and state analysis [41, 67], BQCS was recently proposed by BQSim [63]. BQCS executes the same circuit over a batch of input states. To make the most of data parallelism, BQSim stores fused gates in sparse representation over the full n-qubit index space, enabling the fusion of larger gate sequences while avoiding tensor rearrangement. However, this representation becomes redundant when gates target only a subset of qubits, incurring significant memory overhead that limits scalability and performance.

2.2 ML Compilers and their Applications

To optimize large ML models on various hardware platforms, the machine learning community has developed a rich set of compiler stacks [80]. Frameworks like JAX and PyTorch leverage backends such as XLA [3] and TorchInductor [1] to automatically generate efficient CPU or GPU code. Standalone compilers like TVM [14] and infrastructure like MLIR [73] further enable performance-portable code generation across CPUs, GPUs, and accelerators (e.g., FPGA, ASIC). These compilers employ various optimization techniques, including kernel fusion [121], memory latency hiding [14], loop tiling and unrolling [100], and autotuning over kernel parameters [114]. Meanwhile, extensive academic research has advanced the compilation and code generation for specific operators within these stacks, such as SpMM [128], attention [94], and general matrix multiplication (GEMM) [69]. While these works bring significant improvement in terms of both kernel efficiency and developer's productivity, they are typically designed for ML workloads (e.g., LLM inference).

Beyond ML workloads, these compilation frameworks are increasingly adopted in non-ML domains characterized by similar underlying tensor operations to leverage automated optimization. For instance, a prior study [107] proposes a transpilation framework that maps digital circuits from hardware description languages (HDL) to neural network graphs in PyTorch. However, this approach is not performance-driven and overlooks hardware-specific optimizations during transpilation. Similarly, prior work [115] explores the use of deep learning compilers for signal processing kernel generation, yet the method struggles with memory efficiency. In scientific computing, an earlier study [116] evaluates the performance of various ML compilers on computational physics problems (e.g., N-body molecular simulation), though the study is limited to multi-core CPU architectures. Most relevantly, recent work has explored applying ML compiler stacks to QCS. Qandle [117] builds on the PyTorch compiler stack, but only expresses quantum operations as generic tensor operators without exploring quantum-specific structure. Specifically, it does not leverage gate sparsity, generate hardware-aware tensor programs, or optimize memory layouts for state tensors.

With the above discussions, we argue that there is a need for a BQCS transpilation algorithm that integrates quantum-aware compressed sparse representations with ML compiler optimizations to achieve both memory efficiency and high performance.

3 Algorithm

In this section, we discuss the technical details of our Q-TranSim algorithm. Figure 2 presents the overview of Q-TranSim. Our main contribution is the transpilation algorithm highlighted in the grey box. Q-TranSim takes an optimized quantum circuit as input (e.g., after BQSim gate fusion), and compresses the gates in the circuit into BCSR (Section 3.1). It then performs adaptive batch size planning to saturate GPU memory and thereby improve end-to-end efficiency (Section 3.2). Finally, it constructs a tensor program by iterating over gates and generating high-level tensor code for tensor rearrangement and ELL-based SpMM (Section 3.3). This tensor program is compiled by the PyTorch compiler to produce customized, high-performance Triton kernels for BQCS.

Throughout this paper, n denotes the total number of qubits in a circuit, B denotes batch size, and G denotes a quantum circuit, represented as an ordered sequence of gates.

3.1 Compressed Sparse Gates

To enhance memory efficiency for BQCS, Q-TranSim proposes BCSR for quantum gates. Unlike conventional CSR-based sparse gate matrices that implicitly retain the full n-qubit space, BCSR compresses away dimensions associated with uninvolved qubits. As a result, it achieves a more compact representation for gates applying to only a subset of qubits. Section 3.1.1 details this representation, and Section 3.1.2 presents the algorithm for generating it.

3.1.1 BCSR of Quantum Gates. Although BQSim adopts a sparse gate representation and significantly reduces memory, it can still

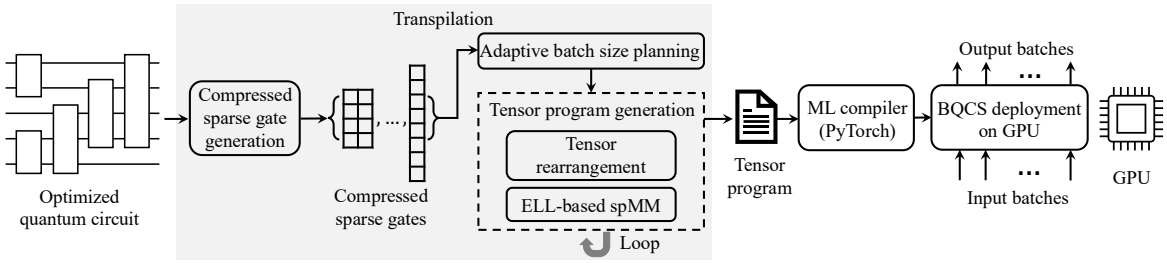


Figure 2: Overview of Q-TranSim: Starting from an optimized quantum circuit, we (i) generate compressed sparse quantum gates in BCSR, (ii) apply adaptive batch size planning to saturate GPU memory and improve BQCS efficiency, (iii) construct a tensor program by looping over gates and generating high-level tensor code for tensor rearrangement and ELL-based SpMM. The tensor program is then compiled with an ML compiler (PyTorch compiler) to generate customized high-performance GPU kernels for efficient BQCS deployment.

include redundancy. In particular, BQSim's sparse encoding implicitly spans the full n-qubit index space, even when a gate acts on only a subset of the n qubits.

To eliminate this redundancy, we propose BCSR that stores only the qubits involved in each gate. For example, Figure 3 shows an optimized 3-qubit gate applying to qubits {0,1,3} in a 4-qubit system. BQSim's ELL-based format treats the gate as a $(2^4, 2^4)$ operator and thus store data for all four qubits, even though qubit 2 is unused. Assuming the dense representation has a maximum number of non-zeros per row (NZR) of 2, BQSim stores non-zero values and their column indices in two ELL matrices, each of shape $(2^4, 2)$, as illustrated in Figure 3a. In contrast, Q-TranSim's BCSR eliminates the redundant dimension corresponding to unused qubits (e.g., qubit 2 in Figure 3). Consequently, it stores the two ELL matrices of shape $(2^3, 2)$, as illustrated in Figure 3b, yielding a 2× reduction in memory usage compared with BQSim.

More generally, for a quantum circuit G, the memory required to store gates under BQSim's sparse representation is given by

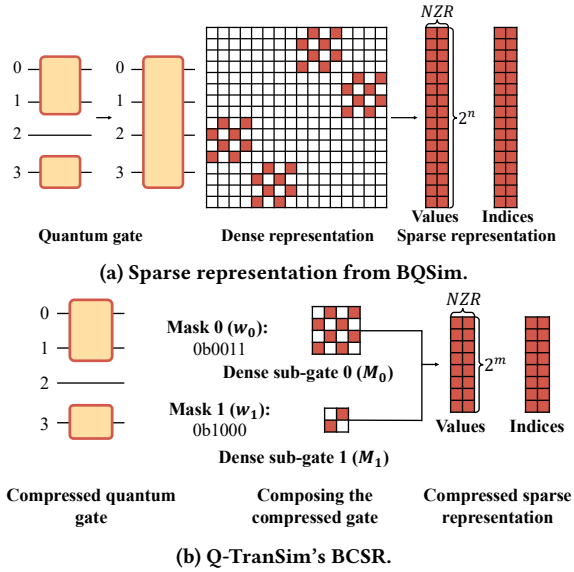
$$S = (sc + si) \cdot 2^n \sum_{i=0}^{|G|-1} NZR_i, \tag{3}$$

whereas Q-TranSim's BCSR reduces this to

$$S_{\text{comp}} = (sc + si) \sum_{i=0}^{|G|-1} 2^{m_i} NZR_i. \tag{4}$$

$|G|$ is the number of gates, m_i is the number of qubits for gate i. sc and si denote the byte sizes of a complex number and an integer, respectively.

3.1.2 BCSR Generation. Algorithm 1 shows how Q-TranSim constructs a compressed sparse gate represented in BCSR from a set of qubit-contiguous sub-gates. Given an optimized gate, we first partition its involved qubits into contiguous groups $Q = \{Q_i\}$ and obtain the corresponding sub-gate matrices $M = \{M_i\}$ (line 1). We then form sequence S_Q by taking the union of qubits appearing in all sub-gates and sorting them from high to low (line 2); the total number of involved qubits is $m = |S_Q|$ (line 3). For example, in Figure 3b, we have $Q = \{Q_0, Q_1\} = \{\{0, 1\}, \{3\}\}$, corresponding to sub-gates $M = \{M_0, M_1\}$. To distinguish circuit qubit indices from compressed-gate bit positions, we define $\pi(q)$ as the



(a) Sparse representation from BQSim.

(b) Q-TranSim's BCSR.

