# GPU Acceleration for Versatile Buffer Insertion

Yuan Pu[†*1,2], Yuhao Ji[†1], Siying Yu[1], Zuodong Zhang[3], Zizheng Guo[3], Zhuolun He[1,2], Yibo Lin[3], David Pan[4], Bei Yu[1],
[1]The Chinese University of Hong Kong [2]ChatEDA Tech [3]Peking University [4] University of Texas at Austin

***Abstract*—With the advancement of circuit design complexity and technology nodes, buffer insertion has become pivotal in mitigating timing violations, significantly impacting the physical design development cycle and highlighting the necessity for acceleration methodologies. In this paper, we present BIGX, a GPU-accelerated algorithmic framework for buffer insertion. BIGX is versatile and can be adapted to implement different dynamic programming (DP) based buffering algorithms for repairing various types of timing violations. In particular, we introduce MCDP, a dedicated DP-based buffering algorithm for repairing maximum capacitance violations, and propose a parallel version of Van Ginneken's algorithm for setup violations, both algorithms are incorporated and implemented in BIGX. Furthermore, to overcome the runtime limitations of DP-based buffering algorithms, BIGX adopts a distributed Branch Merge algorithm based on bucket sorting, which fully leverages the hierarchical memory architecture of modern GPUs to achieve substantial speedups while preserving solution quality. Experimental results on industrial benchmarks demonstrate that, with the integration of MCDP, BIGX repairs 96.6% of maximum capacitance violations. Compared to OpenROAD, BIGX with MCDP repairs 2.54x more maximum capacitance violations and delivers a 3.37x speedup. Additionally, BIGX accelerates the Van Ginneken's algorithm by 11.68x while maintaining comparable solution quality to its CPU-based counterpart.**

## I. Introduction

Buffer insertion is a widely adopted and effective technique for repairing timing violations, such as setup, hold, maximum capacitance, and maximum transition violations. For instance, setup violations caused by excessive interconnect delays can be mitigated by strategically inserting buffers to segment long wires into shorter sections, thereby reducing RC delay, and by isolating downstream capacitance to lower the load on the driver, which in turn decreases cell delay. As VLSI designs become increasingly complex and process nodes continue to scale, challenges such as rising interconnect delays, higher clock frequencies, and process variation uncertainties have intensified. To address these challenges, buffer insertion is repeatedly applied at multiple stages of the design flow to resolve a wide range of timing violations. However, this repeated application introduces significant computational overhead, making the process time-intensive. Moreover, the growing number of inserted buffers not only increases the design complexity but also leads to significant area consumption. These challenges underscore the need for an efficient and scalable buffer insertion framework that can effectively resolve timing violations while minimizing both runtime and area overhead in advanced technology nodes.

Buffer insertion has been extensively studied for VLSI timing closure, primarily to mitigate interconnection delay and address setup violations. Among these, Van Ginneken's algorithm [1] and its variants [2]–[8] are widely used. As shown in Fig. 1, these approaches typically construct a Steiner tree for the net, allocate candidate buffer locations along the tree segments, transform the tree into a binary routing tree, and use dynamic programming (DP) to determine optimal buffering solution. Some works [9]–[12] consider physical obstructions such as blockages or macros during buffer insertion, while others extend the scope from individual nets to path- [13] or network-level [14]–[16] optimizations. Beyond reducing delay, buffer insertion

[†]Two authors contributed equally.
* This work was performed during the first author's visit to UT Austin.

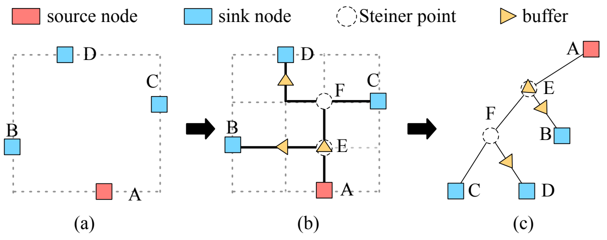


Fig. 1 Illustration of the buffer insertion process. (a) a net with 4 pins, (b) the buffered Steiner tree of the 4-pin net and (c) the buffered binary routing tree of the 4-pin net.

has also been applied to power planning [17], yield improvement [18], and hold violation repair [19], [20]. [21] conducted a case study for GPU accelerated buffer insertion on 2-pin nets, which is impractical to apply in real industrial scenario. Recently, BufFormer [22] explores the generative machine learning methods to improve the efficiency of buffer insertion. However, several challenges arise when applying these previous works to industrial scenarios of buffer insertion. First, although Van Ginneken's algorithm can be extended to handle certain DRV violations (e.g., maximum capacitance, transition, or fanout) by tracking these values during solution propagation and applying constraints during traceback, its reliance on pre-allocating candidate buffer locations before the DP process makes it difficult to balance efficiency and effectiveness in mitigating these violations. Second, the sequential CPU implementations of Van Ginneken's algorithm and its variances result in slow runtime for large-scale circuits. Third, path- and network-level methods suffer from poor scalability and high computational overhead due to their LP formulations. Lastly, machine learning-based methods often struggle with generalizability and transferability in real-world industrial contexts.

To address the aforementioned challenges, we propose BIGX, a novel GPU-accelerated, parallel and dynamic programming-based algorithmic framework for buffer insertion. Efficient parallelization of buffer insertion on GPUs is particularly challenging due to the inherent data dependencies and the intricate control divergence involved in the algorithm. To tackle these issues, BIGX adopts a topological-level parallelism scheme and divides the process into four sequential stages: Sink Initialization, Branch Merge, Source Solution Selection, and Solution Traceback. In each stage, nodes at the same topological level of the binary routing tree are processed in parallel, ensuring fine-grained parallelism and scalability. Among these stages, the Branch Merge stage, responsible for merging buffering solutions from two branches while maintaining Pareto Optimality, is the runtime bottleneck due to its $O(N^2)$ time complexity ($N$ is the number of solutions per branch). To overcome this, we develop a distributed and parallel Branch Merge algorithm that leverages the hierarchical memory architecture of GPUs. By employing a bucket sorting strategy in a distributed manner, the algorithm efficiently merges solution sets while maintaining uniform distribution of solutions on the Pareto front, significantly accelerating this stage without compromising solution quality. Additionally, BIGX is versatile and can be adapted to implement different DP-based buffering algorithms for repairing various types of timing violations. To repair maximum capacitance violations, we propose MCDP (Maximum Capacitance-driven Dynamic Programming algorithm). Unlike Van Ginneken's algorithm, which pre-allocates candidate buffer locations, MCDP dynamically calculates optimal buffer insertion locations along wire segments, resolving maximum capacitance violations with high computational efficiency. For repairing setup violations, we propose a parallel version of Van Ginneken's algorithm. Both MCDP and parallel Van Ginneken's algorithm are integrated into BIGX, enabling efficient and effective buffer insertion for large and complex circuit designs.

The major contributions of this paper are summarized as follows:

- We propose BIGX, the first systematic GPU-accelerated framework for buffer insertion. Following a novel topological-level parallelism scheme, BIGX adopts a GPU-friendly Branch Merge algorithm that resolves runtime bottlenecks while ensuring high-quality buffering solutions.
- BIGX is a versatile framework that integrates MCDP, a novel parallelizable, effective and efficient DP algorithm we propose for repairing maximum capacitance violations, and parallel Van Ginneken's algorithm for setup violation repair. Furthermore, BIGX can be extended to repair other types of timing violations.
- Experimental results show that BIGX, incorporating the MCDP algorithm, repairs 96.6% of maximum capacitance violations of the post-placement circuit designs, achieves a 3.37x speedup and repairs 2.54x more maximum capacitance violations compared to OpenROAD. Additionally, BIGX accelerates the Van Ginneken's algorithm by 11.68x over its CPU-based implementation.

