SimPart: A Simple Yet Effective Replication-Aided Partitioning Algorithm for Logic Simulation on GPU

Yi-Hua Chung[1], Shui Jiang[2], Wan-Luan Lee[1], Yanqing Zhang[3], Haoxing Ren[3], Tsung-Yi Ho[2], and Tsung-Wei Huang[1(✉)]

[1] University of Wisconsin-Madison, Madison, WI, USA
{yihua.chung,wanluan.lee,tsung-wei.huang}@wisc.edu
[2] The Chinese University of Hong Kong, Hong Kong, China
{sjiang22,tyho}@cse.cuhk.edu.hk
[3] Nvidia Corporation, Santa Clara, CA, USA
{yanqingz,haoxingr}@nvidia.com

Abstract. Replication-aided partitioning (RAP) has recently been introduced to facilitate the design of parallel logic simulation algorithms. By replicating overlapped work, RAP can significantly reduce the cost of inter-thread synchronization. However, the state-of-the-art RAP algorithm, RepCut, relies on time-consuming hypergraph construction and partitioning, where minimizing cut size corresponds to reducing replication. To overcome this runtime challenge, we introduce *SimPart*, a simple yet highly effective and efficient GPU-parallel replication-aided partitioner. SimPart tackles the partitioning problem directly without solving another proxy problem and proposes a hybrid strategy that can maximally utilize GPU threads for simulation atop our partitions. Compared to RepCut, SimPart achieves an average speedup of 23× in partitioning and 1.58× in GPU-parallel simulation, while increasing the original graph size by only 0.3%.

AQ1

Keywords: RTL simulation · Graph partitioning · Task graph parallelism

1 Introduction

Graph partitioning is an integral component in the design of parallel logic simulation algorithms [1,9,13,18]. Traditional partitioning algorithms focus on partitioning a circuit graph into a top-down *task dependency graph* (TDG) where each task represents a disjoint partition of work and each edge represents a dependency between two tasks. For instance, in Fig. 1(a), the input graph is partitioned to a TDG of three tasks and two dependencies, where each task encapsulates a disjoint set of work from the original graph (e.g., nodes 1, 2, and 4 in task 1). By delegating TDG scheduling to a task graph runtime, such as [6], we can efficiently parallelize logic simulation algorithms without the need

Y.-H. Chung and S. Jiang—Contributed equally to this work.

W. Nagel et al. (Eds.): Euro-Par 2025, LNCS 15902, pp. 1–14, 2025.
https://doi.org/10.1007/978-3-031-99872-0_14

to manage complex scheduling details. As a result, this type of *disjoint set-based partitioning* (DSP) has been widely used in existing logic simulators, such as Verilator [13], RTLFlow [9], TaroRTL [11], and ESSENT [1].

Although DSP can offer promising speedup through TDG parallelism, it incurs non-negligible scheduling overhead related to task synchronization and load balancing [6]. This overhead is further exacerbated when leveraging GPUs to accelerate logic simulation [18]. Specifically, excessive task dependencies can significantly impact both device-level and streaming multiprocessor (SM)-level scheduling performance, primarily due to load balancing and inter-block synchronization [12]. To address this issue, *replication-aided partitioning* (RAP) has been recently introduced to eliminate task dependencies via *replicating overlaps*. Take Fig. 1(b) for example, instead of partitioning the circuit graph to three non-overlapped tasks, RAP divides the circuit into two totally independent tasks with node 5 replicated in both task 1 and task 2. While replication introduces additional work per task, modern GPUs offer thousands of threads that applications can leverage to exchange replication for performance gains. For instance, by adding one more thread to run node 5, we can replace expensive inter-block synchronization (microseconds) with more efficient intra-block synchronization (tens to hundreds of cycles [9,17]).

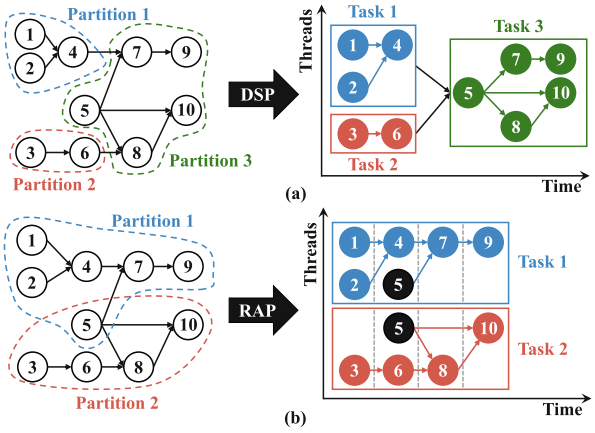

Fig. 1. Two partitioning approaches in parallel logic simulation: (a) disjoint set-based partitioning (b) replication-aided partitioning.

