FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array

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

Rongliang Fu
The Chinese University of Hong Kong
Hong Kong
rlfu@cse.cuhk.edu.hk

Lukas Burgholzer
Technical University of Munich
Germany
lukas.burgholzer@tum.de

Robert Wille
Technical University of Munich
Germany
robert.wille@tum.de

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

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

ABSTRACT

Quantum circuit simulator (QCS) is essential for designing quantum algorithms because it assists researchers in understanding how quantum operations work without access to expensive quantum computers. Traditional array-based QCSs suffer from exponential time and memory complexities. To address this problem, Decision Diagram (DD) was introduced to compress simulation data by exploring the circuit regularity. However, for irregular circuit structures, DD-based simulation incurs significant runtime and memory overhead. To overcome this challenge, we present `FlatDD`, a high-performance QCS that capitalizes on the strength of both DD- and array-based approaches. `FlatDD` parallelizes the simulation workload at multiple levels and leverages an efficient caching technique to reuse historical results. To further enhance the simulation performance for deep circuits, `FlatDD` introduces a gate-fusion algorithm to reduce the computational cost. Compared to state-of-the-art QCSs on commonly used quantum circuits, `FlatDD` achieves 34.81× speed-up and 1.93× memory reduction.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Simulation tools**; • **Computer systems organization** → **Quantum computing**.

KEYWORDS

Quantum Circuit Simulation, Decision Diagram, Exponentially Weighted Moving Average

ACM Reference Format:
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 *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3673038.3673073

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673073

1 INTRODUCTION

Quantum computing (QC) has the potential to efficiently handle certain types of problems that are classically intractable, such as quantum chemistry [9], finance [35], and cryptography [20]. Powered by two fundamental quantum phenomena, *superposition* and *entanglement*, many available quantum computers have shown significant speed-up over classical computers. For example, a recent photonic quantum computer `Jiuzhang` can solve the Gaussian boson sampling problem within five minutes, whereas a supercomputer needs billions of years [95]. To enable widespread use of this quantum advantage, researchers have been actively building software stacks to support quantum computer designs [4, 11, 41, 81, 82, 90].

Among various QC applications, developing an efficient *quantum circuit simulator* (QCS) on a classical computer is a crucial task because it helps researchers understand how quantum operations work and verify the functionality of a quantum algorithm. However, this task is extremely challenging because it demands large space and time complexity to compute state amplitudes of qubits. For instance, state-of-the-art QCSs [1, 4, 5, 19, 63, 68, 89, 91, 94] use arrays to represent quantum gate matrices and state vectors. An n-qubit circuit may result in a worst case of multiplying a $2^n \times 2^n$ quantum gate matrix by a $2^n \times 1$ state vector. To tackle this challenge, [86, 99] have proposed *decision diagram* (DD) to compress simulation data in a compact graph-based data structure by exploring regularity in the circuit, such as state amplitude distribution and gate matrices structures. As a result, DD can significantly reduce the space complexity and largely improve the simulation time. It is widely used by many quantum software projects [11, 21, 22, 36, 37, 97].

Despite superior performance on regular circuit structures (e.g., quantum arithmetic), DD-based simulators cannot efficiently handle circuits that exhibit irregular distributions of state amplitudes, such as deep neural network (DNN) [10], variational quantum eigensolver (VQE), and Google's quantum supremacy circuits [7]. When simulating these irregular circuits, DD has few advantages but suffers from exponential runtime and memory overhead due to its graph structure–which would otherwise be more efficient using 1D arrays. Figure 1 illustrates this problem by showing the runtime and memory results between a DD-based simulator [99] and an array-based simulator [19] on two regular (Adder, GHZ State [88]) and two irregular (DNN, VQE) quantum circuits.

To solve this problem, we have identified an important property: In DD-based simulation, the quantum gate matrix has a regular

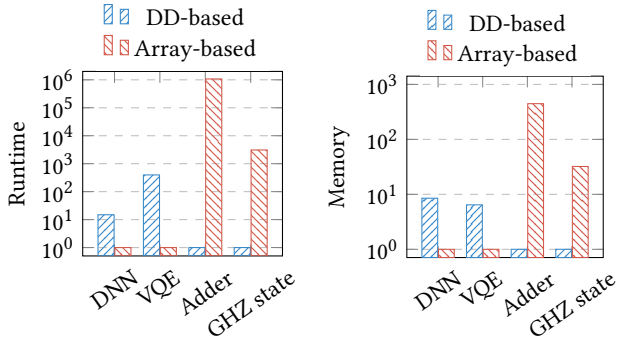


Figure 1: Normalized runtime and memory results between a DD-based simulator [99] and an array-based simulator [19] on four structurally different quantum circuits.

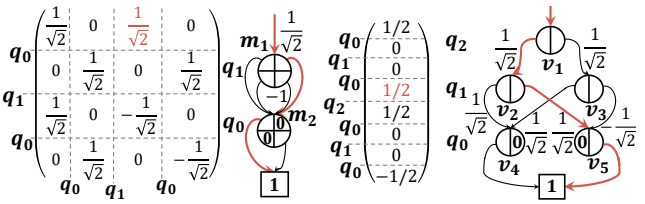
structure as it can be recursively decomposed to unitary operators through Kronecker product. On the other hand, quantum state vectors exhibit a different behavior because of superposition and amplification over the state space [21]. Typically, the vector starts with a highly regular distribution and gradually becomes irregular as the simulation progresses. With this property, we present a high-performance QCS called `FlatDD` that capitalizes on the strength of both DD- and array-based approaches. We summarize our technical contributions below:

- We introduce a hybrid data structure *DMAV*, which uses DD-based gate matrix and array-based state vector for matrix-vector multiplication. DD-based gate matrix enhances indexing efficiency, while array-based state vector avoids exponential overhead from irregularity.
- We introduce a parallel DMAV algorithm with an efficient caching technique to reuse simulation data. Our DMAV algorithm overcomes the limitation of DD-based simulation which is inherently sequential, thus largely enhancing its runtime scalability on a multicore architecture.
- We introduce a moving average-based algorithm to effectively decide when to convert the simulation from DD to DMAV. Since such a conversion can be time-consuming for large circuits, we introduce a parallel algorithm to maximize the conversion efficiency.
- We introduce a DMAV-aware gate-fusion algorithm to enhance `FlatDD`'s efficiency in handling large quantum circuits with deep simulation length.

We evaluated `FlatDD` on a set of widely-used quantum circuits from [7, 69, 88]. `FlatDD` can outperform two highly optimized state-of-the-art DD-based and array-based simulators, `DDSIM` [99] and `Quantum++` [19], with significant runtime improvement. For example, `FlatDD` achieves an average of 34.81× and 17.31× speed-up over `DDSIM` [99] and `Quantum++` [19]. The source code is available at https://github.com/IDEA-CUHK/FlatDD.

2 QUANTUM CIRCUIT SIMULATION

Given a quantum circuit, the goal of quantum circuit simulation is to derive the final state value after applying all quantum gate operations to an initial state. Mainstream simulation methods can be categorized to array-based and DD-based, explained below:



(a) Matrix M in DD. **(b) Vector V in DD.**

Figure 2: Examples of array-based data represented in decision diagrams. Edges without labels have a weight of one by default.

2.1 Array-based Simulation

In array-based simulation [1, 4, 5, 19, 63, 68, 89, 91, 94], gate matrices and state vectors are stored in 2D and 1D arrays. Multiplying a 2D array with a 1D array expresses the action of exerting a quantum gate on a state vector. For instance, when we apply a single-qubit Hadamard gate H to an input state $|\psi\rangle = |0\rangle$, it yields the resulting state $|\psi'\rangle$ (Equation 1).

$$|\psi'\rangle = H \cdot |\psi\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\cdot\begin{pmatrix}1\\0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} \tag{1}$$

This can also extend to larger circuits with $n > 2$ qubits. However, we do not have to construct an entire $2^n \times 2^n$ gate matrix [89, 91]. Instead, array-based simulators can manipulate the amplitudes of state vectors locally. For instance, if we apply a single-qubit gate $U = (u_{ij})$ to the k-th qubit of state vector $|\psi\rangle = (a_i)^T$, the operation to derive resulting state $|\psi'\rangle = (a'_i)^T$ can be expressed in Equation 2.

$$\begin{pmatrix}a'_{*\cdots*0_k*\cdots*}\\ a'_{*\cdots*1_k*\cdots*}\end{pmatrix} = \begin{pmatrix}u_{11} & u_{12}\\ u_{21} & u_{22}\end{pmatrix}\cdot\begin{pmatrix}a_{*\cdots*0_k*\cdots*}\\ a_{*\cdots*1_k*\cdots*}\end{pmatrix} \tag{2}$$

On the other hand, if we apply a two-qubit controlled gate $V = (v_{ij})$ to the state vector, where c represents the control qubit and t is the target qubit, the in-place manipulation of the state vector is expressed in Equation 3.

$$\begin{pmatrix}a'_{*1_c*\cdots*0_t*\cdots*}\\ a'_{*1_c*\cdots*1_t*\cdots*}\end{pmatrix} = \begin{pmatrix}v_{11} & v_{12}\\ v_{21} & v_{22}\end{pmatrix}\cdot\begin{pmatrix}a_{*1_c*\cdots*0_t*\cdots*}\\ a_{*1_c*\cdots*1_t*\cdots*}\end{pmatrix} \tag{3}$$

2.2 Decision-diagram-based Simulation

Compared with array-based methods, decision diagram (DD)-based simulators [86, 98, 99] are particularly good at handling the regularity in gate matrices and state vectors. For example, if we equally partition the matrix in Figure 2a into four sub-matrices, we observe that the upper-left, upper-right, and lower-left sub-matrices are identical. The lower-right sub-matrix is exactly the opposite of the other three sub-matrices. As a result, these four sub-matrices can be efficiently stored as one single 2×2 sub-matrix with different weights. This idea extends to all gate matrices and state vectors, which can be recursively partitioned until they become scalar values. This process forms a graph-based data structure, DD, where nodes represent sub-matrices or sub-vectors at various partitioning levels connected by weighted edges. Figure 2 shows DD examples for gate matrix M and state vector V.