## II. Preliminaries

### A. Timing Violations

In VLSI design, timing violations, including the design rule violations (DRVs) such as maximum capacitance/transition/fanout violations, along with setup and hold violations, are common challenges caused by high interconnect delay, excessive load, or improper signal timing.

- Max Cap violations occur when the total capacitance driven by a driver pin exceeds its limit, leading to degraded performance and signal integrity issues.
- Max Transition violations arise when signal rise/fall transition times are too slow due to excessive load or long interconnects, leading to potential timing failures in the circuit.
- Max Fanout violations happen when a driver pin drives too many sinks, increasing load capacitance and degrading signal speed.
- Setup/Hold violations occur when signals fail to meet timing requirements for data capture within flip-flops.

### B. Van Ginneken's Algorithm

Van Ginneken's algorithm [1] is the first systematic buffer insertion method based on dynamic programming. Using the Elmore delay model, it computes optimal buffering solutions to maximize required arrival time (RAT) and resolve setup violations. As illustrated in Fig. 1, for a given net, the algorithm first constructs a Steiner tree, allocates candidate buffering locations along the tree segments, and then builds a binary routing tree. Buffers can only be inserted at these pre-allocated candidate positions. The algorithm operates in two phases: bottom-up and top-down. In the bottom-up phase, candidate solutions are propagated from sinks to the source through three operations: wire insertion, buffer insertion, and branch merging. Each candidate solution tracks RAT and load capacitance. Wire insertion propagates the solution upstream, buffer insertion places a buffer at a candidate position, and branch merging combines solutions from child nodes to form a Pareto front. In the top-down phase, the optimal solution is selected and traced from the source to the sinks along the tree topology.

## III. GPU Accelerated Buffer Insertion Framework

### A. Motivation for Topology-Level Parallelism

Traditional DP-based buffering algorithms, such as Van Ginneken's algorithm [1], adopt a net-based approach that solves the buffer insertion problem for each net independently. Therefore, a natural strategy for GPU acceleration is net-level parallelism, where each GPU thread is assigned to process a single net. However, this approach faces two challenges: (1) the DP algorithm contains multiple interdependent steps with complex control divergence, which are difficult for a single thread to handle efficiently, and (2) variations in net sizes and routing tree depths lead to significant workload imbalance across threads, thus reducing parallel efficiency.

To overcome these challenges, we propose a topology-level parallelism scheme. This approach divides the DP algorithm into two phases: bottom-up and top-down, each consisting of several sequential stages. During the bottom-up (top-down) phase, the nodes in all nets' binary routing trees are processed in topological (reverse-topological) order. At each stage, nodes at the same topological level are processed in parallel. This scheme has two key advantages: (1) it facilitates parallel implementation by breaking the DP algorithm into fine-grained stages with straightforward, independent tasks and (2) it distributes the computation for large nets across multiple threads, ensuring better workload balance and higher parallel efficiency.

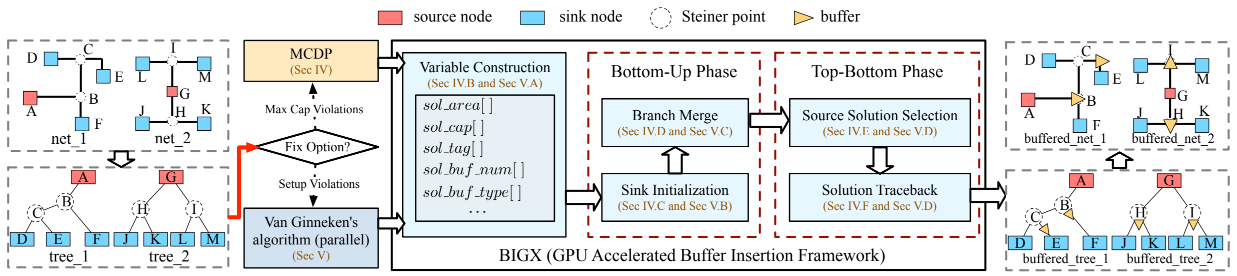
### B. Algorithmic Flow of BIGX

Fig. 2 illustrates the algorithmic flow of BIGX. Starting with the given nets to be buffered (e.g., net_1 and net_2), their binary routing trees are first constructed. Before buffering begins, a violation fix option is selected between Maximum Capacitance Violations and Setup Violations. Maximum capacitance violations are repaired using our proposed MCDP algorithm (detailed in Section IV), while a parallel version of Van Ginneken's algorithm (detailed in Section V) is applied for setup violations. Based on the selected option, the **Variable Construction** step allocates the necessary variables in GPU memory. The core of BIGX consists of four sequential stages. It starts with **Sink Initialization**, where buffering solutions (e.g., buffer type, count, total area, and load capacitance) for each sink's upstream branch are initialized. Next, in the **Branch Merge** stage, solutions for the two child branches of each Steiner point[1] are merged and pruned to maintain Pareto optimality. After processing all Steiner points in topological order, buffering solutions for upstream branches are stored in the dynamic programming (DP) table. In the final two stages, **Source Solution Selection** and **Solution Traceback**, buffering solutions for each downstream branch are determined in a top-down topological order. These stages optimize specific objectives based on whether the chosen violation fix option targets maximum capacitance or setup violations. Finally, the obtained buffering solutions obtained from BIGX will be applied to the nets. Fig. 3 visualizes the topology-level parallel implementation of BIGX on the two nets (net_1 and net_2) shown in Fig. 2.

## IV. MCDP: Maximum Capacitance Buffering Algorithm

In this section, we propose MCDP (Maximum Capacitance-driven Dynamic Programming algorithm), a parallel DP-based buffer insertion algorithm customized for repairing maximum capacitance violations. In the remainder of this section, we first gives the algorithmic

[1]During the implementation of BIGX, we directly use the Steiner point as the internal node of the binary routing tree.



source node
sink node
Steiner point
buffer
net_1
net_2
tree_1
tree_2
MCDP (Sec IV)
Max Cap Violations
Fix Option?
Setup Violations
Van Ginneken's algorithm (parallel) (Sec V)
Variable Construction (Sec IV.B and Sec V.A)
sol_area[ ]
sol_cap[ ]
sol_tag[ ]
sol_buf_num[ ]
sol_buf_type[ ]
Bottom-Up Phase
Branch Merge (Sec IV.D and Sec V.C)
Sink Initialization (Sec IV.C and Sec V.B)
Top-Bottom Phase
Source Solution Selection (Sec IV.E and Sec V.D)
Solution Traceback (Sec IV.F and Sec V.D)
BIGX (GPU Accelerated Buffer Insertion Framework)
buffered_net_1
buffered_net_2
buffered_tree_1
buffered_tree_2

Fig. 2 The algorithmic flow of BIGX, our proposed GPU accelerated buffer insertion framework.

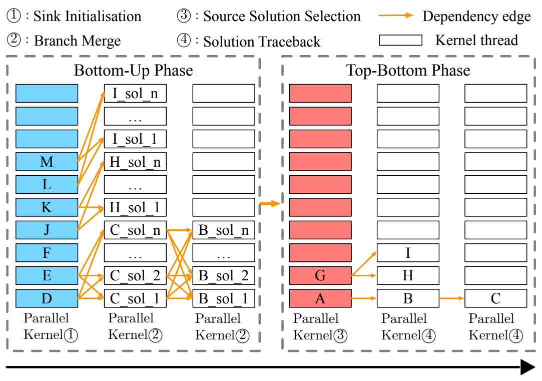


Fig. 3 Illustration of topology-level parallel implementation of BIGX on net_1 and net_2 in Fig. 2. The name on the kernel thread indicates the node being processed by the thread.