Figure 3: Sparse representation vs. BCSR for quantum gates.

bit position assigned to original qubit q in a compressed index. Specifically, if $S_Q = (s_0, \ldots, s_{m-1})$ is ordered from high to low, then $\pi(s_t) = m - 1 - t$, where bit position 0 is the least significant bit. Thus, for the example above, $S_Q = (3, 1, 0)$ and $\pi(3) = 2$, $\pi(1) = 1$, and $\pi(0) = 0$. The four-bit masks shown in Figure 3b illustrate qubit selection in the original circuit index space; Algorithm 1 does not apply those masks directly to compressed indices, but instead gathers the corresponding compressed positions through π. Next, we allocate the output ELL-style sparse storage: value matrix V_{comp} and column index matrix C_{comp}, and initialize K as a per-row write pointer that records the next free column position in $V_{\text{comp}}[r, :]$ and $C_{\text{comp}}[r, :]$ (lines 5–6).

The core of Algorithm 1 is a recursive function `BCSRGen` parameterized by a recursion depth idx and the current row/column indices r, c in the compressed gate (line 7). Intuitively, the recursion builds the row and column indices bit by bit over the m involved qubits. At each level, `BCSRGen` branches over the two choices for

the next row bit $j \in \{0,1\}$ and the next column bit $k \in \{0,1\}$, appends those bits to r, c, and descends to the next level (lines 21–23). The recursion starts with BCSRGen$(0,0,0)$ and has depth m; when $idx = m$, it reaches a leaf corresponding to one specific pair (r, c) in the compressed gate (line 8).

At a leaf, BCSRGen computes the value v for (r, c) by combining all sub-gates via the Kronecker-product rule [126]. For each sub-gate i, the algorithm visits its original qubits in high-to-low order and gathers the corresponding bits of r and c through π to form the sub-gate-local indices (r'_i, c'_i) (lines 11–14). For example, M_0 gathers compressed bit positions $\pi(1) = 1$ and $\pi(0) = 0$, whereas M_1 gathers position $\pi(3) = 2$. The algorithm then multiplies $M_i[r'_i, c'_i]$ into v (line 15). Finally, Algorithm 1 writes v and c to the next slot in row r of V_{comp} and C_{comp} using $K[r]$, and increments $K[r]$ (lines 17–18).

Algorithm 1 BCSR generation (BCSRGen)

1: **Global** $Q = \{Q_i\}, M = \{M_i\}$: Q_i, M_i are the qubit set and dense matrices of i-th qubit-contiguous sub-gate.
2: **Global** S_Q = ReverseSort(Sequence($\bigcup_i Q_i$)): the sequence of qubits from all sub-gates, sorted from high to low.
3: **Global** $m = |S_Q|$: the number of qubits in the output compressed sparse gate.
4: **Global** $\pi(S_Q[t]) = m - 1 - t$: mapping from an original qubit to its compressed-gate bit position; Bit(x, p) returns bit p of x.
5: **Global** $V_{\text{comp}}, C_{\text{comp}}$: the value and column index matrices of the output compressed sparse gate.
6: **Global** K: a per-row write pointer that records the next free column position in $V_{\text{comp}}[r, :]$ and $C_{\text{comp}}[r, :]$.
7: **Function** **BCSRGen**(idx: current call index, r, c: current row and column index of the output gate in BCSR):
8: **if** $idx == m$ **then**
9: $v = 1$
10: **for** $i \in \{0, \cdots, |Q| - 1\}$ **do**
11: $r'_i \leftarrow 0$; $c'_i \leftarrow 0$
12: **for** $q \in$ ReverseSort(Q_i) **do**
13: $r'_i \leftarrow 2r'_i + \text{Bit}(r, \pi(q))$; $c'_i \leftarrow 2c'_i + \text{Bit}(c, \pi(q))$
14: **end for**
15: $v \leftarrow v \cdot M_i[r'_i, c'_i]$
16: **end for**
17: $V_{\text{comp}}[r, K[r]] \leftarrow v$; $C_{\text{comp}}[r, K[r]] \leftarrow c$
18: $K[r] \leftarrow K[r] + 1$
19: **return**
20: **end if**
21: **for** $j \in \{0, 1\}$ **do**
22: **for** $k \in \{0, 1\}$ **do**
23: **BCSRGen**(idx+1, $2r + j$, $2c + k$)
24: **end for**
25: **end for**

3.2 Adaptive Batch Size Planning

To saturate GPU memory across different circuit sizes and thereby improve end-to-end efficiency, Q-TranSim plans the batch size B adaptively. Gate tensors are shared across all states in a batch, so the cost of loading and storing gates is independent of B. Therefore, choosing the largest feasible batch amortizes this fixed gate memory traffic over more states and improves end-to-end efficiency. We compute the maximum batch size $B_{\max}$ under a runtime memory budget:

$$2^n B_{\max} \cdot N \cdot sc \; + \; (sc + si) \sum_{i=0}^{|G|-1} 2^{m_i} \cdot NZR_i \; = \; C_{\text{avail}} \qquad (5)$$

In Equation 5, C_{avail} is the GPU memory available at planning time. Since fragmentation and allocator state may reduce the usable capacity, we measure C_{avail} at runtime. N denotes the number of batch copies required during execution (e.g., to hold intermediate tensors). sc and si refer to the byte sizes of a complex number and an integer, respectively. The second term accounts for the memory footprint of the compressed sparse gates (Equation 4). Throughout the paper, unless stated otherwise, the batch size refers to $B_{\max}$ computed here.

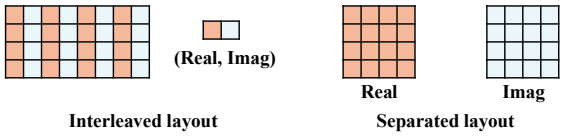


Figure 4: Different Complex tensor layout strategies in GPU memory.

3.3 Tensor Program Generation

The goal of tensor program generation is to rewrite BQCS of compressed sparse quantum gates into high-level, PyTorch-compatible tensor programs for PyTorch compilation, which in turn produces high-performance GPU kernels for BQCS. Listing 1 shows the overall workflow. For each gate g, we (i) rearrange the batch tensor into a layout aligned with g's target qubits, (ii) apply SpMM to g and the batch tensor, and (iii) rearrange the batch tensor back to the original layout. Section 3.3.1 introduces the separated memory layout for complex tensors. Section 3.3.2 describes the tensor program generation for tensor rearrangement (lines 3 and 5 in Listing 1), and Section 3.3.3 presents SpMM generation (line 4).

```
1 prog = []
2 for i, g in enumerate(gates):
3   prog+=rearrange_gen(batch, g['qubits'], B, 'fwd'
      )
4   prog+=spmm_gen(g, i)
5   prog+=rearrange_gen(batch, g['qubits'], B, 'bwd'
      )
```

Listing 1: Tensor program generation

3.3.1 Separated Memory Layout for Complex Tensor. To enhance BQCS efficiency, we adopt a separated memory layout for complex tensors. BQSim uses an interleaved layout (Figure 4), where the real and imaginary parts are packed into a `float2`. In Q-TranSim, we instead store the real and imaginary components in two separate FP32 tensors for two reasons. First, PyTorch does not reliably generate optimized code for complex-valued operators [108], which limits compiler optimizations such as kernel fusion. Second, for

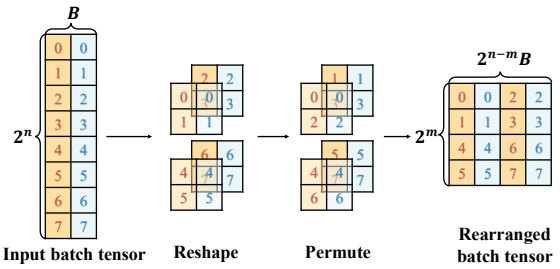


Figure 5: Rearrangement of a batch tensor (forward pass).

some circuits, the imaginary component might be zero (e.g., VQE). Under a separated layout, we can prune the corresponding memory traffic and computation. Section 3.3.3 discusses this optimization in more detail.

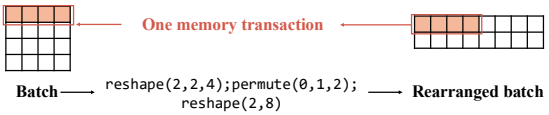
3.3.2 Tensor Rearrangement. When an m-qubit gate is applied to a batch of n-qubit state vectors ($m \le n$), the gate operates on only a subset of qubits, whereas the batch state is stored as a $(2^n, B)$ tensor. To make the gate application amenable to SpMM, we rearrange the batch so that the gate's 2^m-dimensional gate subspace become the leading row dimension. Listing 2 and Figure 5 show how we generate the corresponding tensor program.

Given the gate's qubit set `qubits`, we first compute the m target tensor axes associated with those qubits (line 2) and collect the other axes (line 3). We then build a forward permutation `p_fwd` that places the target axes first, followed by the other axes, and finally the batch axis (line 4). Next, we construct a reshape specification that views the original $(2^n, B)$ batch as an $(n+1)$-order tensor of shape $(2, 2, \cdots, 2, B)$ (line 6). This enables a standard reshape–permute–reshape pipeline.