In Figure 2a, the root node m_1 represents the entire matrix M, and the four outgoing edges represent four equally partitioned sub-matrices in M. The partition runs on the most significant qubit,

q_1, at the topmost level of the DD. Recursive partitions progress level by level, from the most to the least significant qubit. Each qubit matches a unique level in DD. The weight for m_1's incoming edge is $\frac{1}{\sqrt{2}}$, and the weights for its outgoing edges are 1, 1, 1 and -1, corresponding to the upper-left, upper-right, lower-left, and lower-right sub-matrices. The weights are uniquely decided by *normalization* [86, 99]. After dividing m_1's incoming and outgoing weights, the four sub-matrices are identical, which can be expressed with one single node m_2. m_2 represents a 2×2 identity matrix, which can be further partitioned on qubit q_0. In this partition, the upper-left and lower-right elements point to terminal constant node *one*, and the upper-right and lower-left elements are zero. The matrix value of an index pair equals the product of edge weights along the corresponding path in DD. For instance, for $M[0][2]$ (i.e., $M[|00\rangle][|10\rangle]$), we have $q_1 = q_0 = 0$ and $q_1 = 1, q_0 = 0$ for row and column indices, respectively. Multiplying the edge weights along the path (thick red edges in Figure 2a) yields $M[0][2] = \frac{1}{\sqrt{2}} \times 1 \times 1 = \frac{1}{\sqrt{2}}$.

In Figure 2b, the root node v_1 represents the entire state vector. We equally partition the vector into two sub-vectors on the most significant qubit q_2. The two sub-vectors are neither zero vectors nor a scalar multiple of each other. Therefore, they are represented in two unique nodes v_2 and v_3. The incoming weights for v_2 and v_3 are $\frac{1}{\sqrt{2}}$, also uniquely determined by normalization. Thus, the weight products along the paths to v_2 and v_3 are both $\frac{1}{\sqrt{2}}$. After dividing the weight products, the two sub-vectors represented by v_2 and v_3 can be further partitioned into $\left(\frac{1}{\sqrt{2}}\ \ 0\right)^T$, $\left(0\ \ \frac{1}{\sqrt{2}}\right)^T$, $\left(\frac{1}{\sqrt{2}}\ \ 0\right)^T$ and $\left(0\ \ -\frac{1}{\sqrt{2}}\right)^T$ on q_1 level, where the first and the third are identical, represented in node v_4, and the second and the fourth are opposite, represented in node v_5 with opposite incoming weights. After further dividing weight products, v_4 and v_5 represent $\left(1\ \ 0\right)^T$ and $\left(0\ \ 1\right)^T$ on the q_0 level, respectively. Similarly, the amplitude of an index is equal to the product of edge weights along the corresponding path in DD. For example, to determine $V[3]$ (i.e., $V[|011\rangle]$), we have $q_2 = 0$, $q_1 = 1$ and $q_0 = 1$. Multiplying the edge weights along the path (thick red edges in Figure 2b) gives $V[3] = 1 \times \frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} \times 1 = \frac{1}{2}$.

DD-based matrix-vector multiplication is done in a depth-first-search (DFS) fashion, where each matrix node always finds its corresponding vector node counterpart on the same level. Identical matrix-vector multiplications are avoided using hash tables [86, 98, 99].

3 ALGORITHM

In this section, we discuss the technical details of our `FlatDD` algorithm. Figure 3 gives an overview of `FlatDD`. As aforementioned, the quantum state vector typically begins with a highly regular distribution and gradually turns irregular as the simulation progresses. For example, in Figure 3, DD-based simulation has a fast runtime until about 24 gates at which the cost to maintain an irregular DD-based state vector outweighs its advantage. Considering this property, `FlatDD` begins with DD-based simulation using `DDSIM` [99] and converts to parallel simulation with DD-based gate matrix and array-based state vector multiplication (DMAV). We

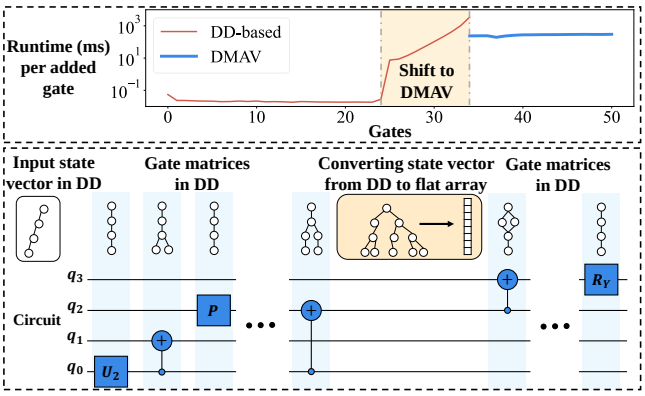


Figure 3: Overview of the `FlatDD` algorithm. The top box displays the runtime for each quantum gate, while the bottom box shows the quantum circuit and simulation data structures used to compute state vectors.

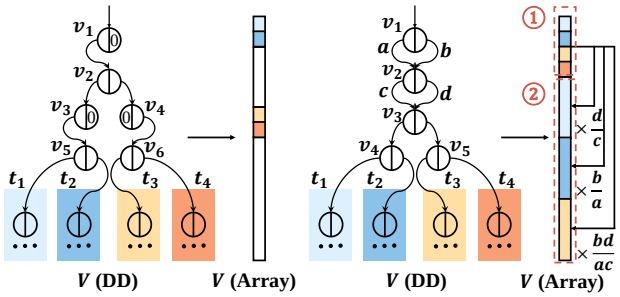
record a window of the DD size using moving average and examine if the state vector's regularity has experienced a significant change. When such a change happens, we use a parallel algorithm to convert the state vector from DD to array (Section 3.1), thereby converting the simulation from DD-based to DMAV. In Section 3.2, we introduce an efficient parallel DMAV algorithm with a caching technique to maximize performance. To further enhance the performance of `FlatDD` on large, deep circuits with thousands of gates, we introduce a DMAV-aware gate-fusion algorithm (Section 3.3). Throughout this paper, we use t to represent the number of threads and n to represent the number of qubits.

3.1 Conversion from DD-based Simulation to DMAV

When the regularity of a DD-based state vector decreases to a certain level, DD-based simulation becomes less advantageous due to the cost of maintaining a highly irregular DD structure. To address this problem, a potential solution is to convert the DD-based state vector to an array representation when the DD size exceeds a certain threshold. The challenge is thus to decide when and how to perform this conversion. To this end, we decide the conversion timing via moving average (Section 3.1.1) and introduce a parallel algorithm to efficiently convert DD to array (Section 3.1.2).

3.1.1 Conversion timing. To decide the best timing for conversion from DD to array, we monitor the DD size and perform conversion when the regularity of DD experiences a drastic increase. Specifically, we apply a widely used learning method, *exponentially weighted moving average* [59] (EWMA). EWMA introduces little computational overhead and is suitable for different quantum circuits and simulation platforms. While simulating, gate i is assigned an EWMA value v_i based on the previous EWMA value v_{i-1} and the DD size s_i, of the state vector at gate i. v_i is computed through Equation 4.

$$v_i = \beta \cdot v_{i-1} + (1 - \beta) \cdot s_i \tag{4}$$



(a) Load balancing: Avoiding idle threads at zeros. (b) Scalar multiplication: Facilitating SIMD parallelism.

Figure 4: Parallel DD-to-array conversion algorithm with two optimizations. The four colors represent four threads.

In Equation 4, β is a parameter and v_0 is initialized to 0. β captures the learning feature of EWMA based on a weighted window of historical data. To decide whether to convert at gate i, we compare $\epsilon \cdot v_i$ with s_i , where ϵ is a threshold parameter. (1) If $\epsilon \cdot v_i \geq s_i$, then s_i is not large enough to benefit from DMAV. (2) On the other hand, if $\epsilon \cdot v_i < s_i$, we convert from DD-based simulation to DMAV (Section 3.2) by converting state vector from DD to 1D flat array.