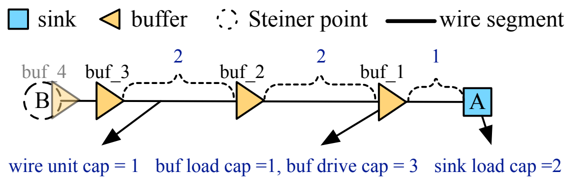


Fig. 4 Illustration of buffer insertion on a wire segment to eliminate maximum capacitance violations.

overview of MCDP, and then provide an in-depth discussion of the algorithmic techniques employed to integrate the MCDP algorithm into the BIGX framework.

### A. MCDP: Algorithmic Overview

To repair maximum capacitance violations, buffers are typically inserted along long wire segments to reduce excessive capacitive loads, ensuring each driver operates within its driving capability. Although Van Ginneken's algorithm is widely used for repairing setup violations, it is not well-suited for maximum capacitance violations due to its reliance on pre-allocated candidate buffering locations. Allocating fewer candidate locations risks leaving violations unresolved when the capacitance of long wire segments exceed the buffer's driving capacity, while allocating more locations significantly increases computational overhead. Striking a balance between effectiveness and efficiency becomes challenging for this algorithm.

To address the aforementioned limitations, we propose MCDP, a parallel DP-based buffer insertion algorithm to repair maximum capacitance violations. Unlike Van Ginneken's algorithm, which pre-allocates candidate buffering locations, MCDP dynamically determines optimal buffering locations along wire segments during the DP process. When a buffer is placed on a wire segment, MCDP ensures that its upstream driver has sufficient driving strength to drive both the wire segment's capacitance and the buffer's input load capacitance, guaranteeing that all maximum capacitance violations on buffered segments are resolved. To provide a larger design space, MCDP supports multiple buffer types. However, using multiple buffer types on the same wire segment increases algorithmic complexity and makes it challenging to derive a closed form of the optimal buffering solution. To balance efficiency and solution quality, MCDP restricts each wire segment to a single buffer type, while allowing different segments to use different buffer types. During the DP process, MCDP minimizes buffer area while resolving violations by placing buffers to precisely drive the cumulative load capacitance of downstream components and intermediate wire segments. In the optimal solution with a single buffer type per segment, the spacing between consecutive buffers remains a constant.

Similar to [3] and [23], we now derive the closed form of the buffer numbers and intervals for the optimal buffering solution. Given a sink with its upstream branch as shown in Fig. 4, we denote the load capacitance of the sink as $C_{\text{sink}}$, the wire capacitance of the branch segment as $C_{\text{wire}}$, the load/driving capacitance of the candidate type-$k$ buffer as $C^k_{\text{load}}$ and $C^k_{\text{drive}}$, and the wire capacitance for each unit length as $C_{\text{unit}}$. Starting from the sink node, the first inserted buffer should drive the accumulative capacitance load of the sink and the in-between wire. Therefore, the distance between the first type-$k$ buffer and the sink, denoted as $d^k_{\text{init}}$, can be calculated as:

$$d^k_{\text{init}} = \frac{C^k_{\text{drive}} - C_{\text{sink}}}{C_{\text{unit}}} \tag{1}$$

Moreover, the distance between the consecutively-inserted type-$k$ buffers, denoted as $d^k_{\text{buf}}$, can be calculated as:

$$d^k_{\text{buf}} = \frac{C^k_{\text{drive}} - C^k_{\text{load}}}{C_{\text{unit}}}, \tag{2}$$

where the inserted buffer can precisely drive the cumulative load capacitance of its downstream cell and the intermediate wire segment. Finally, combining $d^k_{\text{init}}$ and $d^k_{\text{buf}}$, we calculate the total number of inserted buffers (num_buf) for the optimal buffering solution on a wire segment in Equation (3):

$$\text{num\_buf} = \texttt{floor}\left(\frac{\frac{C_{\text{wire}}}{C_{\text{unit}}} - d^k_{\text{init}}}{d^k_{\text{buf}}}\right) + 1 = \texttt{floor}\left(\frac{C_{\text{wire}} + C_{\text{sink}} - C^k_{\text{load}}}{C^k_{\text{drive}} - C^k_{\text{load}}}\right), \tag{3}$$

where the usage of `floor` function guarantees the buffer is not inserted on the Steiner point. Fig. 4 shows one example: Suppose the load capacitance of the sink A is 2 units, the buffer can drive 3 units of capacitance and has a load capacitance of 1 unit, and the wire has a unit capacitance of 1. Then in the MCDP buffering solution, the first buffer (buf_1) would be placed 1 unit from sink A, while the distances between subsequent buffers (buf_1, buf_2 and buf_3) would be 2 units. Additionally, the MCDP algorithm supports two buffering

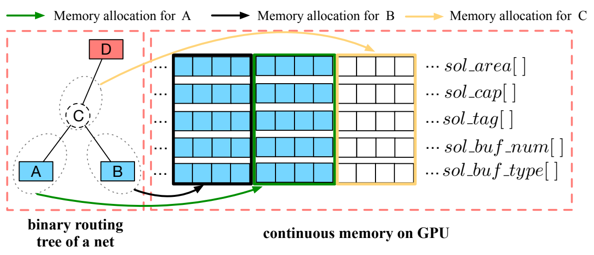


Fig. 5 Flattened and continuous memory allocation of variables for the MCDP algorithm on the GPU.

choices at Steiner points (for example, buf_4): inserting a buffer reduces upstream load capacitance but increases buffer area, while not inserting reduces area usage but increases upstream capacitance, enlarging the solution space of the MCDP algorithm.

### B. MCDP: Variable Construction

To enhance GPU memory access efficiency and improve the performance of the MCDP algorithm, we designed a set of flattened array variables to store the buffering solutions for each node and its upstream branch. Since the source node does not have an upstream branch, no additional variables are required for it. The buffering solution for each node is represented by five array variables: $sol\_area[\,]$, $sol\_cap[\,]$, $sol\_tag[\,]$, $sol\_buf\_num[\,]$, and $sol\_buf\_type[\,]$. Specifically, $sol\_area$ stores the total buffer area inserted along the node's upstream and downstream branches, $sol\_cap$ records the load capacitance perceived on the upstream Steiner point of the branch, $sol\_tag$ is a boolean variable indicating whether to insert a buffer at the upstream Steiner point, $sol\_buf\_num$ records the number of buffers inserted along the upstream branch, and $sol\_buf\_type$ records the type of buffers used. These five flattened array variables are stored continuously in GPU memory to maximize memory access efficiency. As shown in Fig. 5, the buffering solution variables for the sinks and Steiner points of a net are allocated in GPU memory.

### C. MCDP: Sink Initialization

Algorithm 1 describes the CUDA kernel implementation for the Sink Initialization stage, where each kernel thread is responsible for processing a sink. In the pseudo code, the `insert_solution` function is used to store buffering solutions in the appropriate indices of the array variables. The algorithm first calculates the wire capacitance between the sink and its parent Steiner point ($C_{\text{wire}}$) and the sink's intrinsic capacitance ($C_{\text{sink}}$) (line 3). If any candidate buffer type exists whose driving strength exceeds the sum of $C_{\text{wire}}$ and $C_{\text{sink}}$, a candidate buffering solution with no buffer inserted is generated (lines 4-6). The condition on line 4 ensures that any capacitance violation on the segment can be resolved by upstream buffering at higher topological levels. This solution is depicted in Fig. 6 (a), where the responsibility for repairing violations shifts to the upstream buffering, resulting in a larger load capacitance and zero area cost in the solution. Next, additional candidate buffering solutions are generated for each buffer type (lines 7-13). For each buffer type $k$, the algorithm computes the required number of buffers (line 9), the remaining load capacitance after buffering (line 10), and inserts the solution (line 11), as illustrated in Fig. 6 (b). An additional solution is created by placing an extra buffer at the Steiner point, updating the buffer count, and setting the remaining capacitance to $C^k_{\text{load}}$ (line 12), shown in Fig. 6 (c). By initializing a variety of buffering solutions with different trade-offs between area and load capacitance shown in Fig. 6, our algorithm significantly expands the solution space for the DP process, while ensuring capacitance violations on any wire segment can be resolved, either locally or through upstream buffering.