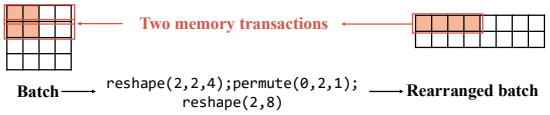
Because rearrangement is used both before and after SpMM (lines 3 and 5 in Listing 1), `rearrange_gen` supports a forward and a backward mode. In the forward pass, each of the real and imaginary tensors is reshaped to $(2, 2, \ldots, 2, B)$, permuted by `p_fwd`, and reshaped into a 2D tensor of shape $(2^m, 2^{n-m}B)$, so that the number of rows matches the gate dimension (lines 8–9 and Figure 5). In the backward pass, we invert the forward permutation to obtain `p_bwd`, reshape the tensor back to $(2, 2, \cdots, 2, B)$, apply `p_bwd`, and finally reshape back to $(2^n, B)$, restoring the original layout.

```
1  def rearrange_gen(batch, qubits, B, d='fwd'):
2    prog=[]; tgt=sorted([n-1-q for q in qubits])
3    other=[d for d in list(range(n)) if d not in tgt
       ]
4    p_fwd = tgt + other + [n] # fwd
5    l_tgt = len(tgt); l_other = len(other)
6    res_str=','.join(['2'] * n + [str(B)])
7    if d == 'fwd':
8      prog+=[f'  re=re.reshape({res_str}).permute({
       p_fwd}).reshape({2**l_tgt}, {(2**l_other)*B})'
       ]
9      prog+=[f'  im=im.reshape({res_str}).permute({
       p_fwd}).reshape({2**l_tgt}, {(2**l_other)*B})'
       ]
10   else: # bwd
11     p_bwd = [0] * (n + 1)
12     for pos, d in enumerate(p_fwd):
13       p_bwd[d] = pos
14     prog+=[f'  re=re.reshape({res_str}).permute({
       p_bwd}).reshape(-1, {B})']
15     prog+=[f'  im=im.reshape({res_str}).permute({
       p_bwd}).reshape(-1, {B})']
16   return prog
```

Listing 2: Tensor rearrangement program generation



(a) Innermost-batch permutation: keeps the batch axis innermost, yielding coalesced accesses.



(b) Alternative permutation: batch axis is not innermost, leading to less coalesced accesses.

Figure 6: GPU memory efficiency of tensor rearrangement enabled by innermost-batch permutation.

Note that in the forward output (and the corresponding backward input), the batch index is always placed in the innermost dimension. Although there are multiple valid permutation choices that map $(2^n, B)$ to $(2^m, 2^{n-m} \cdot B)$ while matching the m gate qubits, we apply an optimization named *innermost-batch permutation*, which keeps the batch dimension innermost, ensuring contiguous memory access. This optimization is critical for memory efficiency: it promotes coalesced accesses and reduces global memory (DRAM) traffic.

Figure 6 illustrates this effect using a rearranged $(2, 8)$ batch tensor, where its original shape is $(4, 4)$ and its batch size is 4. Applying the permutation $(0, 1, 2)$ preserves the batch axis as the innermost (contiguous) dimension (Figure 6a), whereas permutation $(0, 2, 1)$ swaps the last two axes and moves the batch axis to the middle dimension (Figure 6b). Although both permutations produce a layout compatible with SpMM, they induce different memory-access patterns. GPU memory requests are serviced in fixed-size units (i.e., transactions), and coalescing minimizes the number of memory transactions [106]. Assuming a memory transaction size of four elements, populating the first four contiguous elements in the rearranged batch tensor requires only a single memory transaction from the original tensor when the batch axis is the innermost dimension (Figure 6a). However, this requires two transactions when the batch axis is not innermost (Figure 6b), because those four elements are no longer contiguous and may span across transaction boundaries. Therefore, adopting a batch-last layout directly reduces the number of memory transactions, significantly improving memory efficiency and overall performance.

3.3.3 ELL-based SpMM.. Existing GPU-accelerated SpMM kernels [42] typically rely on irregular access patterns, such as indirect indexing and dynamic loops, to access non-zeros efficiently.

Unfortunately, these patterns often trigger *graph breaks* [124] in PyTorch, fragmenting the computation into multiple compiled regions and reducing opportunities for kernel fusion. To address this challenge, we implement SpMM using tensor code compatible with the PyTorch compiler. This enables the compiler to capture the computation, fuse it with surrounding permutation kernels, and apply in-kernel native optimizations. While the generated tensor program may appear to introduce redundant intermediate tensors at the source level, compilation can fuse these operations and eliminate temporaries, producing GPU kernels that are often faster than invoking standalone state-of-the-art SpMM kernels.

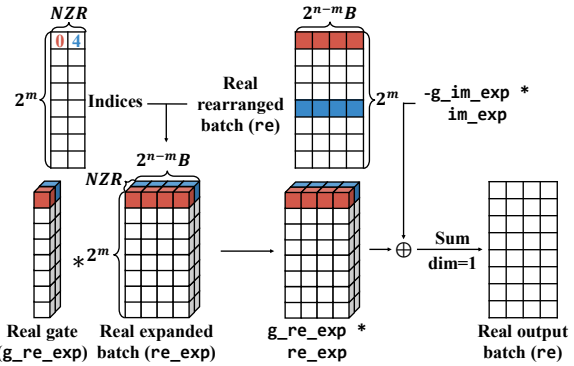
Listing 3 shows how we generate PyTorch-compatible tensor code for ELL-based SpMM. As illustrated in Figure 7a, given a rearranged batch tensor of shape $(2^m, 2^{n-m} \cdot B)$, we use the ELL column-index matrix `g['idx']` (shape $(2^m, NZR)$) to gather the NZR input entries for each output row. This expands the real and imaginary parts (`re` and `im`) into `re_exp` and `im_exp`, both of shape $(2^m, NZR, 2^{n-m}B)$ (lines 3–4). We then broadcast the gate's ELL value matrix by unsqueezing the last dimension to align with the expanded batch tensor.

```
1  def spmm_gen(g, i):
2    prog=[]
3    prog+=[f'  re_exp=re[gates[{i}]['idx']]']
4    prog+=[f'  im_exp=im[gates[{i}]['idx']]']
5    z=torch.all(g['im']==0)
6    prog+=[f'  g_re_exp=gates[{i}]['re'].unsqueeze
       (-1)'];
7    if z:
8      prog+=[f'  re=(g_re_exp * re_exp).sum(dim=1)']
9      prog+=[f'  im=(g_re_exp * im_exp).sum(dim=1)']
10   else:
11     prog+=[f'  g_im_exp=gates[{i}]['im'].unsqueeze
       (-1)']
12     prog+=[f'  re=(g_re_exp * re_exp - g_im_exp *
       im_exp).sum(dim=1)']
13     prog+=[f'  im=(g_re_exp * im_exp + g_im_exp *
       re_exp).sum(dim=1)']
14   return prog
```

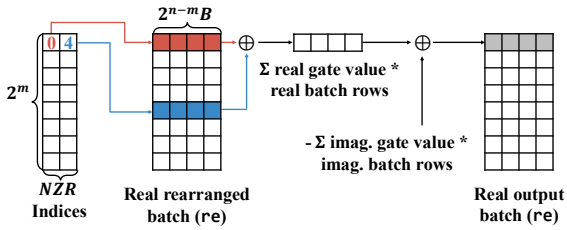
Listing 3: ELL-based SpMM program generation

Next, we specialize the generated code based on whether the gate has a non-zero imaginary component. If g is complex, we generate the code for standard complex multiply-and-accumulate, followed by a reduction `sum(dim=1)` over the NZR dimension to produce the output of shape $(2^m, 2^{n-m}B)$ (lines 10–13). This process is also shown in Figure 7a. If g is purely real (detected at compile time; line5), we omit all memory accesses and arithmetic involving the imaginary gate values and reduce to lines 7–9, pruning unnecessary memory traffic and computation in the compiled GPU kernel. For a purely real gate, both the real and imaginary components of the state are multiplied by the real gate values, so accesses to the imaginary gate tensor can be eliminated.

Figure 7a shows the SpMM computation as expressed in our generated high-level tensor code. In practice, however, ML compilation eliminates these apparent intermediates: the expanded tensor (`re_exp`, `im_exp`) is not materialized in GPU memory. Instead, the compiler lowers the gather-and-reduction pattern into



(a) Transpiled high-level tensor code for ELL-based SpMM.



(b) ML-compiler-generated GPU kernel for ELL-based SpMM.

Figure 7: ELL-based SpMM: comparison between transpiled high-level tensor code and the compiler-generated GPU kernel.

a fused loop that performs indexed scalar–vector multiply-and-accumulate, where each scalar is a nonzero gate value and each vector is the corresponding row slice from the rearranged batch tensor (Figure 7b).

3.3.4 Compiler-generated Triton Kernel. A key advantage of Q-TranSim is that it jointly optimizes memory layout and generates high-level tensor code for SpMM, while avoiding graph breaks. This enables fusing multiple tensor operations into a single kernel application, substantially improving performance. In practice, three steps are typically fused: (i) backward rearrangement of the previous batch state, (ii) forward rearrangement of the current batch state, and (iii) the SpMM corresponding to the current gate.