3.1.2 Parallel Conversion from DD to Array. Although DD-based QCS `DDSIM` [99] already incorporates a conversion algorithm, it is inherently sequential. As a result, it can consume a large portion of the total simulation runtime. To overcome this challenge, we introduce a parallel algorithm with two optimizations, load balancing and scalar multiplication, to improve the conversion efficiency.

Figure 4 illustrates how our parallel conversion algorithm converts state vector V from DD to array using four threads (t_1, t_2, t_3, and t_4). In our algorithm, each DD node serves as a junction that divides threads across its two outgoing edges, progressing from top to bottom until further division of threads is not possible. Then, each thread traverses the unvisited nodes using depth-first-search (DFS), and computes the state amplitudes through the products of weights along the traversal paths. In practice, however, edges may be zero. For example, as shown in Figure 4a, the right outgoing edges of v_1 and v_3, and the left outgoing edge of v_4 are zero. We address this problem with load-balancing optimization: If a DD node's outgoing edge is zero, threads are not divided; instead, they all proceed along the non-zero edge. With this optimization, in Figure 4a, all four threads follow the left outgoing edge at node v_1 to node v_2. Here, the threads divide: two take left to v_3 and two take right to v_4. At nodes v_3 and v_4, the threads continue to nodes v_5 and v_6, respectively, and divide at v_5 and v_6. This optimization prevents any threads from remaining idle.

Additionally, we optimize the conversion by identifying opportunities for scalar multiplication, as it benefits from single instruction multiple data (SIMD) parallelism. For example, in Figure 4b, we can derive the second, third, and fourth quarters of the array from its first quarter with scalar multiplication. This is because v_1 and v_2 each have identical child nodes, indicating that the four quarters of the array are scalar multiples of one another [86]. Consequently, we break the conversion in Figure 4b into two steps: ① Convert the first quarter. With multi-threading, all four threads make two consecutive left turns at v_1 and v_2, for they each have identical child nodes, and divide at nodes v_3, v_4, and v_5, to convert the first quarter in parallel. ② SIMD-enabled scalar multiplication fills the second, third, and fourth quarters with three threads, using the first quarter data. For instance, the fourth quarter is calculated by multiplying the first quarter by scalar bd/ac. This optimization enhances performance by leveraging both SIMD and multi-threading parallelism.

3.2 Simulation with DMAV

DMAV is different from existing matrix-vector multiplications [8, 85] because a DD-based gate matrix has a specific regularity property due to the tensor product. This regularity allows reusing computed results through caching. However, caching requires additional memory to store historical data, which can introduce overhead that, for certain gates, can outweigh its benefits. To effectively determine which gates benefit from caching, we introduce a computational cost model for DMAV (Section 3.2.3). Using this model, we apply DMAV without caching (Section 3.2.1) when its computational cost is lower. Otherwise, we apply DMAV with caching (Section 3.2.2).

3.2.1 DMAV without Caching. We describe DMAV without caching, using Algorithm 1 and an example in Figure 5. In Figure 5, we multiply a three-qubit (i.e., $n = 3$) gate matrix M by a state vector V using two threads (in blue and red, $t = 2$), to derive state vector W. Algorithm 1 consists of three functions: `DMAV`, `Assign`, and `Run`. The top-level function `DMAV` takes input variables M (the topmost edge to DD node m_1 in Figure 5), as well as V and W (the input and output state vectors in flat arrays). `DMAV` has two steps: assigning multiplication tasks to be executed on t threads via `Assign` (line 2), and running them in parallel via `Run` (lines 3-5).

In DMAV without caching, each thread evaluates M in row space by computing h rows from M and the entire vector V, resulting in an h-sized sub-vector in W, where $h = 2^n/t$. `Assign` decomposes this process for each thread into smaller multiplication tasks. `Assign` divides matrix M into $h \times h$ sub-matrices, and vectors V and W into h-sized sub-vectors. Each sub-matrix is paired with the corresponding sub-vectors, forming a multiplication task. A DD sub-matrix can be located using its incoming edge, while a sub-vector can be located using its start index in V or W. Thus, to record the multiplication tasks for t threads, we use 2D vectors v_M, v_V, v_f (each of length t) to track the sub-matrices' DD edges, sub-vectors' start indices, and the weight products along the DD traversal paths. `Assign` is a recursive function, where the input arguments are dynamically decided during the recursive call. At line 8, M_r is the input sub-matrix's DD edge, and f_r is the weight product along the DD traversal path. If each DD edge M_{r_i} has weight $M_{r_i}.w$ along DD traversal path P, f_r is given by $f_r = \Pi_{i \in P} M_{r_i}.w$. Argument u is the thread index for task assignment. I_V is the start index for a sub-vector in V. l denotes the DD level of the current node, pointed to by edge M_r (i.e., $M_r.n$). `Assign` is called from `DMAV` (line 2) using DD edge M. The call initializes weight product f_r, thread index u, and start index I_V to 1, 0, and 0 respectively, on the topmost level $l = n - 1$.

In `Assign`, if M_r is zero, it returns (line 9). $n - \log_2 t - 1$ represents the *border level*: `Assign` ends here, and `Run` starts from here. In Figure 5, the border level is q_1 (i.e., $n - \log_2 t - 1 = 1$). `Assign` spans levels q_2 and q_1 (i.e., $l = 2, 1$), while `Run` spans levels q_1, q_0, and

the final level where the bottom-most terminal node one resides (i.e., $l = 1, 0, -1$). Upon reaching the border level, we push the sub-matrix's DD edge M_r, the sub-vector's start index I_V, and the weight product f_r of thread u to the corresponding vectors (lines 10-11). If the border level is still not reached, we traverse the four outgoing edges of $M_r.n$ in row-major order (i.e., $M_r.n.e[i][j], \forall i, j \in \{0,1\}^2$) and call Assign with the updated weight product $M_r.w \cdot f_r$ (where $M_r.w$ is the weight of M_r), thread index $u + i \cdot t/2^{n-l}$ and sub-vector's start index, $I_V + 2^l j$, going one level lower to $l-1$ (lines 12-13). The purposes of $u + i \cdot t/2^{n-l}$ and $I_V + 2^l j$ are to split t threads and V into halves, quarters, and so on, at each successive level, until all threads are allocated a multiplication task. In Figure 5, on level q_2, we call Assign with $u = 0, 1$ and $I_V = 0, 4$, essentially halving t threads and V. After Assign, the blue thread in Figure 5 is assigned $a \cdot m_2 \cdot V[0:4]$ and $b \cdot m_2 \cdot V[4:8]$, while the red thread is assigned $c \cdot m_2 \cdot V[0:4]$ and $d \cdot m_2 \cdot V[4:8]$.

Algorithm 1 DMAV without caching

Input: $h = \frac{2^n}{t}$, the size for a sub-matrix or a sub-vector; $v_M = v_V = v_f = \underbrace{\{\{\}, \cdots, \{\}\}}_{t}$, vectors keeping track of the sub-matrices' DD edges, sub-vectors' start indices, and weight products assigned to each thread, respectively.

```
1:  function DMAV(M, V, W)
2:    Assign(M, 1, 0, 0, n − 1)
3:    parallel for i ∈ [0, t)
4:      for j ∈ [0, size(v_M[i]))
5:        Run(v_M[i][j], V, W, n − log2 t − 1, v_V[i][j], ih, v_f[i][j])
6:  end function
7:
8:  function Assign(M_r, f_r, u, I_V, l)
9:    if M_r is zero edge then return
10:   if l == n − log2 t − 1 then
11:     v_M[u] ∪ {M_r}; v_V[u] ∪ {I_V}; v_f[u] ∪ {f_r}; return
12:   else for i, j ∈ {0, 1}^2
13:     Assign(M_r.n.e[i][j], M_r.w · f_r, u + (i·t)/2^{n−l}, I_V + 2^l j, l − 1)
14: end function
15:
16: function Run(M_r, V, W, l, I_V, I_W, f_r)
17:   if M_r is zero edge then return
18:   if M_r.n is terminal node then
19:     W[I_W] ← W[I_W] + f_r · M_r.w · V[I_V]; return
20:   for i, j ∈ {0, 1}^2
21:     Run(M_r.n.e[i][j], V, W, l − 1, I_V + 2^l j, I_W + 2^l i, f_r M_r.w)
22: end function
```

Run is also a recursive function. We launch t threads (line 3), and iterate through the edges in v_M determined by Assign (line 4). At line 5, we calculate each multiplication task $v_f[i][j] \cdot v_M[i][j] \cdot V[v_V[i][j] : v_V[i][j] + h]$ via Run. The input parameters (line 16) for Run consist of the following: M_r, sub-matrix's DD edge; V and W, the input and output state vectors; l, the level on which node $M_r.n$ is located; I_V and I_W, start indices for accessing V and W; and f_r, weight product. In Run, if M_r is zero, then we return directly (line 17). If $M_r.n$ is a terminal node (lines 18-19), we multiply $M[I_W][I_V]$ by $V[I_V]$, and add the result to $W[I_W]$. If $M_r.n$ is not yet terminal, we traverse the four outgoing edges of $M_r.n$ and call Run accordingly. We update the weight product with $f_r \cdot M_r.w$, I_V with $I_V + 2^l j$, and I_W with $I_W + 2^l i$ (lines 20-21). The purposes of $I_V + 2^l j$, $I_W + 2^l i$ are to divide the h-sized sub-vectors in V and W into halves, quarters, and so on, at each successive level, until the sub-vectors cannot be divided further, at which point M_r also reaches a terminal node.