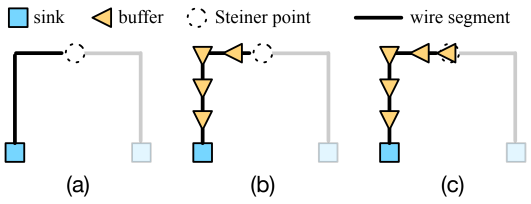


Fig. 6 Three buffer insertion candidate solutions generated for one sink (and its upstream branch) during the Sink Initialization Stage.

**Algorithm 1** Sink Solution Initialization

```
Input: $S$: #sinks, $B$: #buffer types.
Output: $sol\_area[\,]$, $sol\_cap[\,]$, $sol\_tag[\,]$, $sol\_buf\_num[\,]$,
    $sol\_buf\_type[\,]$: array variables for storing buffering solutions.
1: sink_id = blockIdx.x × blockDim.x + threadIdx.x
2: if sink_id > $S$ then return
3: Compute $C_{\text{wire}}$ (wire capacitance) and $C_{\text{sink}}$ (load capacitance)
4: if $\exists k \in [1..B]$ such that buffer $k$ has enough strength to drive
    $C_{\text{wire}} + C_{\text{sink}}$, then
5:     insert_solution(area = 0, cap = $C_{\text{wire}} + C_{\text{sink}}$, tag =
    False, buf_num = 0)    ▷ solution with no buffer
6: end if
7: for $k = 1$ to $B$ do
8:     Retrieve $C^k_{\text{load}}$ (load capacitance), $C^k_{\text{drive}}$ (drive capacitance)
    and $A^k$ (gate area) for buffer type $k$
9:     $n\_b$ = number of buffers calculated by Equation (3)
10:    left_cap = left capacitance after inserting $n\_b$ buffers
11:    insert_solution(area = $A^k \times n\_b$, cap = left_cap,
    tag = False, buf_num = $n\_b$, buf_type = $k$)  ▷ Solution without
    buffering at Steiner point
12:    insert_solution(area = $A^k \times (n\_b + 1)$, cap = $C^k_{\text{load}}$,
    tag = True, buf_num = $n\_b + 1$, buf_type = $k$) ▷ Solution with
    buffering at Steiner point
13: end for
```

### D. MCDP: Branch Merge

The Branch Merge stage is responsible for merging the solutions of the left and right branches at each Steiner point, inserting buffers into the upstream branch based on the merged results, and maintaining the Pareto front for all merged and buffered solutions. The Pareto front is characterized by two trade-off axes: capacitance (cap) on the x-axis and area on the y-axis, where adding more buffers (increasing area) reduces the perceived capacitance of the upstream branch. In DP-based buffering algorithms like Van Ginneken's algorithm, branch merge is a runtime bottleneck due to its $O(N^2)$ time complexity [1], where $N$ is the number of solutions per branch. Additionally, maintaining the Pareto front involves data dependencies among candidate solutions, making it challenging to parallelize.

To overcome the runtime bottleneck, we propose a **distributed, GPU-accelerated Branch Merge algorithm leveraging the memory hierarchy and massive parallelism of modern GPUs**. By applying bucket sorting, the task of Pareto Optimality maintenance is transformed into a distributed process, where each solution pair (from the left and right branches) combined with a specific buffer type is processed in parallel across all Steiner points at the same topological level. This algorithmic design significantly improves computational efficiency while maintaining a uniform and high-quality Pareto front.

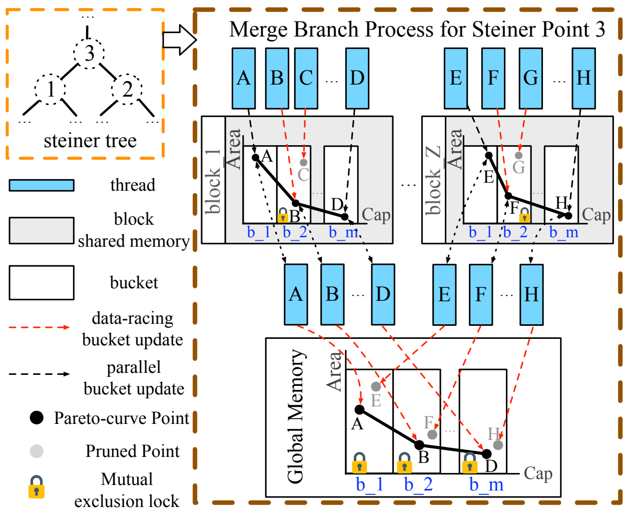


Fig. 7 Illustration of the distributed and GPU-accelerated bucket-sorting-based algorithm for Branch Merge.

Fig. 7 illustrates our Branch Merge algorithm: for each Steiner point (e.g., node 3), we assign $Z$ GPU blocks (e.g., block_1, block_2, ..., block_Z), each allocating shared memory to store its local Pareto front. Solutions are organized into buckets, each with an equal capacitance range, indexed by capacitance and valued by area. For instance, in block_1, solutions $B$ and $C$ are placed in the same bucket b_2 due to similar capacitance. Each thread within a block is responsible for handling the merge of a unique solution pair from the left and right branches, generating buffering solution on the upstream branch, and update the solution in the corresponding bucket in the shared memory for **local Pareto Optimality maintaining**. Specifically, when multiple threads try to insert solutions into the same bucket (e.g., threads handling solutions $B$ and $C$ in block_1), shared memory bank conflicts can occur. To handle this, we use **mutual exclusion locks (mutex) implemented with atomic operations** on the conflicting buckets. This ensures that all solutions are correctly inserted while inferior points (e.g., $C$) are pruned, maintaining the local Pareto front efficiently. After all threads finish updating, each block maintains its own Pareto front in shared memory. In the final step of the distributed Branch Merge algorithm, **all local Pareto fronts are merged into a global Pareto front**. For each solution on a block's Pareto front, a thread inserts it into the corresponding bucket in global memory in parallel. If multiple threads attempt to insert to the same bucket, mutex locks are applied again to resolve conflicts and prune inferior points, maintaining the global Pareto front. By leveraging atomic operations in shared memory, which are significantly faster than in global memory, our algorithm efficiently constructs distributed Pareto fronts while **avoiding the overhead of frequent atomic operations in global memory**, greatly improving computational efficiency.

Algorithm 2 shows the topology-parallel process of Branch Merge for all nets. All the Steiner points are first sorted topologically, and following the order from sinks to source, these Steiner points are segmented into $L$ levels (line 1). The solutions of all Steiner points in the same topology level are merged by the `branch_merge_kernel` function in parallel. Denote the number of solution for each Steiner point (with its upstream branch) as $N$ and the number of buffer type as $B$, there will be $N \times N \times B$ solution pairs to process during the Branch Merge step for each Steiner point. Given the block size as $BZ$, we calculate the corresponding grid size (denoted as $GZ$) in lines 3-

**Algorithm 2** Parallel Branch Merge for All Nets

```
Input: $B$ as #buffer types, $BZ$ as the block size, $N_{sp}$ as #Steiner points of all nets
Input: $SP[0..N_{sp}]$, list of all Steiner points
1: Sort $SP$ by topological order from sinks to sources $\rightarrow L$ levels
2: for $l \in [1, 2..L]$ do
3:     $N_l$ = #Steiner points for all nets in topological level $l$.
4:     $GZ = \frac{N_l \times N \times N \times B}{BZ}$ ▷ The grid size of branch merge kernel
5:     branch_merge_kernel<<< $GZ, BZ$ >>>($l$)  ▷ to maintain local Pareto front in the shared memory of each block.
6:     Integrate local Pareto fronts of each block into one global Pareto front in the global memory.
7: end for
```

4 to guarantee that the merging process for each solution pair with one specific buffer type can be handled by a unique GPU thread in parallel. The `branch_merge_kernel` function merges branching solutions and maintain local Pareto front in the share memory of each block (line 5), and finally the local Pareto fronts distributed in each block's share memory will be integrated and pruned to construct the ultimate Pareto front for each Steiner point.