As an example, consider a quantum-circuit QNN with $n = 22$ qubits, where a gate g1 applied to qubits $\{0, 1\}$ is followed by a gate g2 applying to qubits $\{0, 1, 2, 3\}$. Listing 4 shows the transpiled tensor program implementing the backward rearrangement for g_1, the forward rearrangement for g_2, and the SpMM for g_2.

```
1  ...
2  # g1's backward rearrangement
3  re = re.reshape(2,2,...,2,2,148).permute([2, 3,
     ..., 0, 1, 22]).reshape(-1, 148)
4  im = im.reshape(2,2,...,2,2,148).permute([2, 3,
     ..., 0, 1, 22]).reshape(-1, 148)
5  # g2's forward rearrangement
6  re = re.reshape(2,2,2,2, ...,148).permute([18,
     19, 20, 21, ..., 22]).reshape(16, 38797312)
7  im = im.reshape(2,2,2,2, ...,148).permute([18,
     19, 20, 21, ..., 22]).reshape(16, 38797312)
```

```
  # g2's SpMM
  re_exp = re[g2['cols']]
  im_exp = im[g2['cols']]
  g_re_exp = g2['re'].unsqueeze(-1)
  g_im_exp = g2['im'].unsqueeze(-1)
  re = (g_re_exp*re_exp-g_im_exp*im_exp).sum(dim
    =1)
  im = (g_re_exp*im_exp+g_im_exp*re_exp).sum(dim
    =1)
  ...
```

Listing 4: Generated tensor program snippet: Backward rearrangement of g1–forward rearrangement of g2–SpMM of g2.

Listing 5 shows the corresponding Triton GPU kernel produced by the PyTorch compiler from Listing 4. Importantly, the three stages in Listing 4 are fused into a single kernel invocation, avoiding materialization of intermediate rearrangement tensors and explicit expanded-state tensors. While the final kernel is generated by the compiler, it is enabled by Q-TranSim's transpilation, which performs careful memory-layout planning and generates fusion-friendly tensor code across reshapes, permutations, gathers, and SpMM. This example demonstrates the effectiveness of Q-TranSim's transpilation in enabling the compiler to generate a customized, high-performance GPU kernel for BQCS.

```
@triton.jit
def triton_poi_fused(in_ptr0, in_ptr1, in_ptr2,
    in_ptr3, in_ptr4, out_ptr0, out_ptr1, xnumel,
    XBLOCK: tl.constexpr):
  xoffset = tl.program_id(0) * XBLOCK
  xindex  = xoffset + tl.arange(0, XBLOCK)[:]
  x1 = xindex // 38797312
  x0 = xindex %  38797312
  ...
  # load data from compressed sparse gate
  tmp0 = tl.load(in_ptr0 + 4 * x1, eviction_policy
    ="evict_last") # g2['re'][0][:]
  tmp1 = tl.load(in_ptr1 + 4 * x1, eviction_policy
    ="evict_last") # g2['cols'][0][:]
  tmp9 = tl.load(in_ptr3 + 4 * x1, eviction_policy
    ="evict_last") # g2['im'][0][:]
  tmp13 = tl.load(in_ptr0 + 4 * (1 + 4*x1),
    eviction_policy="evict_last") # g2['re'][1][:]
  # same pattern repeats for other terms
  ...
  # index bounds check (same pattern repeats for
    other terms)
  tmp5 = tl.where(tmp1 < 0, tmp1 + 16, tmp1)
  tl.device_assert((0 <= tmp5) & (tmp5 < 16), "
    index out of bounds: 0 <= tmp5 < 16")
  ...
  # Computed offset is fused with permute/reshape;
     no intermediate tensor is materialized.
  a=tl.load(in_ptr2 + (148*(((tmp5//4)%4)) + 592*(
    x0//148) + 155189248*(tmp5%4) + (x0%148)))
  b=tl.load(in_ptr4 + (148*(((tmp5//4)%4)) + 592*(
    x0//148) + 155189248*(tmp5%4) + (x0%148)))
  ...
  # scalar-vector multiply-and-accumulate
  real = tmp0 * a - tmp9 * b
  imag = tmp0 * b + tmp9 * a
  # same pattern repeats for other terms
  ...
  tl.store(out_ptr0 + xindex, real) # output re
  tl.store(out_ptr1 + xindex, imag) # output im
```

Listing 5: Excerpt of the compiler-generated Triton kernel for Listing 4 (omitting repetitive lines).

4 Experimental Evaluation

We evaluated the performance of Q-TranSim on 11 quantum circuits selected from MQT-Bench [109], which are representative of different quantum applications, including machine learning and optimization. Section 4.1 introduces the simulator baselines used in this study. Section 4.2 presents an overall runtime comparison across simulators and analyzes compilation overhead and sensitivity to batch size. Section 4.3 compares the memory usage of quantum circuits between our BCSR and the sparse representation used in BQSim [63]. In Section 4.4, we analyze the speed-up achieved by Q-TranSim, separating the contributions from GPU kernel execution and from overhead outside the kernels. In Section 4.5, we analyze the development effort required for the kernels generated by Q-TranSim.

For each simulation batch, we report the average steady-state runtime over five runs. Unless otherwise stated, reported execution times exclude one-time `torch.compile` tracing, Triton code generation, and compilation-time autotuning. All experiments are conducted on an Ubuntu 22.04.2 LTS system equipped with two Intel Xeon Gold 6226R CPUs at 2.9 GHz, 256 GB of memory, and one RTX 3090 GPU with 24 GB of memory. Our implementation is built on PyTorch 2.9.1+cu128, leveraging the PyTorch compiler via `torch.compile` for GPU kernel generation. All four evaluated simulators (BQSim, Hybrid, D-Tran, and Q-TranSim) produce numerically equivalent simulation results. We additionally validate their outputs against the Qiskit simulator using a relative tolerance of 10^{-3} and an absolute tolerance of 10^{-6}.

4.1 Baselines

We consider three baselines in this study:

BQSim: BQSim is the only published state-of-the-art GPU simulator specifically designed for BQCS. It employs a static SpMM kernel to simulate all quantum gates and stores the gates using a sparse representation, which still incurs substantial memory overhead. As a result, the original BQSim cannot fit circuits with more than 24 qubits into the 24 GB DRAM of the RTX 3090 GPU and fails due to memory exhaustion. To enable fair comparison with other simulators at larger scales, we use a modified BQSim baseline with host-memory-assisted execution. Specifically, unlike the original implementation, our baseline does not require all gates to reside in GPU memory: when the circuit cannot fit on the GPU, quantum gates are offloaded to CPU memory by default, and only the gate being simulated is asynchronously transferred to the GPU.

Hybrid: The Hybrid baseline combines our BCSR with two state-of-the-art, manually implemented GPU kernels: the tensor-rearrangement kernel from HyQuas [136] and the SpMM kernel from BQSim [63]. This baseline represents the hand-written

optimized-kernel axis: it uses highly optimized kernels following the rearrange-permute-rearrange steps listed in Listing 1 for the simulation of every gate. At the same time, as these kernels are selected from a finite set of pre-implemented designs rather than generated by a tensor compiler, Hybrid serves as a useful comparison for highlighting two key strengths of Q-TranSim: (i) tensor-compiler-generated kernels that adapt more effectively to structurally diverse gates, and (ii) greater opportunities for kernel fusion through tensor transpilation. It therefore evaluates whether compiler-generated fusion provides value beyond optimized, manually implemented kernels.

D-Tran: D-Tran uses the same BCSR and the same PyTorch compiler backend as Q-TranSim, but directly transpiles the computation without Q-TranSim's optimized tensor layout, innermost-batch permutation, and fusion-friendly tensor program generation. While this approach enables customized kernels for different gates, the lack of transpilation-level optimizations leads to inefficient execution. This baseline highlights the performance benefits of our optimized transpilation algorithm.

All baselines will be open-source, allowing ML compiler researchers to further enhance the performance of PyTorch stacks beyond ML. Additionally, the first two baselines are highly optimized, further showcasing the performance improvements achieved by Q-TranSim.

4.2 Overall Performance Analysis

Table 1 compares the performance of Q-TranSim, BQSim, Hybrid, and D-Tran on eleven representative circuits. As described in Section 3.2, we maximize per-batch throughput by choosing the largest batch size ($B = B_{\max}$) that fits in GPU memory. Because gate sizes differ across circuits and GPU allocations can lead to fragmentation, the feasible B can vary across simulators and workloads. To enable fair comparison under different batch sizes, we report both B and time per state (TPS), defined as the runtime per batch divided by B. This metric reflects practical deployment, where a fixed total number of states is processed over many batches.

In Table 1, Q-TranSim typically supports a larger B than Hybrid and D-Tran. For example, on Deutsch–Jozsa of $n = 23$, Q-TranSim uses $B = 85$, while Hybrid and D-Tran use $B = 83$ and $B = 25$, respectively. We attribute this to more compact intermediates: Hybrid's execution incurs additional memory allocations/copies that can introduce minor fragmentation, whereas D-Tran generates large and redundant intermediates. Specifically, for Q-TranSim, BQSim, and Hybrid, a typical intermediate has size $2^n \cdot B$. In contrast, D-Tran can create intermediates of size $2^n \cdot B \cdot NZR_i$, where NZR_i is the maximum number of non-zero elements of compressed sparse gate i, substantially increasing memory consumption. Q-TranSim's B can be lower than BQSim in some cases (e.g., 20 vs. 24 on QAOA of $n = 25$) because BQSim may offload gates to the CPU and keep only a single gate resident on the GPU, leaving more GPU memory for states.