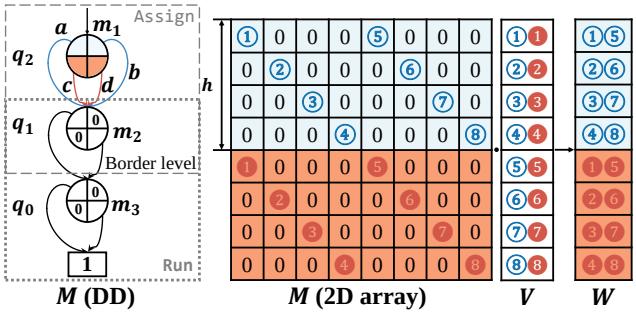


Figure 5: DMAV without caching on two threads, illustrated in blue and red colors. The input matrix and vector are M and V, and the resulting vector is W (i.e., $M \cdot V \rightarrow W$).

In Figure 5, the blue thread first computes $a \cdot m_2 \cdot V[0:4]$ (allocated by Assign). We make two recursive Run calls, using the upper-left outgoing edges of m_2 and m_3 for the M_r inputs, and setting both I_V and I_W to 0, leading to $W[0] \leftarrow W[0] + M[0][0] \cdot V[0]$ at step ①. Then, at step ②, we exit the outermost Run and call another Run ($M_r = m_3$'s lower-right edge, and $I_V = I_W = 1$), leading to $W[1] \leftarrow W[1] + M[1][1] \cdot V[1]$. This continues until $a \cdot m_2 \cdot V[0:4]$ is finished and we make a switch for task $b \cdot m_2 \cdot V[4:8]$ at step ⑤. A similar process applies to the red thread.

DMAV outperforms array-based QCSs (e.g., Quantum++ [19]) due to its efficient indexing. Unlike Quantum++, which requires $O(n)$ operations per state, DMAV uses a recursive Run function within a DD structure, enhancing data locality and reducing indexing to a constant average number of operations. This results in an $n\times$ increase in indexing speed for DMAV compared to Quantum++.

3.2.2 DMAV with Caching. Figure 6 illustrates an example of computation reduction with caching in DMAV. Executing the DMAV in Figure 6 involves two multiplications: ① $am_2 \cdot V[0:2]$ and ② $bm_2 \cdot V[0:2]$. Without caching, both require four multiply-accumulate (MAC) [2] operations. However, ① and ② share identical left and right operands (sub-matrix DD node and sub-vector) and differ only in their scalar coefficients (a or b). Caching the result of ① allows its reuse for ② by scaling it by b/a, yielding only two MAC operations, compared to four without caching.

DMAV with caching is notably different from DMAV without caching. Specifically, to facilitate data reuse, each thread evaluates the gate matrix in column space instead of row space. We describe DMAV with caching using both Algorithm 2 and Figure 7. In Figure 7, we multiply gate matrix M by state vector V using four threads, each with its own cache, to obtain the resulting state vector W. In Algorithm 2, we present two functions: top-level DMAVCache,

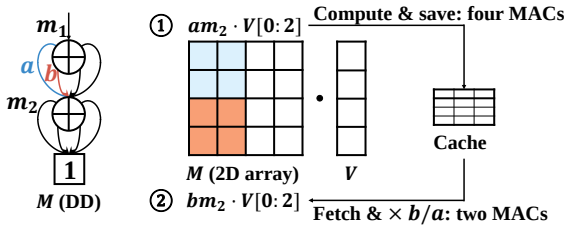


Figure 6: Saving the number of MAC operations by reusing historical results in DMAV.

which conducts DMAV with caching, and AssignCache, which assigns multiplication tasks to t threads. For previously computed multiplication tasks, we can fetch the results from the cache, while other multiplication tasks are computed using Run from Algorithm 1. We only apply caching to the multiplication tasks directly assigned by AssignCache to avoid frequent memory allocation and deallocation on different threads.

In DMAV with caching, each thread computes h columns in M and an h-sized sub-vector in V ($h = 2^n/t$), generating a partial output equivalent in size to V. These partial outputs are then summed to obtain the output state vector W. Given the sparse nature of quantum gate matrices [34], the partial outputs from different threads often have non-overlapping segments. Thus, to save memory and time, multiple partial outputs can share a single memory *buffer*, and we sum the buffers to obtain the output state vector.

AssignCache, which is adapted from Assign in Algorithm 1, divides matrix M into $h \times h$ sub-matrices, and vector V as well as the partial output vectors into h-sized sub-vectors. Each sub-matrix is paired with the corresponding sub-vectors, forming a multiplication task. To record the multiplication tasks for t threads, we use vectors v_M, v_P, and v_f to keep track of the sub-matrices' DD edges, start indices of sub-vectors in a partial output, and weight products assigned to each thread. Vector v_B associates each thread with a specific buffer index, which can be used to access buffers in B, the vector of buffers. The input parameters for AssignCache are: M_r, input matrix's DD edge; f_r, weight product along DD traversal path; u, thread index; I_P, start index of a sub-vector in a partial output; and l, DD level of the current node. AssignCache returns if M_r is zero (line 17). Upon reaching the border level at $n - \log_2 t - 1$, we push the sub-matrix's edge M_r, the sub-vector's start index I_P, and the weight product f_r of thread u to the corresponding vectors (lines 18-19). At lines 20-21, we traverse the four outgoing edges of $M_r.n$ in column-major order and call AssignCache with the updated weight product, thread index, sub-vector's start index, and DD level, similar to Assign (line 13 in Algorithm 1). After all threads have been assigned multiplication tasks, we assign each thread to its buffer. Specifically, for each thread i, we check if there is another thread j with a non-overlapping partial output. If such a thread j exists, threads i and j will share a buffer (line 24); otherwise, thread i will receive its own buffer (line 25).

In DMAVCache, each thread executes its multiplication tasks from AssignCache (line 2). We focus on threads t_1 and t_2 in Figure 7 (in black dashed boxes). t_1 is assigned $a \cdot m_4 \cdot V[0:h]$ and $b \cdot m_4 \cdot V[0:h]$, while t_2 is assigned $c \cdot m_5 \cdot V[h:2h]$ and $d \cdot m_5 \cdot V[h:2h]$, where sub-vectors $V[0:h]$ and $V[h:2h]$ are the first two quarters in V. As these four tasks have non-overlapping outputs, they can share the same output buffer $B[0]$. At step ①, t_1 and t_2 execute $am_4 \cdot V[0:h]$ and $cm_5 \cdot V[h:2h]$ simultaneously. First, they search in caches (line 5). Finding no historical results, they directly call the Run function from Algorithm 1. Upon obtaining the result, t_1 caches m_4 with a pair consisting of m_4's weight product and the start index of the resulting sub-vector in t_1's partial output (i.e., $\{a, 0\}$). We only cache the left operand (sub-matrix DD node m_4) because the right operand (h-sized sub-vector in V) is the same for all assigned multiplication tasks on the same thread. Similarly, t_2 caches m_5 and $\{c, h\}$ (lines 8-10). At step ②, t_1 and t_2 execute $bm_4 \cdot V[0:h]$ and $dm_5 \cdot V[h:2h]$ simultaneously. Again, they check if m_4 and m_5 are cached, discovering historical results (lines 5-6). t_1 finds an h-sized sub-vector in its partial output (stored in buffer $B[0]$) at start index 0 with weight a. It then multiplies this sub-vector by

Algorithm 2 DMAV with caching

Input: $h = \frac{2^n}{t}$, the size for a sub-matrix or a sub-vector; $v_M = v_P = v_f = \underbrace{\{\{\}, \cdots, \{\}\}}_{t}$, vectors keeping track of the sub-matrices' DD edges, start indices of sub-vectors in a partial output, and weight products, respectively; v_B, a vector of buffer indices assigned to each thread; B, a vector of buffers for partial outputs; *cache*, a vector where each element is a cache specific to a thread.