As outlined in Algorithm 3, the `branch_merge_kernel` function begins by identifying the Steiner point ID (point_id), buffer type ID (buf_id), and the indices of the left and right child solutions (l_sol and r_sol) (lines 1-5). Each thread is designed to handle a unique pair of l_sol and r_sol with a unique buffer type (buf_id), ensuring all solution combinations for a given point_id are processed. Subsequently, the solutions from l_sol and r_sol are merged, and the resulting total capacitance ($C_{\text{total}}$) and total area ($A_{\text{total}}$) are computed (lines 6-8). To efficiently manage the merged solutions, the algorithm uses the `update_bucket` function. This function inserts each buffering solution into the appropriate bucket, removes inferior solutions, and maintains the local Pareto front in the shared memory of the GPU block. For Pareto optimality, a solution $x$ is considered inferior to another solution $y$ if both $A^x > A^y$ and $C^x > C^y$. Similar to the stage of Sink Initialization (line 4 in Algorithm 1), if there exists any buffer type whose driving strength exceeds the merged capacitance $C_{\text{total}}$, the candidate buffering solution with no buffer inserted will be generated and updated in the bucket (lines 9-11), ensuring the merged capacitance can be driven by upstream buffering solutions and enlarging the solution space in the DP process. Next, referring to Equation (3), the number of inserted buffers ($n\_b$) and the remaining load capacitance (left_cap) are then calculated (lines 13-14). This solution, without buffering at the upstream Steiner point, is inserted into shared memory (line 15). Finally, based on the previously inserted solution, a new solution with one additional buffer at the upstream Steiner point is generated and inserted into the shared memory (line 16). Note that in this function, the l_sol and r_sol values for each solution will be stored and used in the Solution Traceback stage (Section IV-F).

Although the overall complexity of our proposed Branch Merge algorithm remains $O(N^2)$, our distributed algorithmic design, which maintains local Pareto fronts in each block's shared memory and subsequently integrates them into a global Pareto front, enables high degrees of GPU thread-level parallelism for acceleration. Furthermore, during the bucket sorting process, we set an appropriate value of capacitance range of all buckets by observing the solution distribution, and logarithmically scale the capacitance values of solutions to mitigate the long tail distribution, ensuring a uniform distribution of points across all buckets. By leveraging bucket-based merging

**Algorithm 3** `branch_merge_kernel`

```
Input: $P$: #nodes, $N$: #solutions per branch, $B$: # buffer types
Input: $l$: topological level of Steiner points
Output: $sol\_area[]$, $sol\_cap[]$, $sol\_tag[]$, $sol\_buf\_num[]$, $sol\_buf\_type[]$: arrays for buffering solutions
 1: tid ← threadIdx.x + blockDim.x × blockIdx.x
 2: point_id ← tid/$(B \times N \times N)$, buf_id ← tid mod $B$
 3: if point_id is not in level $l$ then return
 4: l_child, r_child ← left and right children of point_id
 5: l_sol, r_sol ← unique solution indices of l_child and r_child
 6: Retrieve wire capacitance $C_{\text{wire}}$ between point_id and its parent
 7: $C_{\text{total}} \leftarrow sol\_cap[\text{l\_sol}] + sol\_cap[\text{r\_sol}] + C_{\text{wire}}$    ▷ Merged capacitance
 8: $A_{\text{total}} \leftarrow sol\_area[\text{l\_sol}] + sol\_area[\text{r\_sol}]$    ▷ Merged area
 9: if $\exists k \in [1..B]$ such that buffer $k$ has enough strength to drive $C_{\text{total}}$ then
10:     update_bucket(area = $A_{\text{total}}$, cap = $C_{\text{total}}$, tag = False, buf_num = 0)    ▷ Solution with no buffer inserted
11: end if
12: Retrieve $C^b_{\text{load}}$ (load capacitance), $C^b_{\text{drive}}$ (drive capacitance) and $A^b$ (gate area) for buffer type buf_id
13: $n\_b$ = number of buffers calculated by Equation (3)
14: $left\_cap$ = left capacitance after inserting $n\_b$ buffers
15: update_bucket(area = $A_{\text{total}} + A^b \times n\_b$, cap = left_cap, tag = False, buf_num = $n\_b$, buf_type = buf_id)    ▷ Solution without buffering at Steiner point
16: update_bucket(area = $A_{\text{total}} + A^b \times (n\_b + 1)$, cap = $C^b_{\text{load}}$, tag = True, buf_num = $n\_b + 1$, buf_type = buf_id)    ▷ Solution with buffering at Steiner point
```

to construct the Pareto optimal curve in the global memory (as shown in Fig. 7), the resulting Pareto curve achieves a more uniform distribution of values, guaranteeing good performance during the Branch Merge stage.

### *E. MCDP: Source Solution Selection*

At this stage, each source node across all nets is processed in parallel, with one GPU thread assigned per source node to independently determine its optimal buffering solution. For each source, the algorithm first retrieves its driving capacitance ($C_{\text{source}}$). If the source has two child branches, all valid combinations of buffering solutions from both children are exhaustively evaluated. Each solution pair is checked to ensure that the combined load capacitance does not exceed $C_{\text{source}}$. Combinations violating this constraint are discarded, as they cannot satisfy the maximum capacitance requirement at the source. Among the valid candidates, the solution pair with the **minimum area** is selected as the optimal buffering configuration for the source. The selected solution pair is stored for use in the subsequent traceback stage. In cases where a source node has only a single child branch, the process simplifies to evaluating and selecting the minimal-area solution from that branch alone, following the same pruning criteria.

### *F. MCDP: Solution Traceback*

After the Source Solution Selection stage, the buffering solution selected for each source node's child Steiner point has been determined, along with the specific solutions from the preceding topology level that are merged to form it. By **leveraging the dependency relationships between solutions across different topology levels** utilizing the saved l_sol and r_sol values introduced at line 5 of Algorithm 3, we can traverse all Steiner points and sinks in topological order, from sources to sinks, to obtain their final buffering solutions. In the GPU-accelerated implementation of the Solution Traceback stage, similar to Algorithm 2, we first perform a topological sorting of all nodes except the source. Following the order from sources to sinks, it is known that while there is a dependency between solutions across different levels, solution tracebacks within the same topology level are independent of each other. Hence, solution traceback operations within the same topology level are executed in parallel, whereas solution tracebacks across different levels are executed sequentially. After the Solution Traceback stage, the buffering solutions for all Steiner points and sinks are fully determined, marking the completion of MCDP.

## V. Parallel Van Ginneken's Algorithm

In this subsection, we propose a parallel version of Van Ginneken's algorithm and incorporate it into the BIGX framework. Before the DP process, all Steiner points in the Steiner tree of the nets are allocated as candidate buffering locations. Similar to MCDP, the parallel Van Ginneken's algorithm follows the same four-stage algorithmic flow in BIGX: Sink Initialization, Branch Merge, Source Solution Selection, and Solution Traceback. However, it introduces additional variables and adjusts certain algorithmic details to meet its specific needs. Due to space constraints, a complete description of Van Ginneken's algorithm's implementation is omitted. Instead, we build upon the algorithmic flow of MCDP to explain how this algorithm is incorporated into BIGX.