In terms of TPS, Q-TranSim is faster than BQSim across nearly all benchmarks, with speed-ups up to 12.99× on QAOA of $n = 25$, largely because BQSim is dominated by repeated CPU–GPU memory copies when gates cannot remain on the GPU. When BQSim's gates already reside entirely in GPU memory, its CPU–GPU gate-transfer overhead is removed, so the relative performance depends more strongly on kernel efficiency and batch size. In some cases the advantage can narrow: Q-TranSim achieves a modest 1.05× speed-up on QNN of $n = 22$ and is 0.91× as fast as BQSim on VQE of $n = 22$.

Q-TranSim is consistently faster than Hybrid on all circuits, with speedups up to 2.17× on QNN of $n = 23$, which we attribute to Q-TranSim's ability to express the simulation as high-level tensor programs that enable stronger compiler optimizations and kernel fusion. Q-TranSim also consistently outperforms D-Tran, with speed-ups up to 8.92× on Deutsch–Jozsa of $n = 23$ and 7.02× on average, because D-Tran produces inefficient tensor programs with inflated intermediates and reduced kernel fusion opportunities for compilation. Because D-Tran uses the same BCSR and PyTorch compiler backend but omits Q-TranSim's transpilation-level optimizations, the 7.02× average speed-up isolates the importance of Q-TranSim's optimized tensor-program generation rather than merely the use of PyTorch compilation.

Overall, this result highlights Q-TranSim's ability to generate tensor programs compatible with the PyTorch compiler stack, which are subsequently compiled into high-performance GPU kernels. A detailed breakdown is provided in Section 4.4.

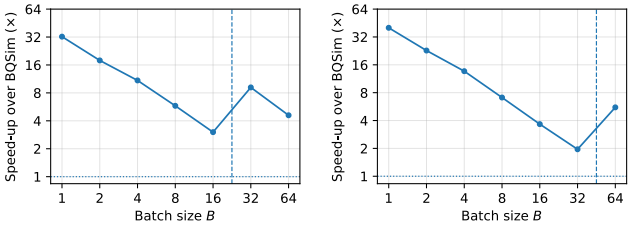
4.2.1 Compilation Overhead. Table 1 reports Q-TranSim's one-time cold compilation cost, measured with an independent compiler cache for each run and averaged over three runs. Compilation takes 26.8–90.8 s across the evaluated circuits (39.9 s at the median), while one steady-state batch takes 0.59–1.56 s. Therefore, the one-time compilation cost is equivalent to approximately 19–154 steady-state batches, with a median of 36 batches. Since BQCS workloads repeatedly execute the same circuit over many input batches, this upfront cost can be amortized over subsequent batches. The compilation time is circuit-dependent because different circuits generate tensor programs and specialized kernels of different complexity.

4.2.2 Sensitivity to Batch Size. To separate Q-TranSim's execution benefits from BQSim's GPU-memory-capacity limitation, we additionally run BQSim and Q-TranSim at identical fixed batch sizes on QAOA and Deutsch–Jozsa with $n = 23$. As in the main evaluation, TPS measures steady-state execution and excludes Q-TranSim's one-time compilation cost, which is analyzed separately above. Figure 8 reports Q-TranSim's speed-up over BQSim as B varies from 1 to 64.

When all BQSim gates reside in GPU memory, Q-TranSim remains faster. At the largest tested GPU-resident batch sizes, Q-TranSim achieves a 3.02× speed-up on QAOA ($B = 16$) and a 1.96× speed-up on Deutsch–Jozsa ($B = 32$), showing that its performance improvement does not arise solely from avoiding CPU–GPU gate transfers. Within this GPU-resident regime, BQSim becomes more efficient per state as B increases, narrowing the relative gap, while Q-TranSim's TPS remains nearly constant. Once the batch becomes large enough that BQSim can no longer keep all gates resident, BQSim switches to host-staged gate execution and its TPS increases sharply. At the first tested host-staged points, Q-TranSim achieves 9.16× speed-up on QAOA ($B = 32$) and 5.55× on Deutsch–Jozsa ($B = 64$). Thus, Q-TranSim provides both an execution-efficiency benefit in the non-memory-starved regime

Table 1: Overall performance comparison among Q-TranSim, BQSim, Hybrid, and D-Tran. Because simulators may use different batch sizes (B) for the same circuit, we report time per state (TPS), computed as runtime per batch divided by B. "Compile" reports Q-TranSim's one-time cold compilation cost, averaged over three independent cold-cache runs.

Circuits			BQSim		Hybrid		D-Tran		Q-TranSim (ours)			Q-TranSim's speed-up vs.		
Name	$\|G\|$	n	B	TPS (ms)	B	TPS (ms)	B	TPS (ms)	B	TPS (ms)	Compile (s)	BQSim	Hybrid	D-Tran
VQE	44	22	113	7.70	168	16.50	51	70.05	172	8.45	26.8	0.91×	1.95×	8.29×
	48	23	112	48.93	84	34.92	25	130.23	86	18.10	29.6	2.70×	1.93×	7.19×
	48	24	53	216.82	35	71.41	10	258.57	38	36.94	27.6	5.87×	1.93×	7.00×
	52	25	24	1000.63	17	154.23	5	504.29	19	79.09	31.5	12.65×	1.95×	6.38×
QAOA	36	23	20	19.88	85	26.29	26	103.22	87	13.40	38.2	1.48×	1.96×	7.71×
	36	24	53	157.74	38	53.22	11	192.81	40	27.71	39.9	5.69×	1.92×	6.96×
	39	25	24	759.36	19	114.45	5	394.83	20	58.47	42.0	12.99×	1.96×	6.75×
QNN	44	22	113	7.68	133	15.32	40	58.93	149	7.32	49.6	1.05×	2.09×	8.05×
	47	23	112	46.49	43	33.05	13	112.42	59	15.23	50.9	3.05×	2.17×	7.38×
Deutsch-Jozsa	23	23	52	9.33	83	15.50	25	64.57	85	7.24	79.0	1.29×	2.14×	8.92×
	24	24	53	107.38	37	32.23	11	127.37	39	15.12	90.8	7.10×	2.13×	8.42×



(a) QAOA, $n = 23$ **(b) Deutsch–Jozsa, $n = 23$**

Figure 8: Q-TranSim speed-up over BQSim at identical fixed batch sizes. The vertical dashed line marks the transition after which BQSim uses host-staged gates; points to its left keep all BQSim gates resident on the GPU. Compilation time is excluded from TPS and analyzed separately in Section 4.2.1.

Table 2: Memory usage comparison for all the gates in each circuit between BQSim's sparse gate representation and our BCSR.

Circuit	n	Sparse (BQSim, GB)	BCSR (ours, GB)	Compression ratio
VQE	22	11.81	0.81	14.67×
	23	24.70	0.81	30.67×
	24	51.54	3.22	16×
	25	107.37	3.22	33.33×
QAOA	23	18.52	0.54	34.5×
	24	38.65	2.15	18×
	25	80.53	2.15	37.5×
QNN	22	11.81	3.94	3×
	23	24.70	8.05	3.07×
Deutsch-Jozsa	23	12.08	0.89	13.5×
	24	25.23	2.51	10.07×

and an additional memory-capacity benefit when larger batches force BQSim to stage gates from host memory.

4.3 Comparison of Gate Memory Usage

Table 2 compares the memory usage (GB) of all gates across 11 circuits for BQSim's sparse representation vs. our BCSR. Our BCSR is up to 37.5× more memory-efficient than BQSim, and 19.48× more memory-efficient on average. This reduction is practically significant: besides lowering the memory budget required to store optimized circuit parameters, it can also translate into improved runtime efficiency when deploying BQCS, by keeping more gate data resident on the GPU's DRAM and reducing CPU–GPU data transfers.

In contrast, BQSim's sparse representation can require as much as 107.37 GB (VQE of $n = 25$), exceeding the memory of an RTX3090 and approaching the capacity of newer GPUs (e.g., NVIDIA H200 [101]). To run BQCS with this representation, one must either offload gates to CPU memory or shard gates across multiple GPUs for data parallelism; both options become increasingly constrained by interconnect bandwidth (PCIe or NVLink), limiting scalability to larger qubit counts. Our BCSR remains consistently small, peaking at 8.05 GB, which fits within RTX 3090 memory and is therefore more scalable.

We attribute the memory efficiency to storing only the target qubits of each gate, rather than encoding all qubits in the circuit, thereby reducing redundancy. We also observe cases where compressed storage stays roughly constant as n increases (e.g., QAOA of $n = 24$ vs. $n = 25$). This is because a larger n does not necessarily imply that each gate operates on more qubits, so BCSR can have comparable memory requirements for circuits with $n = 24$ and $n = 25$. However, the representation in BQSim cannot exploit this effect and grows exponentially with n. Overall, this result highlights the efficiency of our BCSR for storing optimized quantum gates.