1: **function** DMAVCache(M, V, W)
2: AssignCache(M, 1, 0, 0, $n-1$)
3: **parallel for** $i \in [0, t)$
4: **for** $j \in [0, size(v_M[i]))$
5: $r \leftarrow cache[i]$.lookup($v_M[i][j].n$)
6: **if** $r \neq \{\}$ **then**
7: $B[v_B[i]][v_P[i][j] : v_P[i][j]+h] \leftarrow$ SIMDMul($v_f[i][j]/r[0]$, $B[v_B[i]][r[1] : r[1]+h]$)
8: **else**
9: Run($v_M[i][j]$, V, $B[v_B[i]]$, $n - \log_2 t - 1$, $h \cdot i$, $v_P[i][j]$, $v_f[i][j]$) /* From Algorithm 1 */
10: $cache[i]$.insert($v_M[i][j].n$, $\{v_f[i][j], v_P[i][j]\}$)
11: **parallel for** $i \in [0, t)$
12: **for** $j \in [0, size(B))$
13: $W[i \cdot h : (i+1) \cdot h] \leftarrow$ SIMDAdd($W[i \cdot h : (i+1) \cdot h]$, $B[j][i \cdot h : (i+1) \cdot h]$)
14: **end function**
15:
16: **function** AssignCache(M_r, f_r, u, I_P, l)
17: **if** M_r **is** zero edge **then return**
18: **if** $l == n - \log_2 t - 1$ **then** /* Border level */
19: $v_M[u] \cup \{M_r\}$; $v_P[u] \cup \{I_P\}$; $v_f[u] \cup \{f_r\}$; **return**
20: **else for** $j, i \in \{0, 1\}^2$
21: AssignCache($M_r.n.e[i][j]$, $f_r M_r.w$, $u + \frac{j \cdot t}{2^{n-l}}$, $I_P + 2^l i$, $l-1$)
22: **if** $l == n-1$ **then for** $i \in [0, size(v_P))$
23: **if not** Overlap($v_P[i]$, $v_P[j]$), $\exists j \in [0, i)$ **then**
24: $v_B[i] \leftarrow v_B[j]$
25: **else** $v_B[i] \leftarrow size(B)$; $B \cup \{\underbrace{\{0, \cdots, 0\}}_{2^n}\}$
26: **end function**

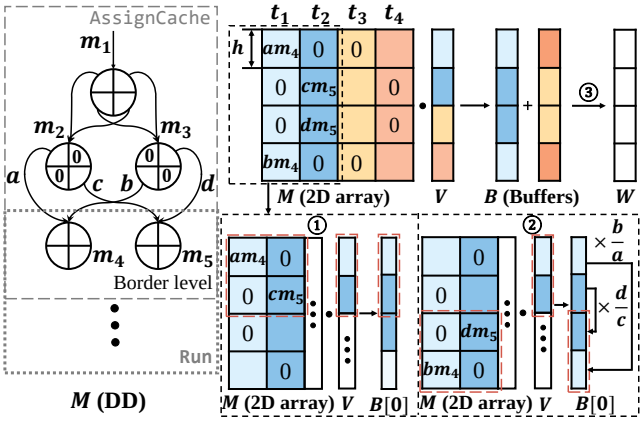


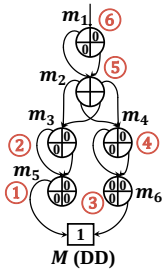
Figure 7: DMAV with caching on four threads, illustrated in the four colors. The input matrix and vector are M and V, and the resulting vector is W (i.e., $M \cdot V \rightarrow W$). DD edges without labels have a weight of one by default.

b/a using SIMD-enabled scalar multiplication and places it in the fourth quarter of the $B[0]$ (line 7). Likewise, t_2 performs a similar scalar multiplication. Finally, at step ③, we sum the buffers to form the final result W using multi-threading and SIMD (lines 11-14).

3.2.3 Computational Cost in DMAV. Based on our observation, the MAC operation dominates DMAV computation. For example, within the Run function (Algorithm 1), which accounts for 99.99% of DMAV's runtime, the only non-trivial computation is performed by line 19, which executes a MAC operation. Therefore, we introduce the computational cost model based on the number of MAC operations.

First, it is necessary to determine the number of MAC operations in a DMAV. As shown in Figure 8, we use DFS to traverse DD nodes and a look-up table (MAC count table T) to track MAC operations for unique nodes, given that identical DD nodes incur the same number of MAC operations. The number of MAC operations of each node is the sum of that of its children, and the terminal node has one MAC operation. For each node, we traverse the outgoing edges from top to bottom and from left to right. Following this rule, at step ①, we go through m_1, m_2, m_3, and arrive at m_5. m_5 only has one outgoing edge to a terminal node with one MAC (i.e., $T(m_5) = 1$). At step ②, we proceed to m_3's lower-right outgoing edge, which also points to m_5. Thus, m_3 has two MAC operations (i.e., $T(m_3) = 2T(m_5) = 2$). Then, we return to m_2, and visit the upper-right edge pointing to m_4. Similarly, we derive that $T(m_6) = 1$ and $T(m_4) = 2$, at steps ③ and ④. The lower outgoing edges of m_2 connect to the previously visited nodes m_3 and m_4, giving us $T(m_2) = 2T(m_3) + 2T(m_4) = 8$ at step ⑤. Finally, at step ⑥, we return to m_1, obtaining $T(m_1) = 2T(m_2) = 16$. The total MAC count for this DMAV is $T(m_1) = 16$.

Then, we model the computational cost based on the number of MAC operations. We observe that DMAV's workload distribution among threads, with or without caching, is always balanced. Therefore, the computational cost model evenly divides the number of MAC operations by t threads. Suppose K_1 is the number of MAC



MAC count table T

	Node	#MACs
⑥	m_1	$2T(m_2) = 16$
⑤	m_2	$2T(m_3) + 2T(m_4) = 8$
②	m_3	$2T(m_5) = 2$
④	m_4	$2T(m_6) = 2$
①	m_5	1
③	m_6	1

Figure 8: Counting the number of MAC operations of a DMAV.

operations for a DMAV without caching, Equation 5 models its computational cost C_1.

$$C_1 = \frac{K_1}{t} \tag{5}$$

For DMAV with caching, we consider three costs. (1) **The cost of scalar multiplication.** By identifying H cache hits from repeated nodes in v_M (Algorithm 2), each corresponding to a scalar multiplication of size $2^n/t$, we determine that the number of MAC operations for scalar multiplication is $2^n H/t$. If SIMD computes d data elements simultaneously, on t threads, the cost of scalar multiplication is $2^n H/(d \cdot t^2)$. (2) **The cost of summing buffers.** Summing b buffers, each of size 2^n, incurs $2^n b$ MAC operations. When using SIMD on t threads, the cost of summing buffers is $2^n b/(d \cdot t)$. (3) **The cost unrelated to caching.** We count the MACs unrelated to caching in DMAV, as in Figure 8, this time avoiding the addition of the repeated nodes mentioned in (1). Suppose there are K_2 MAC operations unrelated to caching, the corresponding cost is K_2/t. In summary, we model the computational cost C_2, for DMAV with caching as the following.

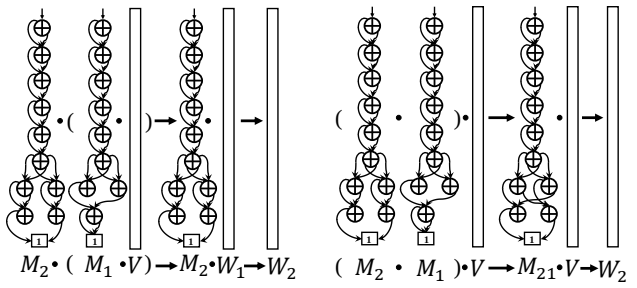
$$C_2 = \frac{K_2}{t} + \frac{2^n}{d \cdot t}\left(\frac{H}{t} + b\right) \tag{6}$$

To minimize the total computational cost, we choose DMAV with caching if $C_1 > C_2$, and without caching if not. To sum up, the computational cost for a DMAV operation is $\min\{C_1, C_2\}$.

3.3 DMAV-Aware Gate-Fusion Algorithm

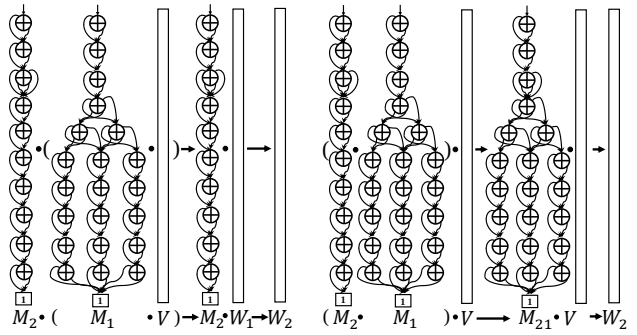
To further boost FlatDD's performance on large circuits, we propose a DMAV-aware gate-fusion algorithm that greedily fuses gates to reduce the total computational cost of DMAV.

Figure 9 shows the advantage of gate fusion. Consider two quantum gates M_1, M_2 in DD, there are two approaches to applying them consecutively to a state vector V. (1) **Sequential DMAV**: Multiply M_1 by V to obtain intermediate state W_1, and then multiply M_2 by W_1 to obtain the final state W_2, as shown in Figure 9a. (2) **Gate fusion**: Apply DD-based matrix-matrix multiplication (DDMM) to M_1 and M_2, fusing them into M_{21}, and then multiply M_{21} by V to obtain the final state W_2, as shown in Figure 9b. For simplicity, we evaluate two options using Equation 5 to determine which one has a lower computational cost. In sequential DMAV, the two consecutive DMAVs result in a total computational cost of $512/t$. In contrast, gate fusion performs one DDMM and one DMAV. The DMAV computational cost is $256/t$. The DDMM calls 12 multiplications and 4 additions, which is negligible compared to the DMAV



(a) Sequential DMAV. (b) Gate fusion.