### *A. Parallel Van Ginneken's algorithm: Variable Design*

Different from MCDP which maintains only area and capacitance values on the Pareto front of buffering solutions, Van Ginneken's algorithm aims to minimize the slack of the entire net, or equivalently, maximize the source's required arrival time (RAT). To achieve this, in addition to the variables introduced in Fig. 5, we introduce a new variable, $sol\_rat$. Similar to the variables $sol\_area$ and $sol\_cap$ in Fig. 5, the purpose of $sol\_rat$ is to store the RAT value associated with each buffering solution.

### *B. Parallel Van Ginneken's algorithm: Sink Initialization*

The Sink Initialization process of Van Ginneken's algorithm follows the algorithmic flow described in Algorithm 1, with two key technical differences. The first difference lies in the initialization of the required arrival time (RAT) during solution insertion. Specifically, let the original RAT value of the sink be $Q_{\text{sink}}$ and the delay of the intermediate wire between the sink and its upstream Steiner point be $D(e)$. The resulting RAT value, denoted as $Q_u$, for the case where no buffer is inserted at the upstream branch (corresponding to line 5 in Algorithm 1), is computed as:

$$Q_u = Q_{\text{sink}} - D(e). \tag{4}$$

This updated RAT value, $Q_u$, should then be inserted into the solution at line 5 of Algorithm 1. The second difference is that, unlike MCDP, which allows buffers to be flexibly inserted along the wire segment, Van Ginneken's algorithm restricts buffer insertion to the pre-allocated candidate buffering positions only. For each buffer type $k$, a solution is generated by inserting the buffer at the upstream Steiner point. After the buffer is inserted, the corresponding RAT value, denoted as $Q_{u'}$, is updated as follows:

$$Q_{u'} = Q_u - D(k), \tag{5}$$

where $D(k)$ is the gate delay of buffer type $k$, obtained using the NLDM delay model. Finally, the updated solution, including the newly calculated RAT value for each buffer type $k$, is inserted into the solution set, following the same procedure as line 12 in Algorithm 1.

### *C. Parallel Van Ginneken's algorithm: Branch Merge*

For a given Steiner point to be merged, the Branch Merge stage combines the candidate solutions from its branches to construct the

TABLE I Benchmark statistics.

| Benchmark | #Macros | #Cells | #Nets | Period (ns) | Tech node (nm) |
|---|---|---|---|---|---|
| Vortex | 43 | 113175 | 122705 | 2.0 | 14 |
| Gemmini | 737 | 926064 | 980634 | 2.0 | 14 |
| Vortex-Large | 376 | 1020677 | 1098249 | 3.0 | 14 |
| Nvdla-Large | 80 | 1539105 | 1695098 | 5.0 | 14 |
| Nvdla-Small | 108 | 269289 | 288877 | 2.0 | 14 |
| LargeBoom | 636 | 736553 | 772344 | 2.0 | 14 |
| SmallBoom | 314 | 296339 | 314287 | 2.0 | 14 |
| NVDLA-Small | 108 | 330761 | 352942 | 3.0 | 28 |
| Pulpino | 3 | 18583 | 19457 | 3.0 | 28 |
| XScore | 176 | 4295515 | 4530869 | 5.0 | 28 |

Pareto front. The implementation of the Branch Merge stage in Van Ginneken's algorithm generally follows the algorithmic flows outlined in Algorithm 2 and Algorithm 3, with several key differences. Unlike MCDP, which uses a 2-dimensional Pareto Optimality curve, the Pareto curve in Van Ginneken's algorithm is 3-dimensional, incorporating area ($A$), capacitance ($C$), and required arrival time ($Q$). For Pareto Optimality, a buffering solution $x$ is inferior to solution $y$ if all three conditions, $A^x > A^y$, $C^x > C^y$, and $Q^x < Q^y$, are satisfied simultaneously. To accommodate this, the `UPDATE_BUCKET` function in Algorithm 3 is modified to maintain the 3D Pareto front. Additionally, since buffer insertion is restricted to Steiner points, the buffering procedure follows a similar approach to that described in Section V-B. Specifically, the unbuffered and buffered required arrival time (RAT) values are computed using Equation (4) and Equation (5), respectively. These solutions are then inserted into the bucket to construct the 3D Pareto Optimality curve.

### *D. Parallel Van Ginneken's algorithm: Source Solution Selection & Solution Traceback*

The Source Solution Selection stage in Van Ginneken's algorithm is similar to the corresponding stage in MCDP, with the key difference being the optimization objective. Van Ginneken's algorithm focuses on maximizing the required arrival time (RAT), selecting the solution with the largest RAT value during the iteration process. The Solution Traceback stage is identical to that of MCDP, with algorithmic details provided in Section IV-F.

## VI. Discussion on Versatility

In this section, we explore extending BIGX to repair other types of timing violations beyond maximum capacitance and setup, including maximum transition, maximum fanout (two types of design rule violations), and hold violations. Due to space constraints, we focus on the general concepts without discussing technical or implementation details. To configure BIGX to fix maximum fanout violations, the fanout count for each sub-tree is tracked during the Sink Initialization and Branch Merge stages. During the Branch Merge step, solutions that exceed the maximum fanout limit are pruned. Meanwhile for maximum transition violations, the repair process is similar to the MCDP implementation. Using a timing model to estimate the transition time at sink gates and buffers, buffers can be uniformly inserted along wire segments to resolve transition violations for the sinks and inserted buffers. Finally, hold violations, caused by insufficient interconnection delay, are repaired by distributing the required delay increment across all nets along the path. In this case, BIGX's configuration is similar to Van Ginneken's algorithm, with the key difference being that the Source Solution Selection stage chooses solutions satisfying the distributed delay requirements.

## VII. Experimental Results

### *A. Experimental Setting*

We implement BIGX using C++ and CUDA kernels, use Flute [24] for Steiner tree construction, and run the program on a Linux server with a 64-core dual-socket Intel Xeon Platinum 8358 CPU at 2.60 GHz, 256GB RAM and an Nvidia A800 GPU with 80 GB of memory. For evaluation, we select 10 industrial circuit designs from CircuiNet [25], [26], Chipyard [27] and Xiangshan [28] as benchmarks. TABLE I summarizes the statistics of these test cases, which encompass designs with diverse functionalities and sizes. These designs are synthesized using either a 14nm or 28nm commercial technology node, reflecting real-world industrial circuit design scenarios. For the BIGX implementation of both and MCDP and parallel Van Ginneken's algorithm, the solution size of flatten variables introduced in Section IV-B and Section V-A is set to be 64 for each node, and all CUDA kernels are run with 512 threads in each CUDA thread block.

### *B. Overall Comparisons*

To evaluate the effectiveness and efficiency of the proposed BIGX framework and the MCDP algorithm in repairing maximum capacitance violations, we compare our approach against OpenROAD [29]. In the OpenROAD flow, the `repair_design` command is used to insert buffers and resize gates to mitigate capacitance violations. For a fair comparison, we disable gate sizing and configure `repair_design` to perform buffer insertion only, which serves as our baseline. TABLE II presents the experimental results, including WNS, TNS, the number of maximum capacitance violations (#max cap), and leakage power, all reported using Cadence Innovus. The "Post-Placement" column reports the metrics of circuit designs placed using DREAMPlace [30], while the second and third columns show the results after applying OpenROAD's `repair_design` and our BIGX-based MCDP implementation, respectively, on the placed designs in the "Post-Placement" column.