4.4 Detailed Analysis of Speed-up

In this section, we study the factors behind the speed-ups reported in Table 1 in detail. Because the computation is ultimately performed by GPU kernels, overall performance depends on keeping kernels active throughout simulation and minimizing time spent

in non-kernel stalls. Moreover, kernel efficiency depends on how much time the compute units (e.g., FP32/INT cores and Tensor Cores [104]) spend performing useful work rather than waiting on memory accesses or suffering from scheduling inefficiencies. Therefore, we analyze the sources of speed-up from two perspectives: non-kernel stall (Section 4.4.1) and in-kernel details (Section 4.4.2).

4.4.1 Non-kernel Stall Analysis. Figure 9 reports the time breakdown between GPU kernel execution and non-kernel stalls for BQSim, Hybrid, D-Tran, and Q-TranSim on three circuits: VQE of $n = 23$, Deutsch–Jozsa (DJ, $n = 24$), and QNN of $n = 23$. Figure 10 shows the per-batch simulation timeline for VQE of $n = 23$ under the four simulators, generated from traces collected with NVIDIA Nsight Systems [103].

The non-kernel stalls mainly come from memory copies (`cudaMemcpy`) and GPU memory allocation (`cudaMalloc`) that extend beyond kernel execution. Although these operations are issued asynchronously to overlap with kernels, stalls can still occur due to data dependencies.

As shown in Figure 9, Q-TranSim has almost no non-kernel stalls (0.01%), and kernels are active for 99.99% of the time on all three circuits. We attribute this to our transpilation algorithm, which guides the PyTorch compiler to generate Triton kernels that avoid redundant intermediate copies and memory allocations. The remaining stalls are caused by kernel launch and synchronization overhead. This result is also observed in Figure 10d.

In Q-TranSim, tensor rearrangement does not appear as a standalone runtime component because the generated program allows the PyTorch compiler to fuse backward rearrangement, forward rearrangement, and SpMM into a single Triton kernel (Section 3.3.4). Consequently, Nsight Systems cannot attribute an independent latency to rearrangement once it has been fused. This does not imply that rearrangement has zero computational cost; rather, its operations are incorporated into the fused kernel and avoid separately materialized intermediate tensors.

In contrast, D-Tran achieves a kernel-active percentage of 81%–91.28% across the three circuits (Figure 9). Figure 10c shows that its stalls are dominated by `cudaMalloc` at the beginning of the simulation. Although D-Tran also uses PyTorch compilation, it is sub-optimized and requires large intermediate tensors to store partial results (Section 4.2). Allocating these tensors can take longer than the first kernel execution and create noticeable stalls.

Hybrid achieves a kernel-active percentage of 71.34%–92.04% across the three circuits (Figure 9). Figure 10b shows that its non-kernel stalls are dominated by frequent `cudaMemcpy` and `cudaMalloc` events throughout the simulation. This is because Hybrid executes the simulation as a sequence of library-kernel invocations with materialized intermediate tensors, which in turn triggers repeated GPU allocations, frees (`cudaFree`), and on-device memory copies.

BQSim has the lowest kernel-active percentage (15.98%–32.11%). As discussed in Section 4.1, the circuit is too large to reside in GPU memory, so gates must be repeatedly transferred from CPU to GPU. These `cudaMemcpy` operations dominate the runtime (Figure 10a). This overhead becomes more severe as n increases. For example, BQSim has a lower kernel-active percentage on DJ of $n = 24$ compared with VQE of $n = 23$ (i.e., 15.98% vs. 32.02%). Since applying

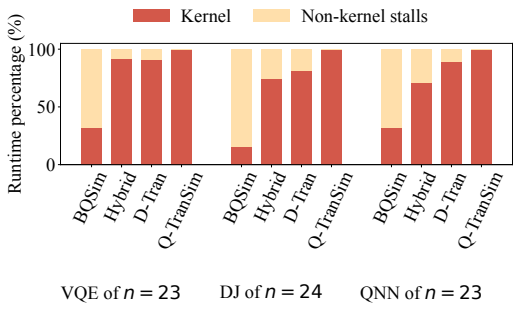


Figure 9: Percentage breakdown of kernel time and non-kernel stalls for four simulators (BQSim, Hybrid, D-Tran, and Q-TranSim) on three circuits: VQE of $n = 23$, Deutsch–Jozsa (DJ, $n = 24$), and QNN of $n = 23$.

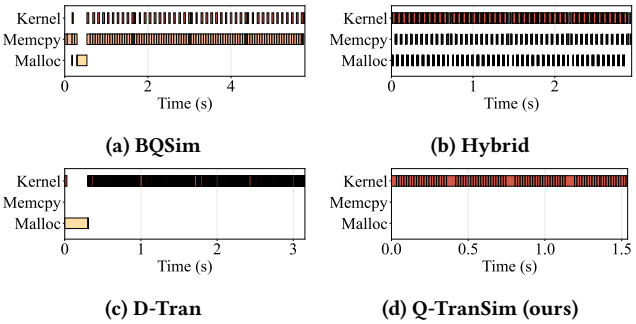


Figure 10: Simulation timeline for running a batch on VQE of $n = 23$ under four simulators (BQSim, Hybrid, D-Tran, and Q-TranSim), generated from traces collected with NVIDIA Nsight Systems [103].

sparse gates is largely memory-bound, kernel time is mostly spent moving data from DRAM to compute units on GPU, where the available bandwidth is much higher than PCIe (PCIe 3.0 x16: 15.75 GB/s vs. GPU DRAM: 936.2 GB/s). As n grows, CPU–GPU memory copy increasingly becomes the bottleneck relative to kernel execution.

4.4.2 Detailed Kernel Performance Analysis. Table 3 compares detailed kernel performance for Q-TranSim, BQSim, Hybrid, and D-Tran on three circuits: QAOA of $n = 23$, Deutsch–Jozsa (DJ, $n = 23$), and QNN of $n = 22$. All metrics are collected with NVIDIA Nsight Compute [102], outlined below:

- **DRAM access per state:** We captures the DRAM traffic required for simulating a state vector on average. This is computed using (`dram__bytes_read.sum`+`dram__bytes_write.sum`)/B.
- **L1 cache hit rate:** We report the average of `l1tex__t_sector _hit_rate.pct` [24] over all BQCS kernels. This reflects how cache-friendly the memory access pattern is.
- **FMA pipeline utilization:** We report the average of `sm__pipe_fma_cycles_active.avg.pct_of_peak_sustained_elapsed` over all BQCS kernels. Since the FMA

pipeline handles most FP32 arithmetic [102], this metric indicates how effectively each simulator uses the FP32 compute pipeline.

- **Warp divergence per state:** We compute `smsp__branch_targets_ threads_divergent`/B, which counts divergent branch targets (including fallthrough) for simulating a state vector on average. This reflects scheduling efficiency and how uniformly threads within a warp follow the same control flow.

The first two metrics characterize memory behavior, while the latter two capture compute utilization and scheduling behavior, respectively. Compared with BQSim, Q-TranSim eliminates warp divergence completely while achieving similar FMA pipeline utilization. Even though Q-TranSim can achieve a much higher L1 cache hit rate (e.g., 43.95% vs. 4.88% on QNN), it may still incur slightly higher DRAM traffic (e.g., 11.69 GB vs. 8.95 GB on QAOA), suggesting that the PyTorch-generated Triton kernels are not always DRAM-traffic-optimal (Section 4.2). Nevertheless, removing warp divergence avoids serialized control-flow paths and improves effective SIMT efficiency. We attribute this to Q-TranSim's ability to generate gate-specific kernels, whereas BQSim repeatedly reuses a fixed kernel.

Compared with Hybrid, Q-TranSim shows substantially lower DRAM access (e.g., 11.69 GB vs. 20.54 GB on QAOA), higher L1 cache hit rate (e.g., 43.95% vs. 4.85% on QNN), and zero warp divergence. This is because Hybrid selects from a pre-implemented kernel family and typically materializes intermediates between gates, which increases DRAM traffic and limits kernel fusion. In contrast, Q-TranSim can fuse kernels and generate kernels tailored to diverse gate structures, reducing intermediate reads/writes.

Compared with D-Tran, Q-TranSim dramatically reduces DRAM traffic (e.g., 6.27 GB vs. 38.46 GB on DJ), while the other metrics remain broadly comparable. D-Tran introduces large intermediate tensors and provides few opportunities for the PyTorch compiler to fuse kernels, even though the generated kernels also exhibit zero warp divergence.

Table 3: Comparison of GPU kernel metrics for Q-TranSim, BQSim, Hybrid, and D-Tran across three circuits: QAOA of $n = 23$, Deutsch–Jozsa (DJ, $n = 23$), and QNN of $n = 22$. Reported metrics include DRAM accesses per state (DRAM acc./B) and average L1 cache hit rate (memory), average FMA pipeline utilization (compute), and warp divergences per state (Warp div./B) (scheduling).