While there is an increasing adoption of RAP by modern logic simulators [2,14,16,18], the process of RAP is time-consuming. As originally introduced in RepCut [15], this process involves first constructing a *proxy hypergraph* from the original graph and then applying a *hypergraph partitioner* to solve this proxy problem, where minimum cut translates to minimum replication under a balance constraint. However, constructing the proxy hypergraph is not cheap since it requires traversing the graph multiple times to identify overlapped cones for hyperedges. For large hypergraphs, even state-of-the-art parallel hypergraph

partitioners [4,7] can take several minutes to converge to a reasonable cut size. As a result, this runtime cost makes RepCut challenging to use in a dynamic environment, such as simulation code optimization [9] and hardware fuzzing [10], where the partitioner is iteratively applied to a simulation graph that changes dynamically.

To tackle this problem, we introduce *SimPart*, a GPU-parallel replication-aided partitioner to facilitate the design of parallel logic simulation algorithms. Unlike RepCut, which was originally designed for CPU-parallel logic simulation algorithms, SimPart is specifically designed for GPU-parallel simulation. We summarize our technical contributions below:

- We introduce a simple yet highly effective and efficient algorithm that directly tackles the partitioning problem without solving another time-consuming proxy problem.
- We leverage the property of circuit graphs to propose a DSP-RAP hybrid strategy that effectively constrains the replication region while balancing the parallel efficiency between DSP and RAP.
- We leverage conditional CUDA Graph to design a GPU-parallel simulation framework atop our partition, which can reduce kernel call and control-flow overheads.

We evaluate the performance of SimPart on a set of RTL simulation graphs generated by RTLFlow [9], an open-source RTL simulator we developed with Nvidia Research. Compared to RepCut, SimPart achieves an average of 23× speedup in partitioning and 1.58× speedup in GPU-parallel simulation, while increasing the original graph size by only 0.3%.

2 Background

Logic simulation is essential for verifying design functionality. A circuit is modeled as a directed graph, where nodes represent logic elements (e.g., RTL instructions, gates) and edges capture data dependencies. Simulation evaluates this graph by propagating inputs through logic elements to produce outputs, often iterated over multiple testbenches and configurations for coverage. Large designs can yield graphs with millions of nodes and edges, leading to long simulation times [18].

To mitigate this runtime challenge, existing simulators have introduced various partitioning algorithms to distribute work across CPU or GPU threads. As shown in Table 1, existing partitioning algorithms can be categorized to DSP and RAP, where the former disallows overlapping nodes among tasks while the latter allows them. For instance, Verilator [13] uses a DSP algorithm that iteratively clusters adjacent nodes into macro tasks and formulates dependent macro tasks into a TDG. Parallel execution of a TDG is achieved with a scheduler that manages inter-task dependencies and load balancing across threads. Most DSP-based simulators follow this paradigm but adopt different clustering heuristics [1,9,11,19].

Table 1. Comparison between SimPart and existing parallel logic simulators.

	Verilator [13]	RTLFlow [9]	RepCut [15]	GL0AM [18]	SimPart **(Ours)**
Partitioning algorithm	DSP	DSP	RAP	RAP	**DSP+RAP**
Simulation platform	CPU	GPU	CPU	GPU	**GPU**

On the other hand, RAP focuses on replicating a small portion of the graph to break task dependencies for reduced inter-thread synchronization. Although initially introduced by RepCut [15] for implementing a CPU-parallel RTL simulator, RAP has proven especially useful for designing GPU-parallel simulators [18]. Specifically, by assigning a few additional GPU threads to run replicated logic elements, we can eliminate expensive inter-block synchronization and enable uninterrupted thread execution. Despite these advantages, RepCut requires solving a *proxy hypergraph partitioning* problem, where a hyperedge cut implies its corresponding nodes need to be replicated across different partitions. Unfortunately, hypergraph partitioning is NP-hard. Even with state-of-the-art parallel hypergraph partitioners [4,7], finding a partition with a decent cut size can still take a long time to converge, especially for large graphs (e.g., 3–5 min for a graph of 23M nodes [4]).