The experimental results in TABLE II demonstrate that our proposed MCDP algorithm, integrated into the BIGX framework, successfully repairs 96.6% of maximum capacitance violations in Post-Placement circuit designs. Compared to OpenROAD's `repair_design`, it repairs 2.54x more maximum capacitance violations and delivers a substantial 3.37× speedup. While MCDP significantly improves violation repair, some residual maximum capacitance violations persist in certain designs. This is primarily attributed to RC estimation differences between BIGX's internal timing engine and the Cadence Innovus timer, resulting in inconsistencies between the violation repair process (performed by BIGX) and the final violation reporting (performed by Innovus). Additionally, for designs such as Vortex and Gemmini, OpenROAD inserts considerably more buffers than BIGX-MCDP. Although this leads to better WNS and TNS metrics, it is less effective in addressing maximum capacitance violations. Moreover, this aggressive buffering incurs a 1.4% increase in leakage power compared to our approach.

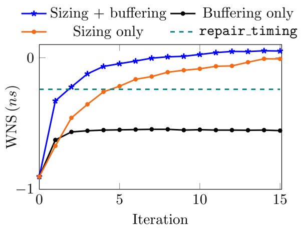
To evaluate the speedup of BIGX in implementing the parallel Van Ginneken's algorithm (referred to as Van Ginneken's algorithm-BIGX) for setup violation repair, we reproduce a single-threading CPU-based reference version (Van Ginneken's algorithm-CPU) following [1]. In Van Ginneken's algorithm-CPU, the number of solution candidates grows exponentially with the topological level during branch merging, leading to high memory and computational overhead. To ensure practicality, we limit the number of solutions at each node to 128. Following standard timing closure procedures, we first apply BIGX-MCDP to fix maximum capacitance violations and obtain Post-DRV results. Based on the Post-DRV results, we then run Van Ginneken's algorithm-CPU and Van Ginneken's algorithm-BIGX separately to repair setup violations for 10% nets of the design with the worst setup slack values. As shown in TABLE III, Van Ginneken's algorithm-BIGX achieves a 11.68x speedup over Van Ginneken's algorithm-CPU while maintaining comparable performance. Notably,

TABLE II Performance of buffering methods for repairing the maximum capacitance violations.

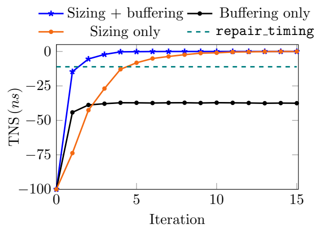
| Benchmark | Post-Placement | | | | OpenROAD: `repair_design` | | | | | BIGX-MCDP (ours) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WNS (ns) | TNS (ns) | #max cap | power (mW) | WNS (ns) | TNS (ns) | #max cap | power (mW) | runtime (s) | WNS (ns) | TNS (ns) | #max cap | power (mW) | runtime (s) |
| Vortex | -1.636 | -1255.0 | 1682 | **2.003** | **-0.898** | -108.2 | 80 | 2.085 | 22 | -0.904 | **-100.0** | **1** | 2.055 | **14** |
| Gemmini | -4.432 | -4235.1 | 4192 | **16.241** | **0.221** | **0** | 2959 | 16.666 | 212 | -4.044 | -3463.0 | **44** | 16.326 | **35** |
| Vortex-Large | -4.686 | -5585.5 | 13411 | **16.471** | -1.134 | **-150.1** | 724 | 17.053 | 234 | **-1.116** | -160.0 | **0** | 16.811 | **115** |
| Nvdla-Large | -27.532 | -276000.0 | 45651 | **28.663** | -4.832 | **-4752.8** | 849 | 32.418 | 404 | **-3.964** | -5589.5 | **521** | 31.037 | **197** |
| Nvdla-Small (14nm) | -7.202 | -34490.1 | 6961 | **6.013** | -1.166 | -1263.6 | 239 | 6.307 | 51 | **-0.895** | **-621.8** | **0** | 6.180 | **36** |
| LargeBoom | -0.579 | -850.6 | 1452 | **13.323** | **0.024** | **0** | 1287 | 13.346 | 123 | -0.586 | -638.4 | **158** | 13.348 | **26** |
| SmallBoom | 0.002 | 0 | 453 | **5.904** | 0.554 | 0 | 900 | 5.993 | 59 | 0.001 | 0 | **19** | 5.913 | **11** |
| NVDLA-Small (28nm) | **-0.628** | -89.5 | 2643 | **6.948** | -0.879 | -516.8 | 2189 | 6.993 | 51 | -0.633 | **-57.2** | **319** | 6.981 | **20** |
| Pulpino | -0.588 | -152.5 | 108 | **1.671** | -1.232 | -430.1 | 28 | 1.673 | 3 | **-0.569** | **-22.2** | **1** | 1.673 | **1** |
| XScore | -7.013 | -22172.6 | 49863 | **119.434** | -6.551 | -154000.0 | 68488 | 119.813 | 1251 | **-1.700** | **-9409.7** | **1970** | 119.680 | **252** |
| Normalize | 1.000 | 1.000 | 1.000 | **1.000** | 0.656 | 1.671 | 0.619 | 1.031 | 3.374 | **0.620** | **0.392** | **0.034** | 1.017 | **1.000** |

TABLE III Performance comparison of buffering methods for repairing setup violations based on post-DRV results. The post-DRV results are generated by applying BIGX-MCDP to the post-placement circuit designs. In these experiments, Van Ginneken's algorithm-CPU and Van Ginneken's algorithm-BIGX are applied to the 10% of nets with the worst setup slack values.

| Benchmark | Post-DRV | | | Van Ginneken's algorithm-CPU | | | | Van Ginneken's algorithm-BIGX (ours) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | WNS (ns) | TNS (ns) | power (mW) | WNS (ns) | TNS (ns) | power (mW) | runtime (s) | WNS (ns) | TNS (ns) | power (mW) | runtime (s) |
| Vortex | -0.904 | -100.0 | **2.055** | **-0.641** | -55.0 | 2.131 | 216 | -0.658 | **-49.6** | 2.224 | **14** |
| Gemmini | -4.044 | -3463.0 | **16.326** | **-0.736** | -512.1 | 16.595 | 214 | -0.767 | **-444.0** | 16.666 | **45** |
| Vortex-Large | -1.116 | -160.0 | **16.811** | -0.498 | -21.645 | 17.826 | 1855 | **-0.423** | **-15.9** | 18.855 | **134** |
| Nvdla-Large | -3.964 | -5589.5 | **31.037** | -3.935 | -5142.1 | 31.598 | 1802 | **-3.885** | **-5130.8** | 32.307 | **150** |
| Nvdla-Small (14nm) | -0.895 | -621.8 | **6.180** | -0.809 | -333.447 | 6.309 | 346 | **-0.793** | **-308.2** | 6.434 | **27** |
| LargeBoom | -0.586 | -638.4 | **13.348** | -0.156 | -247.0 | 13.625 | 714 | **-0.128** | **-190.573** | 13.860 | **51** |
| SmallBoom[2] | 0.001 | 0 | 5.913 | 0.001 | 0 | 5.913 | N/A | 0.001 | 0 | 5.913 | N/A |
| Nvdla-Small (14nm) | -0.633 | -57.2 | **6.981** | **-0.427** | -26.7 | 7.025 | 310 | -0.495 | **-19.0** | 7.205 | **23** |
| Pulpino | -0.569 | -22.2 | **1.673** | -0.450 | -9.0 | 1.680 | 31 | **-0.449** | **-6.6** | 1.710 | **2** |
| XScore | -1.700 | -9409.7 | **119.680** | -0.965 | -8592.8 | 119.874 | 563 | **-0.878** | **-8118.3** | 119.983 | **173** |
| Normalize | 1.000 | 1.000 | **1.000** | 0.615 | 0.496 | 1.021 | 11.676 | **0.608** | **0.436** | 1.045 | **1.000** |



(a)



(b)

Fig. 8 Analysis of different optimization flows for repairing setup violations on **Vortex**: (a) WNS results and (b) TNS results.

the proposed distributed Branch Merge algorithm in BIGX enables a more uniform distribution of solutions along the Pareto curve. This often leads to the selection of solutions with larger required arrival times (and consequently larger area) during the Source Solution Selection stage. As a result, Van Ginneken's algorithm-BIGX shows slight improvements of 0.7% in WNS and 6.0% in TNS, at the cost of 2.4% higher leakage power compared to Van Ginneken's algorithm-CPU.