Figure 9: Computation reduction from gate fusion.



(a) Sequential DMAV. (b) Gate fusion.

Figure 10: Gate fusion can result in more computation.

computational cost since each multiplication or addition only constructs one DD node. Therefore, gate fusion reduces computation compared to sequential DMAV.

However, gate fusion does not always reduce computation, as shown in Figure 10. In Figure 10a, sequential DMAV without gate fusion incurs a computational cost of $6144/t$. Conversely, in Figure 10b, gate fusion results in a DMAV computational cost of $8192/t$ and 21 DDMM multiplication calls. In this case, sequential DMAV has a lower computational cost.

To reduce the total computational cost, we fuse two gates only when the fused gate has a smaller computational cost compared to sequential DMAV. Accordingly, we introduce our DMAV-aware gate fusion in Algorithm 3. Algorithm 3 operates on the group of remaining gates, G, after `FlatDD` conversion. We initialize M_p (the previous gate matrix) to an identity DD matrix and set its DMAV computational cost (Section 3.2.3) C_p to 0 (line 2). Next, we iterate through G (line 3). For the ith gate matrix M_i, we calculate its DMAV computational cost, C_i. Then, we multiply M_i by M_p to form M_{ip}, and calculate its DMAV computational cost C_{ip} (line 4). If $C_i + C_p < C_{ip}$, sequential DMAV is computationally cheaper than gate fusion. We then add M_p to the resulting gate group S. Before the next iteration, we assign the ith gate M_i and its DMAV computational cost C_i to the previous gate M_p and its DMAV computational cost C_p (line 6). If $C_i + C_p < C_{ip}$ fails to satisfy, we fuse the current gate M_i with M_p. Then, we assign the fused gate M_{ip} and cost C_{ip} to the previous gate M_p and cost C_p (line 8). Finally, we return S at line 9.

Algorithm 3 DMAV-aware gate fusion

Input: G, group of remaining gates after `FlatDD` conversion.
Output: S, group of gates after gate fusion.

```
1:  function GateFuse(G)
2:    M_p ← I_DD; C_p ← 0
3:    for M_i ∈ G
4:      C_i ← cost(M_i); M_ip ←DDMM(M_i, M_p); C_ip ← cost(M_ip)
5:      if C_i + C_p < C_ip then
6:        S ∪ {M_p}; C_p ← C_i; M_p ← M_i
7:      else
8:        M_p ← M_ip; C_p ← C_ip
9:    return S
10: end function
```

4 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of `FlatDD` using 12 commonly used quantum circuits from `QASMBench` [69], `MQT Bench` [88] and `Quantum supremacy` [7]. These benchmarks span both regular and irregular structures. First of all, we compare the runtime and memory performance of `FlatDD` with two state-of-the-art QCSs (Section 4.2). Then, we demonstrate the scalability of `FlatDD` over increasing numbers of threads (Section 4.3). We then evaluate the effectiveness of our parallel DD-to-array conversion algorithm (Section 4.4) and DMAV caching technique (Section 4.5). Lastly, in Section 4.6, we evaluate our DMAV-aware gate-fusion algorithm on deep circuits with thousands of gates. All the experiments are conducted on a Ubuntu 22.04.2 LTS machine with 64 Intel Xeon Gold 6226R CPUs at 2.9 GHz and 256 GB memory capacity. We compile all programs with optimization flag `-O3` enabled. All SIMD operations are executed using Intel's `AVX2` [60] vector instructions. For data with exponential difference, we measure the average in geometric mean.

4.1 Baselines

Given the large number of QCSs, it is not possible to compare `FlatDD` with all of them. Instead, we consider two representative QCSs, `DDSIM` [99] and `Quantum++` [19], for two reasons: (1) Both `DDSIM` and `Quantum++` are highly optimized and have demonstrated superior performance over existing QCSs. `DDSIM` introduces a DD-based compact representation of gate matrices and state vectors, as well as an efficient data structure for handling complex numbers [98]. On the other hand, `Quantum++` leverages `OpenMP` [3] to enable multi-threaded simulation based on `Eigen` [23] array and matrix data structures. (2) Both `DDSIM` and `Quantum++` are open-source [6, 87] and widely used by the community, allowing us to fairly study and reason their results.

4.2 Overall Performance Comparison

Table 1 compares the overall performance of `FlatDD` with `DDSIM` and `Quantum++`. In this experiment, we do not incorporate the proposed gate-fusion algorithm but focus on the full-state simulation workload itself. We run `FlatDD` and `Quantum++` using 16 threads and run `DDSIM` using one thread as `DDSIM` does not support multi-threading. For all `FlatDD` runs, we set $\beta = 0.9$ and $\epsilon = 2$, as these values are determined to be effective across multiple quantum circuits. Unless otherwise specified, all runtime and memory are measured in seconds (s) and megabytes (MB). We terminate the runs that take longer than 24 hours. We measure memory usage by recording the maximum resident set size (RSS) of our program using `/bin/time`.

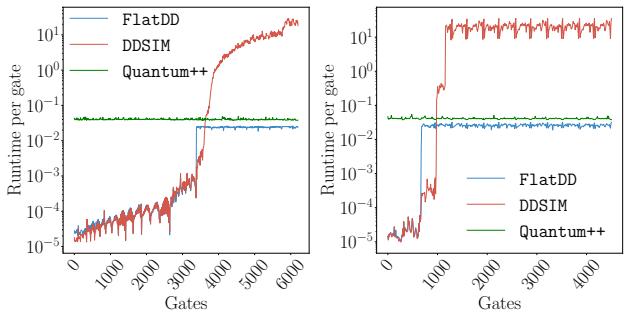
On average, FlatDD is 34.81× and 17.31× faster over DDSIM and Quantum++ on all circuits. In terms of memory usage, FlatDD is 1.70× and 1.93× less than DDSIM and Quantum++. DDSIM demonstrates exceptional performance on regular circuits such as Adder and GHZ state. For instance, the circuit Adder has a very regular distribution of state amplitudes throughout the simulation; DDSIM takes less than 1 s while Quantum++ takes 1793.67 s. However, when the regularity does not appear frequently, such as DNN, VQE, and Quantum supremacy, DDSIM becomes significantly slower than FlatDD and Quantum++. For example, DDSIM is 13.66× and 12.64× slower than FlatDD and Quantum++ when simulating the 16-qubit DNN.

Although DDSIM and Quantum++ have their own strength in simulating certain circuits, FlatDD demonstrates a consistent performance advantage on all. For highly regular circuits, such as Adder, and GHZ state, where DDSIM performs extremely well, FlatDD also achieves fast runtime (< 1 s). This is because FlatDD does not switch from DDSIM to DMAV during the simulation. The only reason that FlatDD can be slower than DDSIM, such as simulating the 23-qubit GHZ state, is the overhead of DD size calculation and conversion timing checking (Section 3.1.1). For irregular circuits, such as 20-qubit DNN and quantum supremacy circuit, FlatDD can still take advantage of DD-based state vector to a certain extent before the state vector turns irregular. After this turning point, as observed from Figure 11a and Figure 11b, the runtime of DDSIM will significantly increase whereas FlatDD cleverly converts to DMAV and stays stable. Additionally, after the turning point, FlatDD is faster than Quantum++ primarily due to the efficient indexing pattern of our DMAV (see Section 3.2).

The memory usage of FlatDD is comparable to DDSIM and Quantum++. When the circuit is small, Quantum++ consumes the smallest memory compared with FlatDD and DDSIM, both of which require additional data storage for DDs. For example, when simulating the 16-qubit DNN, Quantum++ needs only 8.2 MB, whereas FlatDD and DDSIM need 32.26 MB and 69.6 MB, respectively. On the other hand, when the circuit is large, Quantum++ consumes significantly higher memory than FlatDD and DDSIM. For instance, when simulating the 25-qubit KNN, Quantum++ needs 1577.2 MB, while FlatDD and DDSIM only need 1078.78 MB and 388.1 MB, respectively. If the state vector is highly regular (e.g., Adder, GHZ state), FlatDD and DDSIM have similar memory usage, as FlatDD does not switch from DDSIM to DMAV. However, if the state vector contains high irregularity (e.g., DNN, VQE, Quantum supremacy), FlatDD has smaller memory usage than DDSIM because it will switch from DDSIM to DMAV to reduce DD overheads. Lastly, if the state vector is irregular in large circuits, FlatDD consumes smaller memory than both DDSIM and Quantum++. For example, when simulating the 26-qubit quantum supremacy circuit, FlatDD consumes 2132.48 MB memory, while DDSIM and Quantum++ consume 16799.4 and 3156.58 MB memory.