Circuit	Name	QAOA	DJ	QNN
	n	23	23	22
BQSim	DRAM acc./B (GB)	8.95	6.6	6.02
	L1 hit rate	17.92%	11.38%	4.88%
	FMA util.	8.31%	7.88%	8.43%
	Warp div./B	15099494.4	3710345.85	1633180.32
Hybrid	DRAM acc./B (GB)	20.54	9.73	9.24
	L1 hit rate	1.27%	6.33%	4.85%
	FMA util.	7.51%	7.99%	7.88%
	Warp div./B	1311491.01	859863.9	962837.17
D-Tran	DRAM acc./B (GB)	64.71	38.46	35.58
	L1 hit rate	20.49%	20.61%	20.55%
	FMA util.	4.58%	5.26%	4.99%
	Warp div./B	0	0	0
Q-TranSim	DRAM acc./B (GB)	11.69	6.27	6.3
	L1 hit rate	16.07%	18.98%	43.95%
	FMA util.	4.57%	6.57%	7.75%
	Warp div./B	0	0	0

Table 4: Estimated human effort for developing BQCS kernels for representative circuits (VQE, QAOA, and QNN), computed using SLOCCount [125].

Circuits	n	SLOC	Developers	Schedule (Months)	Cost ($)
VQE	22	1404	0.86	3.99	38581
QAOA	20	2030	1.09	4.62	56821
QNN	21	3257	1.48	5.59	93347

4.5 Analysis of Development Effort

Precisely quantifying development effort for BQCS kernels in Q-TranSim is inherently difficult, as it spans multiple teams over extended periods; instead, we approximate this cost using SLOCCount [125], a widely adopted tool for estimating software development effort. Specifically, we use SLOCCount to measure the complexity of generated kernel code by Q-TranSim for three representative circuits. Table 4 reports the estimated source lines of code (SLOC), number of developers, development schedule in months, and monetary cost. Note that we did not compare Q-TranSim with other baselines in terms of development effort, because BQSim and Hybrid rely on handcrafted kernels that do not support circuit-dependent optimization; on the other hand, D-Tran delivers relatively low performance, making such comparison less meaningful. The results in Table 4 show that manually implementing kernels for a single circuit may require up to 3257 lines of code, 1.48 developers, 5.59 months, and an estimated cost of $93347. These estimates reveal the potential development effort otherwise required to optimize BQCS kernels, while Q-TranSim can largely streamline this process through automatic transpilation and kernel generation.

5 Conclusion

In this paper, we have presented Q-TranSim, a transpilation algorithm that generates high-level, PyTorch-compatible tensor programs for BQCS, enabling the PyTorch compiler to produce optimized GPU kernels. Evaluated on circuits from MQT-Bench, Q-TranSim achieves 14.4× lower gate memory usage and delivers an 8.3× speed-up over BQSim on average. Our study further establishes BQCS as a valuable non-ML benchmark for advancing tensor optimizations in ML compilers. Inspired by our prior work [5–13, 15–23, 27–38, 43–66, 70–72, 74–77, 81–93, 96–99, 111–113, 119, 120, 131–133, 135, 139], we plan to study more applications beyond quantum computing.

Acknowledgments

This work was jointly supported by the National Science Foundation (NSF) under Grant No. 2235276, 2349144, 2349143, 2349582, and 2349141, and the Research Grants Council of Hong Kong SAR No. T46-415/25-R.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM international conference on architectural support for programming languages and operating systems, volume 2*. ACM, 929–947.

[2] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, et al. 2023. cuQuantum SDK: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 1050–1061.

[3] James Bradbury et al. 2018. JAX: Autograd and XLA. http://github.com/google/jax.

[4] Lukas Burgholzer and Robert Wille. 2020. The power of simulation for equivalence checking in quantum computing. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[5] Che Chang, Cheng-Hsiang Chiu, Boyang Zhang, and Tsung-Wei Huang. 2024. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 771–774.

[6] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[7] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[8] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2026. C-PathGen: An Efficient CPU-Parallel Critical Path Generation Algorithm. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*.

[9] Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Boyang Zhang, and Tsung-Wei Huang. 2026. SSTA-X: GPU-Accelerated First-Order Block-Based Statistical Static Timing Analysis. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*.

[10] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*. 1–7.

[11] Chih-Chun Chang and Tsung-Wei Huang. 2025. Statistical Timing Graph Scheduling Algorithm for GPU Computation. In *ACM/IEEE Design Automation Conference (DAC)*.

[12] Chih-Chun Chang, Aditya Das Sarma, Cheng-Hsiang Chiu, and Tsung-Wei Huang. 2027. Can ML Frameworks Go Beyond ML? A Case Study in GPU-accelerated VLSI Timing Analysis. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[13] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM International Conference on Parallel Processing (ICPP)*. 565–575.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. USENIX Association, 578–594.

[15] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 283–284.

[16] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*. 1388–1389.

[17] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2024. An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 766–770.

[18] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*. 468–479.

[19] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–8.

[20] Cheng-Hsiang Chiu, Chedi Morchdi, Chih-Chun Chang, Cunxi Yu, Yi Zhou, and Tsung-Wei Huang. 2025. Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*.

[21] Cheng-Hsiang Chiu, Chedi Morchdi, Yi Zhou, Boyang Zhang, Che Chang, and Tsung-Wei Huang. 2024. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[22] Yi-Hua Chung, Shui Jiang, Wan Luan Lee, Yanqing Zhang, Haoxing Ren, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

[23] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*.

[24] Paul Elvinger, Foteini Strati, Natalie Enright Jerger, and Ana Klimovic. 2025. Measuring GPU utilization one level deeper. *arXiv preprint arXiv:2501.16909* (2025).

[25] Aneeqa Fatima and Igor L Markov. 2021. Faster schrödinger-style simulation of quantum circuits. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 194–207.

[26] Felix Gemeinhardt, Antonio Garmendia, Manuel Wimmer, Benjamin Weder, and Frank Leymann. 2023. Quantum Combinatorial Optimization in the NISQ Era: A Systematic Mapping Study. *ACM Comput. Surv.* 56, 3, Article 70 (Oct. 2023), 36 pages. https://doi.org/10.1145/3620668

[27] Serhan Gener, Sahil Hassan, Liangliang Chang, Chaitali Chakrabarti, Tsung-Wei Huang, Umit Ograss, and Ali Akoglu. 2025. A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*.

[28] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[29] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. 4219–4232.

[30] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.

[31] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*. 721–726.

[32] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[33] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[34] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*. 3466–3478.

[35] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[36] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. 4973–4984.

[37] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[38] Zizheng Guo, Zuodong Zhang, Wuxi Li, Tsung-Wei Huang, Xizhe Shi, Yufan Du, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–9.

[39] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 866–874.

[40] Hiroshi Horii and Jun Doi. 2021. *Optimization of Quantum Computing Simulation with Gate Fusion.* Technical Report 2021-QS-2(23). Information Processing Society of Japan. 1–7 pages.

[41] Zhirui Hu, Peiyan Dong, Zhepeng Wang, Youzuo Lin, Yanzhi Wang, and Weiwen Jiang. 2022. Quantum neural network compression. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. ACM, 1–9.

[42] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[43] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–2.

[44] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*. 1–6.

[45] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[46] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 746–756.

[47] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[48] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[49] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*. 1360–1363.

[50] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[51] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*. 1–4.

[52] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 757–764.

[53] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*. 1–2.

[54] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 1070–1083.

[55] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. In *IEEE Design and Test (DAT)*.

[56] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 1303–1320.

[57] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[58] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*. 588–597.

[59] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*. 1–6.

[60] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*. 51–59.

[61] Shui Jiang, Hengrui Chen, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. FlatDD: Parallel Quantum Circuit Simulation using Decision Diagram and Flat Array. In *ACM Transactions on Quantum Computing (TQC)*.

[62] Shui Jiang, Wen Cheng, Yi-Hua Chung, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. QDP: Worst-case Fidelity-aware Qubit Mapping and Routing using Dynamic Programming. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*.

[63] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 79–94.

[64] Shui Jiang, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei Huang. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM International Conference on Parallel Processing (ICPP)*. ACM, 388–399.

[65] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[66] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*. 51–61.

[67] Weiwen Jiang, Jinjun Xiong, and Yiyu Shi. 2021. A co-design framework of neural networks and quantum circuits towards quantum advantage. *Nature communications* 12, 1 (2021), 579.

[68] Chenyang Jiao, Weihua Zhang, and Li Shen. 2023. Communication Optimizations for State-vector Quantum Simulator on CPU+ GPU Clusters. In *Proceedings of the 52nd International Conference on Parallel Processing*. ACM, 203–212.

[69] Hyeoncheol Kim, Taehoon Kim, Taehyeong Park, Donghyeon Kim, Yongseung Yu, Hanjun Kim, and Yongjun Park. 2025. Accelerating LLMs using an Efficient GEMM Library and Target-Aware Optimizations on Real-World PIM Devices. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 225–240. https://doi.org/10.1145/3696443.3708953

[70] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*.

[71] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 278–283.

[72] T.-Y. Lai, Tsung-Wei Huang, and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–6.

[73] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[74] Wan-Luan Lee, Shui Jiang, Dian-Lun Lin, Che Chang, Boyang Zhang, Yi-Hua Chung, Ulf Schlichtmann, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.