Based on the average runtime analysis across the experiments in TABLE II and TABLE III, we observe that for BIGX-MCDP, Van Ginneken's algorithm-CPU, and Van Ginneken's algorithm-BIGX, over 99% of the total runtime is consumed by the Branch Merge stage, underscoring the importance of an efficient parallel algorithm design for this step.

### C. Timing Optimization with Buffering and Gate Sizing

While buffering is a widely used technique for timing optimization, it is typically employed in conjunction with gate sizing in industrial design flows. These two methods are applied iteratively to effectively eliminate timing violations. The rationale behind this iterative integration is that buffering introduces additional gates into the design, which subsequently become candidates for gate sizing. In turn, gate sizing modifies the load capacitance distribution along timing paths, potentially revealing new opportunities for buffering. To validate the effectiveness of BIGX within a realistic timing optimization flow, we implement the sensitivity-guided greedy sizing algorithm [31], and compare timing optimization results across four flows: (1) iterative Van Ginneken's algorithm-BIGX for buffering only, (2) iterative gate sizing only, (3) interleaved execution of Van Ginneken's algorithm-BIGX and gate sizing (Sizing + Buffering), and (4) the `repair_timing` command in OpenROAD, which applies both buffering and sizing for setup violation repair. Experiments are conducted on the Post-DRV results of Vortex. In each buffering iteration, Van Ginneken's algorithm-BIGX targets the top 10% worst-slack nets, while sizing iteratively adjusts gates on the top 200 worst-slack paths. Each flow runs for 15 iterations. As shown in Fig. 8, the "Buffering only" flow exhibit diminishing improvements and remain unsolved timing violations over iterations. Moreover, although both "Sizing only" and "Sizing + Buffering" eliminate WNS and TNS violations at the end and achieves timing performance better than OpenROAD, the interleaved implementation between buffering and sizing speedup the timing closure process with less number of iterations compared with the "Sizing only" flow. These results demonstrate the effectiveness of BIGX in realistic industrial timing optimization scenarios.

## VIII. Conclusion

This paper introduces BIGX, a GPU-accelerated framework for buffer insertion. To address runtime bottlenecks, it employs a parallel and distributed branch merge algorithm based on bucket sorting. BIGX is versatile, supporting the repair of multiple timing violations by integrating MCDP for maximum capacitance fixes and parallel Van Ginneken's algorithm for setup violation repair. Experimental results show that BIGX with MCDP repairs 2.54x more maximum capacitance violations and achieves a 3.37x speedup over OpenROAD, while accelerating Van Ginneken's algorithm by 11.68x compared to its CPU implementation.

[2]For SmallBoom, the setup WNS and TNS of the post-DRV result are already positive, so Van Ginneken's algorithm-CPU and Van Ginneken's algorithm-BIGX are not executed for this case.

## References

[1] L. P. Van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *1990 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1990, pp. 865–868.

[2] J. Lillis, C.-K. Cheng, and T.-T. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, 1996.

[3] C. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," in *Proceedings of the 34th Annual Design Automation Conference*, 1997, pp. 588–593.

[4] C. J. Alpert, A. Devgan, and S. T. Quay, "Buffer insertion with accurate gate and interconnect delay computation," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 479–484.

[5] W. Shi and Z. Li, "An O (nlogn) time algorithm for optimal buffer insertion," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 580–585.

[6] Z. Li, C. Sze, C. J. Alpert, J. Hu, and W. Shi, "Making fast buffer insertion even faster via approximation techniques," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, 2005, pp. 13–18.

[7] W. Shi and Z. Li, "A fast algorithm for optimal buffer insertion," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 879–891, 2005.

[8] M. Waghmode, Z. Li, and W. Shi, "Buffer insertion in large circuits with constructive solution search techniques," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 296–301.

[9] H. Zhou, D. Wong, I.-M. Liu, and A. Aziz, "Simultaneous routing and buffer insertion with restrictions on buffer locations," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 96–99.

[10] M. Lai and D. Wong, "Maze routing with buffer insertion and wiresizing," in *Proceedings of the 37th Annual Design Automation Conference*, 2000, pp. 374–378.

[11] C. J. Alpert, J. Hu, S. S. Sapatnekar, and C. Sze, "Accurate estimation of global buffer delay within a floorplan," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 6, pp. 1140–1145, 2006.

[12] J. Cong, T. Kong, and D. Z. Pan, "Buffer block planning for interconnect-driven floorplanning," in *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051)*. IEEE, 1999, pp. 358–363.

[13] C. N. Sze, C. J. Alpert, J. Hu, and W. Shi, "Path based buffer insertion," in *Proceedings of the 42nd annual Design Automation Conference*, 2005, pp. 509–514.

[14] I.-M. Liu, A. Aziz, D. Wong, and H. Zhou, "An efficient buffer insertion algorithm for large networks based on lagrangian relaxation," in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*. IEEE, 1999, pp. 210–215.

[15] I.-M. Liu, A. Aziz, and D. Wong, "Meeting delay constraints in dsm by minimal repeater insertion," in *Proceedings of the conference on Design, automation and test in Europe*, 2000, pp. 436–440.

[16] R. Chen and H. Zhou, "Efficient algorithms for buffer insertion in general circuits based on network flow," in *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.* IEEE, 2005, pp. 322–326.

[17] C.-P. Lu, I. H.-R. Jiang, and C.-H. Hsu, "GasStation: Power and area efficient buffering for multiple power domain design," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 861–866.

[18] G. L. Zhang, B. Li, and U. Schlichtmann, "Sampling-based buffer insertion for post-silicon yield improvement under process variability," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1457–1460.

[19] P.-C. Wu, M. D. Wong, I. Nedelchev, S. Bhardwaj, and V. Parkhe, "On timing closure: Buffer insertion for hold-violation removal," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–6.

[20] I. Han, D. Hyun, and Y. Shin, "Buffer insertion to remove hold violations at multiple process corners," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 232–237.

[21] W. H. Choi and X. Liu, "Case study: runtime reduction of a buffer insertion algorithm using GPU parallel programming," in *23rd IEEE International SOC Conference*. IEEE, 2010, pp. 121–126.

[22] R. Liang, S. Nath, A. Rajaram, J. Hu, and H. Ren, "Bufformer: A generative ml framework for scalable buffering," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023, pp. 264–270.

[23] J. Cong and Z. Pan, "Interconnect performance estimation models for design planning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 6, pp. 739–752, 2001.

[24] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70–83, 2007.

[25] Z. Chai, Y. Zhao, W. Liu, Y. Lin, R. Wang, and R. Huang, "Circuitnet: An open-source dataset for machine learning in vlsi cad applications with improved domain-specific evaluation metric and learning strategies," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 5034–5047, 2023.

[26] X. Jiang, Y. Zhao, Y. Lin, R. Wang, R. Huang *et al.*, "Circuitnet 2.0: An advanced dataset for promoting machine learning innovations in realistic chip design environment," in *The Twelfth International Conference on Learning Representations*, 2023.

[27] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[28] The XiangShan Project, "High-performance risc-v processor," 2025. [Online]. Available: https://github.com/OpenXiangShan/XiangShan

[29] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–4.

[30] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[31] J. Hu, A. B. Kahng, S. Kang, M.-C. Kim, and I. L. Markov, "Sensitivity-guided metaheuristics for accurate discrete gate sizing," in *Proceedings of the International Conference on Computer-Aided Design*, 2012, pp. 233–239.