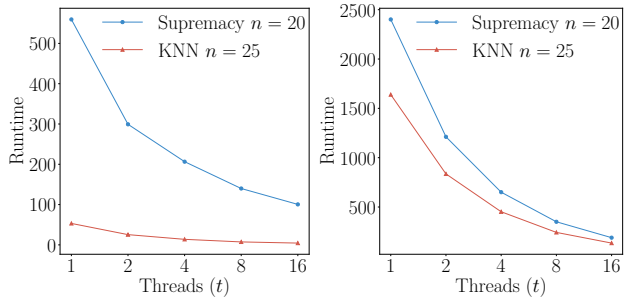
4.3 Scalability of FlatDD

We demonstrate the scalability of FlatDD over increasing numbers of threads. Figure 12 illustrates the runtime of FlatDD and Quantum++ at different numbers of threads on two circuits (Quantum supremacy and KNN). When the number of threads increases,



(a) DNN $n = 20$ (b) Supremacy $n = 20$

Figure 11: Runtime comparison among FlatDD, DDSIM, and Quantum++ per gate.



(a) FlatDD (b) Quantum++

Figure 12: Runtime scalability of FlatDD and Quantum++ under different numbers of threads.

FlatDD can complete the simulation faster, as illustrated in Figure 12a. For instance, at eight threads, FlatDD is 7.26× faster than one thread on KNN. FlatDD's performance saturates at about 16 threads. Likewise, in Figure 12b, Quantum++ shows a similar trend.

4.4 Evaluation of Parallel DD-to-Array Conversion Algorithm

Figure 13 compares the efficiency of FlatDD's parallel DD-to-array conversion algorithm with DDSIM on 10 quantum circuits using 16 threads. In this experiment, we integrate DDSIM's DD-to-array algorithm into FlatDD. Figure 13a compares the conversion time, and Figure 13b compares the cost of conversion in terms of its percentage in the total simulation time.

As shown in Figure 13a, FlatDD outperforms DDSIM in converting DDs to arrays on all circuits. For instance, our DD-to-array algorithm is 61.86× faster than DDSIM on KNN of 25 qubits; on average, FlatDD is 22.34× faster, which is attributed to multi-threading and SIMD parallelism. In terms of conversion cost, as shown in Figure 13b, FlatDD's DD-to-array algorithm takes only about 0.01–7% of the total runtime, while DDSIM can take up to 83.2% (Swap test of 25 qubits). This result highlights the efficiency of our parallel DD-to-array conversion algorithm.

4.5 Evaluation of DMAV Caching Technique

Figure 14 demonstrates the efficiency of the proposed DMAV caching technique in terms of reduced computational cost and speed-up gain at different numbers of threads. We plot the results

Table 1: Comparison of simulation runtime and memory among `FlatDD`, `DDSIM`, and `Quantum++` on 12 widely used circuits that exhibit different regularity structures.

Circuits			`FlatDD` (ours)		`DDSIM` [99]			`Quantum++` [19]		
Name	Qubits (n)	Gates	Runtime	Memory	Runtime	Speed-up	Memory	Runtime	Speed-up	Memory
DNN	16	2032	**4.12**	32.26	56.27	13.66×	69.6	4.45	1.08×	**8.2**
	20	6214	**73.06**	75.26	21847.81	299.02×	504.832	249.62	3.42×	**61.34**
	25	9644	**2283.02**	**1089.54**	> 24 h	> 37.84×	≥ 4249.6	17527.8	7.68×	1586.54
Adder	28	117	0.037	30.21	**0.00167**	0.045×	**28.2**	1793.67	48477.57×	12584.0
GHZ state	23	46	0.028	30.21	**0.004**	0.14×	**29.7**	12.388	442.43×	397.93
VQE	16	95	**0.178**	31.232	79.24	445.92×	50.2	0.20	1.12×	**7.8**
KNN	25	39	**4.75**	1078.78	480.72	101.20×	**388.1**	135.669	28.56×	1577.2
	31	48	**500.22**	67136.51	> 24 h	> 172.72×	≥ 2518.0	13123.5	26.23×	100657.37
Swap test	25	39	**4.84**	1077.76	1649.81	340.94×	**649.7**	136.995	28.31×	1577.6
Quantum supremacy	20	4500	**101.38**	65.54	67814.99	668.92×	700.928	185.207	1.83×	**61.2**
	24	5560	**1050.07**	**558.59**	> 24 h	> 82.28×	≥ 5794.77	5034.7	4.80×	799.23
	26	5990	**3980.07**	**2132.48**	> 24 h	> 21.71×	≥ 16799.4	22632.7	5.69×	3156.58
Geometric mean			**17.08**	**296.38**	> 594.53	> 34.81×	≥ 504.52	295.62	17.31×	571.16

Table 2: Comparison of simulation runtime and computational cost among `FlatDD` with DMAV-aware gate fusion (ours), `FlatDD` without gate fusion, and `FlatDD` with `k-operations` [100] on six deep circuits (> 1000 gates). Red. is the reduction in computational cost.

Circuits			`FlatDD` with DMAV-aware gate fusion (ours)		`FlatDD` without gate fusion				`FlatDD` with `k-operations` [100]			
Name	n	Gates	Runtime	Cost	Runtime	Speed-up	Cost	Red.	Runtime	Speed-up	Cost	Red.
DNN	16	2032	**0.32**	$\mathbf{5.4 \times 10^5}$	4.12	12.81×	8.2×10^6	15.4×	1.27	3.96×	3.7×10^6	6.85×
	20	6214	**5.09**	$\mathbf{2.7 \times 10^7}$	73.06	14.36×	2.4×10^8	8.56×	25.858	5.1×	1.1×10^8	3.89×
	25	9644	**126.48**	$\mathbf{1.2 \times 10^9}$	2283.02	18.0×	1.7×10^{10}	13.4×	688.396	5.44×	5.3×10^9	4.32×
Quantum supremacy	20	4500	**7.68**	$\mathbf{4.7 \times 10^7}$	101.38	13.2×	3.8×10^8	8.04×	36.99	4.82×	2.9×10^8	6.19×
	24	5560	**90.90**	$\mathbf{9.0 \times 10^8}$	1050.07	11.55×	7.4×10^9	8.26×	581.64	6.40×	5.7×10^9	6.37×
	26	5990	**405.04**	$\mathbf{3.9 \times 10^9}$	3980.07	9.83×	3.2×10^{10}	8.21×	2568.12	6.34×	2.6×10^{10}	6.73×
Geometric mean			**19.70**	$\mathbf{1.2 \times 10^8}$	257.45	13.1×	1.2×10^9	9.94×	103.83	5.27×	6.7×10^8	5.59×

in a region (marked in blue) across the six largest quantum circuits, DNN ($n = 16, 20, 25$) and quantum supremacy ($n = 20, 24, 26$), with a line showing the average. As shown in Figure 14a and Figure 14b, our caching technique can reduce the computational cost as the number of threads increases. For instance, with 16 threads, we can achieve 13.53% reduction. A similar trend can be observed in the speed-up plot (Figure 14b). For example, our caching technique contributes to an average of 16.47% speed-up at 16 threads where the performance saturates (see Figure 12). This result highlights the effectiveness of our caching technique under different numbers of threads.

4.6 Evaluation of DMAV-Aware Gate Fusion

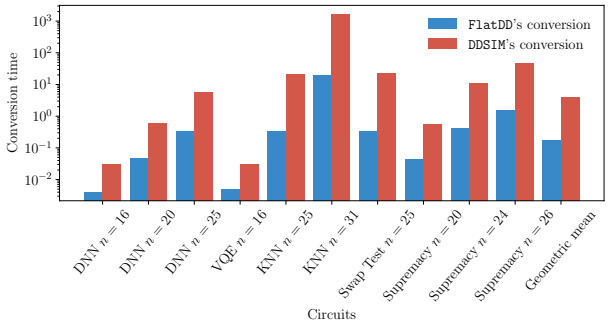
In this experiment, we evaluate the performance advantage of our gate-fusion algorithm by running `FlatDD` on the six deepest circuits (DNN and supremacy) with and without gate fusion. We compare the result with `k-operations` [100], a particularly effective gate-fusion algorithm designed for DD-based simulators. As shown in Table 2, our gate-fusion algorithm achieves 13.1× and 5.27× speed-up in terms of geometric mean when compared with `FlatDD` without gate fusion and `k-operations`, respectively. Our gate-fusion algorithm also reduces computational cost by 9.94× and 5.59× when compared with `FlatDD` without gate fusion and `k-operations`, respectively. This is because our gate-fusion algorithm always fuses a quantum gate that reduces the computational cost. The result in Table 2 shows the effectiveness of our DMAV-aware gate-fusion algorithm.