[75] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. 2025. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[76] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[77] Wan-Luan Lee, Aditya Das Sarma, Che Chang, Chih-Chun Chang, and Tsung-Wei Huang. 2026. iHyperG: Incremental Hypergraph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.

[78] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. Density matrix quantum circuit simulation via the BSP machine on modern GPU clusters. In *Sc20: international conference for high performance computing, networking, storage and analysis*. IEEE, 1–15.

[79] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. ACM, 1001–1014.

[80] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2021. The Deep Learning Compiler: A Comprehensive Survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2021), 708–727. https://doi.org/10.1109/TPDS.2020.3030548

[81] Zhengxiong Li, Tsung-Wei Huang, and Umit Ogras. 2026. CREDIT: Cost-guided Reduction-reuse with Efficient DSMEM Inter-CTA Tiling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[82] Zhengxiong Li, Tsung-Wei Huang, and Umit Ogras. 2026. SET: Stream-Event-Triggered Scheduling for Efficient CUDA Graph Pipelines. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

[83] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*. 2284–2287.

[84] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[85] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 64–71.

[86] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*. 183–188.

[87] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[88] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*. 435–450.

[89] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 3041–3052.

[90] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*. 151–166.

[91] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing (ICPP)*. 1–12.

[92] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[93] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[94] Zhenyu Liu, Xilang Zhou, Faxian Sun, Jianli Chen, Jun Yu, and Kun Wang. 2025. AttentionLib: A Scalable Optimization Framework for Automated Attention Acceleration on FPGA. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7. https://doi.org/10.23919/DATE64628.2025.10992963

[95] Los Alamos National Lab. 2014. ELL format. https://www.lanl.gov/Caesar/node223.html

[96] Pingchuan Ma, Ziang Yin, Qi Jing, Zhengqi Gao, Nicholas Gangi, Boyang Zhang, Tsung-Wei Huang, Rena Huang, Duane Boning, Yu Yao, and Jiaqi Gu. 2026. SP2RINT: Spatially-Decoupled Physics-Constrained Progressive Inverse Optimization for Diffractive Optical Neural Network Training. In *ACM/IEEE Design Automation Conference (DAC)*.

[97] Chedi Morchdi, Cheng-Hsiang Chiu, Tsung-Wei Huang, and Yi Zhou. 2026. SSTA-Trace: LLM-Learned Scheduling of Statistical Static Timing Graphs on GPU. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[98] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 89–95.

[99] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*.

[100] Giridhar Sreenivasa Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran, and P. Sadayappan. 2010. Optimal loop unrolling for GPGPU programs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–11. https://doi.org/10.1109/IPDPS.2010.5470423

[101] NVIDIA. 2024. NVIDIA H200 Tensor Core GPU. https://dam-cdn.nvd.orangelogic.com/AssetLink/5o2qgy5d2835ve2pm11i62kv8mphqta8.pdf.

[102] NVIDIA. 2026. NVIDIA Nsight Compute Documentation. https://docs.nvidia.com/nsight-compute/.

[103] NVIDIA. 2026. NVIDIA Nsight Systems User Guide. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[104] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf Whitepaper (Ampere architecture), accessed 2026-02-20.

[105] NVIDIA Corporation. 2024. cuStateVec Functions. NVIDIA cuQuantum 24.08.0 documentation. https://archive.docs.nvidia.com/cuda/cuquantum/24.08.1/custatevec/api/functions.html

[106] NVIDIA Corporation. 2026. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses Section "Device Memory Accesses" (coalescing and transaction sizes), accessed 2026-02-20.

[107] Devdhar Patel, Ignacio Gavier, Joshua Russell, Andrew Malinsky, Edward Rietman, and Hava Siegelmann. 2022. Automatic transpiler that efficiently converts digital circuits to a neural network representation. In *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 01–08.

[108] PyTorch contributors. 2025. PyTorch v2.9.1. https://raw.githubusercontent.com/pytorch/pytorch/v2.9.1/torch/_inductor/lowering.py.

[109] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* 7, Article 1062 (2023), 14 pages. MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[110] Gokul Subramanian Ravi, Pranav Gokhale, Yi Ding, William Kirby, Kaitlin Smith, Jonathan M Baker, Peter J Love, Henry Hoffmann, Kenneth R Brown, and Frederic T Chong. 2022. CAFQA: A classical simulation bootstrap for variational quantum algorithms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ACM, 15–29.

[111] Xiangyu Ren, Yuexun Huang, Zhaohui Yang, Yuchen Zhu, Tsung-Wei Huang, Tsung-Yi Ho, Zhiding Liang, and Antonio Barbalace. 2026. A Spatial-Topology Preserving Compiler for Quantum Many-Body Systems Simulation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[112] Aditya Das Sarma, Shui Jiang, Wan-Luan Lee, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[113] Aditya Das Sarma, Wan-Luan Lee, Shui Jiang, Boyang Zhang, and Tsung-Wei Huang. 2026. A Differentiable Approach to Task Graph Partitioning: A Case Study in RTL Simulation. In *ACM/IEEE Design Automation Conference (DAC)*.

[114] Richard Arnoud Schoonhoven, Ben van Werkhoven, and Kees Joost Batenburg. 2023. Benchmarking optimization algorithms for auto-tuning GPU kernels. *IEEE Transactions on Evolutionary Computation* 27, 3 (2023), 550–564.

[115] João Alberto Moreira Seródio. 2025. *Deep learning compilers for signal processing kernel generation*. Master's thesis. Universidade Estadual de Campinas.

[116] Niteya Shah, Pi-Yueh Chuang, Paul Sathre, and Wu-chun Feng. 2025. On the Landscape of Scientific Computing Libraries in Python. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[117] Gerhard Stenzel, Sebastian Zielinski, Michael Kölle, Philipp Altmann, Jonas Nüßlein, and Thomas Gabor. 2024. Qandle: Accelerating state vector simulation using gate-matrix caching and circuit splitting. *arXiv preprint arXiv:2404.09213* (2024).

[118] Siwei Tan, Debin Xiang, Liqiang Lu, Junlin Lu, Qiuping Jiang, Mingshuai Chen, and Jianwei Yin. 2024. MorphQPV: Exploiting Isomorphism in Quantum Programs to Facilitate Confident Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, 671–688.

[119] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei Huang. 2024. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 789–793.

[120] Jie Tong, Zhengxiong Li, Umit Yusuf Ogras, and Tsung-Wei Huang. 2025. A Scalable Code Generation Flow for Heterogeneous Parallel RTL Simulation using MLIR. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[121] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202. https://doi.org/10.1109/SC.2014.21

[122] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. Qdiff: Differential testing of quantum software stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 692–704.

[123] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Quito: a coverage-guided test generator for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1237–1241.

[124] William Wen. 2026. Introduction to torch.compile. https://docs.pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[125] David A. Wheeler. n.d.. SLOCCount. https://dwheeler.com/sloccount/.

[126] Wikipedia contributors. 2026. Kronecker product. https://en.wikipedia.org/wiki/Kronecker_product.

[127] Robert Wille, Lukas Burgholzer, Stefan Hillmich, Thomas Grurl, Alexander Ploier, and Tom Peham. 2022. The basis of design tools for quantum computing: arrays, decision diagrams, tensor networks, and ZX-calculus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. ACM, 1367–1370.

[128] Jaeyeon Won, Willow Ahrens, Joel S Emer, and Saman Amarasinghe. 2025. Insum: Sparse GPU Kernels Simplified and Optimized with Indirect Einsums. *arXiv preprint arXiv:2510.17505* (2025).

[129] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2019. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–24.

[130] Mingkuan Xu, Shiyi Cao, Xupeng Miao, Umut A. Acar, and Zhihao Jia. 2024. Atlas: Hierarchical Partitioning for Quantum Circuit Simulation on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 81, 17 pages. https://doi.org/10.1109/SC41406.2024.00087

[131] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[132] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[133] Boyang Zhang, Chih-Chun Chang, Yi-Hua Chung, Che Chang, Cheng-Hsiang Chiu, Aditya Das Sarma, and Tsung-Wei Huang. 2026. G-STAR: GPU-Accelerated Statistical Static Timing Analysis using Level-by-level Replication. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

[134] Boyuan Zhang, Bo Fang, Fanjiang Ye, Luanzheng Guo, Fengguang Song, Nathan Tallent, and Dingwen Tao. 2025. BMQSim: Overcoming Memory Constraints in Quantum Circuit Simulation with a High-Fidelity Compression Framework. In *Proceedings of the 39th ACM International Conference on Supercomputing*. ACM, 689–704.

[135] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[136] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: hybrid partitioner based quantum circuit simulation system on GPU. In *Proceedings of the ACM International Conference on Supercomputing*. ACM, 443–454.

[137] Chen Zhang, Haojie Wang, Zixuan Ma, Lei Xie, Zeyu Song, and Jidong Zhai. 2022. UniQ: a unified programming model for efficient quantum circuit simulation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[138] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[139] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 190–195.

[140] Alwin Zulehner and Robert Wille. 2019. Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2019), 848–859.