5 CONCLUSION

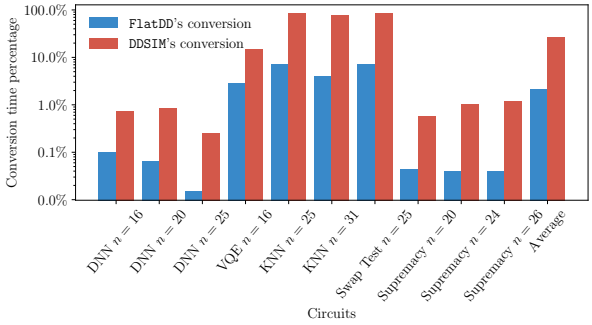
In this paper, we have introduced `FlatDD`, a parallel quantum circuit simulator that combines the strengths of DD-based and array-based simulators. Evaluated on commonly used quantum circuits, `FlatDD` has demonstrated 34.81× speed-up and 1.93× memory reduction compared to state-of-the-art simulators. Inspired by our success of GPU-accelerated computing [12–18, 24–33, 38–58, 61, 62, 64–67, 70–80, 83, 84, 92, 93, 96], our future work will enhance the simulation performance using GPU.

ACKNOWLEDGMENTS

This research was supported by ACCESS – AI Chip Center for Emerging Smart Systems (sponsored by InnoHK funding), the JC STEM Lab of Intelligent Design Automation (funded by the Hong Kong Jockey Club Charities Trust), Hong Kong SAR, and US NSF under awards 2235276, 2349144, 2349143, 2349582, and 2349141.

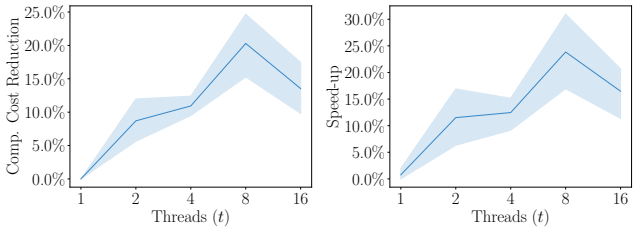


(a) Conversion time comparison.



(b) Conversion percentage in total runtime comparison.

Figure 13: Comparison between `FlatDD`'s parallel DD-to-array algorithm and `DDSIM`'s DD-to-array algorithm.



(a) Computational cost **(b) Speed-up**

Figure 14: Performance comparison between DMAV with caching and DMAV without caching over different numbers of threads on various quantum circuits.

REFERENCES

[1] [n. d.]. Cirq. https://quantumai.google/cirq
[2] [n. d.]. Multiply–accumulate operation. https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_operation
[3] [n. d.]. OpenMP. https://www.openmp.org/
[4] [n. d.]. Qiskit. https://qiskit.org/
[5] 2020. qsim. https://doi.org/10.5281/zenodo.4023103
[6] 2024. mqt-ddsim. https://github.com/cda-tum/mqt-ddsim
[7] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
[8] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *PACT*. IEEE, 116–128.
[9] Bela Bauer, Sergey Bravyi, Mario Motta, and Garnet Kin-Lic Chan. 2020. Quantum algorithms for quantum chemistry and quantum materials science. *Chemical Reviews* 120, 22 (2020), 12685–12717.
[10] Kerstin Beer, Dmytro Bondarenko, Terry Farrelly, Tobias J Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. 2020. Training deep quantum neural networks. *Nature communications* 11, 1 (2020), 808.
[11] Lukas Burgholzer and Robert Wille. 2020. Advanced equivalence checking for quantum circuits. *IEEE TCAD* 40, 9 (2020), 1810–1824.
[12] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE DAC*.
[13] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE HPEC*.
[14] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM HPDC*.
[15] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE DAC*.
[16] 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 *AMTE*.
[17] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM ICCAD*.
[18] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE IPDPS Workshop*.
[19] Vlad Gheorghiu. 2018. Quantum++: A modern C++ quantum computing library. *PloS one* 13, 12 (2018), e0208073.
[20] Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. 2002. Quantum cryptography. *Reviews of modern physics* 74, 1 (2002), 145.
[21] Thomas Grurl, Jürgen Fuß, Stefan Hillmich, Lukas Burgholzer, and Robert Wille. 2020. Arrays vs. decision diagrams: A case study on quantum circuit simulators. In *IEEE ISMVL*. 176–181.
[22] Thomas Grurl, Jürgen Fuß, and Robert Wille. 2022. Noise-aware quantum circuit simulation with decision diagrams. *TCAD* 42, 3 (2022), 860–873.
[23] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen. http://eigen.tuxfamily.org.
[24] 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 DAC*.
[25] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE TCAD* (2023).
[26] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM ICCAD*.
[27] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM DAC*.
[28] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM DATE*.
[29] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM ICCAD*.
[30] 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 DAC*.
[31] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM ICCAD*.
[32] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE TCAD* (2023).
[33] 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 DATE*.
[34] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *SC*. IEEE, 866–874.
[35] Dylan Herman, Cody Googin, Xiaoyuan Liu, Yue Sun, Alexey Galda, Ilya Safro, Marco Pistoia, and Yuri Alexeev. 2023. Quantum computing for finance. *Nature Reviews Physics* 5, 8 (2023), 450–465.
[36] Stefan Hillmich, Igor L. Markov, and Robert Wille. 2020. Just Like the Real Thing: Fast Weak Simulation of Quantum Computation. In *DAC*. 1–6.
[37] Stefan Hillmich, Alwin Zulehner, and Robert Wille. 2020. Concurrency in DD-based quantum circuit simulation. In *IEEE ASP-DAC*. 115–120.
[38] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM ICCAD*.
[39] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE ProTools*.
[40] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE HPEC*.
[41] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE IPDPS*.
[42] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD* (2021).

[43] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE HPEC*.

[44] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE DAC*.

[45] 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)*.

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

[47] 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 DAC*.

[48] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM ICCAD*.

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

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

[51] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE TPDS* (2022).

[52] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE HIPS*.

[53] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM ICCAD*.

[54] Tsung-Wei Huang and Martin Wong. 2016. UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. *IEEE TCAD* (2016).

[55] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM DAC*.

[56] Tsung-Wei Huang, P.-C. Wu, and Martin Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM ICCAD*.

[57] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE/ACM ICCAD*.

[58] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM ISPD*.

[59] J Stuart Hunter. 1986. The exponentially weighted moving average. *Journal of quality technology* 18, 4 (1986), 203–210.

[60] Intel. [n. d.]. AVX2. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intrinsics-for-avx2.html

[61] Shui Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE HPEC*.

[62] Shui Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM ICPP*.

[63] Chenyang Jiao, Weihua Zhang, and Li Shen. 2023. Communication Optimizations for State-vector Quantum Simulator on CPU+ GPU Clusters. In *ICPP*. 203–212.

[64] 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 DAC*.

[65] 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 ASPDAC*.

[66] 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 ICCAD*.

[67] 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 DAC*.

[68] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Heim, Martin Roetteler, and Sriram Krishnamoorthy. 2021. SV-Sim: scalable PGAS-based state vector simulation of quantum circuits. In *SC*. 1–14.

[69] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. 2023. Qasmbench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM TQC* 4, 2 (2023), 1–26.

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

[71] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE HPEC*.

[72] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE ICPADS*.

[73] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM GLSVLSI*.

[74] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE HPEC*. 1–7.

[75] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *Euro-Par*.

[76] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE TPDS* (2022).

[77] 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 *Euro-Par*.

[78] 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 ICPP*.

[79] 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 DAC*.

[80] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

[81] Ji Liu, Luciano Bello, and Huiyang Zhou. 2021. Relaxed Peephole Optimization: A Novel Compiler Optimization for Quantum Circuits. In *CGO*. 301–314.

[82] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2022. Qubit mapping and routing via MaxSAT. In *MICRO*. IEEE, 1078–1091.

[83] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM ASPDAC*.

[84] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE ESPM2*.

[85] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. 2023. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In *ASPLOS*. 252–265.

[86] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A Thornton, and Rolf Drechsler. 2015. QMDDs: Efficient quantum function representation and manipulation. *IEEE TCAD* 35, 1 (2015), 86–99.

[87] qpp. 2024. https://github.com/softwareQinc/qpp

[88] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking software and design automation tools for quantum computing. *Quantum* 7 (2023), 1062.

[89] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. 2016. qHiPSTER: The quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195* (2016).

[90] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *ASE*. 692–704.

[91] 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 *SC*. 1–24.

[92] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE HPEC*.

[93] 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 DAC*.

[94] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: hybrid partitioner based quantum circuit simulation system on GPU. In *ICS*. 443–454.

[95] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. 2020. Quantum computational advantage using photons. *Science* 370, 6523 (2020), 1460–1463.

[96] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM ASPDAC*.

[97] Alwin Zulehner, Stefan Hillmich, Igor L Markov, and Robert Wille. 2020. Approximation of quantum states using decision diagrams. In *ASP-DAC*. IEEE, 121–126.

[98] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to efficiently handle complex values? Implementing decision diagrams for quantum computing. In *IEEE ICCAD*. 1–7.

[99] Alwin Zulehner and Robert Wille. 2018. Advanced simulation of quantum computations. *IEEE TCAD* 38, 5 (2018), 848–859.

[100] Alwin Zulehner and Robert Wille. 2019. Matrix-vector vs. matrix-matrix multiplication: Potential in DD-based simulation of quantum computations. In *IEEE DATE*. 90–95.