3 SimPart

Figure 2 gives an overview of SimPart, which consists of three stages, *majority-based grouping*, *DSP-RAP hybrid partitioning*, and *GPU-parallel simulation*. First, SimPart introduces a majority-based grouping algorithm that assigns group IDs to nodes based on their major connectivity, which is efficiently implemented through a one-pass BFS traversal on the GPU. Second, SimPart employs a DSP-RAP hybrid partitioning strategy with multiple backward traversals to identify partitions that balance synchronization cost and GPU thread utilization. Finally, we present the GPU-parallel simulation framework atop our partition.

3.1 Majority-Based Grouping

Instead of solving another time-consuming proxy hypergraph partitioning problem like RepCut [15], we propose a simple yet effective algorithm called *majority-based grouping*. The goal of majority-based grouping is to determine a partition group for every primary output (PO) that will minimize the number of replications. Specifically, we assign each node in the circuit graph to a group based on a majority count that reflects its predecessors' connectivity. This grouping method can achieve a similar effect to the hypergraph formulation in RepCut but with significantly reduced computation time. Our majority-based grouping algorithm consists of two main steps, *initial grouping* and *group propagation*.

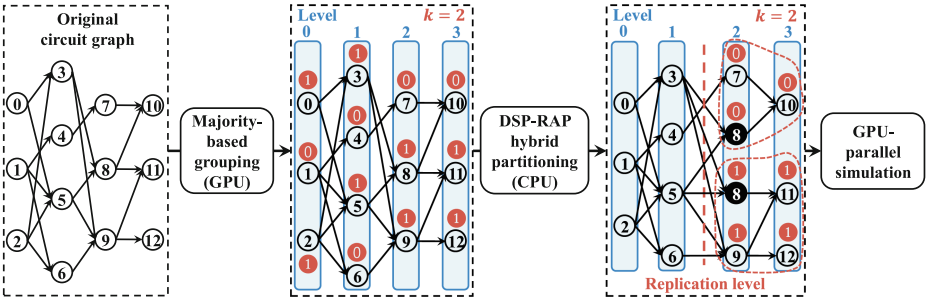

Fig. 2. Overview of SimPart under two partitions ($k = 2$ and $\gamma = 3$). In majority-based grouping, we leverage GPU to quickly assign a partition group to each node through a level-by-level parallel traversal. In DSP-RAP hybrid partitioning, we perform a backward traversal from the POs to the threshold replication level, where parallel cones are identified by replicating their overlaps (node 8 in both cones). Nodes before the replication level are partitioned into disjoint, non-overlapping groups level by level ({0, 1, 2} and {3, 4, 5, 6}).

The goal of initial grouping is to assign each primary input (PI) an initial group ID for later propagation. In this step, each PI is assigned a group ID based on the majority count of its successors, with the count calculated under modulo k for k partitions. The intuition is to assign group IDs based on primary node connections and encourage the later propagation to place connected nodes in the same group. For example, node 1 in Fig. 2 is assigned a group ID of 0, since the modulo outputs of its three successors are 0, 1, and 0 for nodes 4, 5, and 6 under $k = 2$. After assigning group IDs to the PIs, the goal of group propagation is to propagate these IDs from PIs to POs, keeping connected nodes in the same group as much as possible. To this end, we perform a BFS to traverse the circuit graph level by level. When a node is traversed (once its predecessors' dependencies are resolved), we will assign it a group ID based on the majority count of its predecessors' group IDs. For example, node 5 in Fig. 2 is assigned a group ID of 1, since the group IDs of its three predecessors are 1, 0, and 1 for nodes 0, 1, and 2, respectively.

A key advantage of our majority-based grouping algorithm is its high data parallelism during group propagation, as nodes at the same level can operate independently of each other. To maximally leverage this parallelism, we introduce a GPU-parallel majority-based grouping algorithm. We present a GPU-parallel majority-base grouping algorithm, as shown in Algorithm 1. Algorithm 1 consists of two steps, initial grouping (lines 1–13) and group propagation (lines 14–30). We use the compressed sparse row (CSR) data structure to store the input graph, as commonly done in many GPU-accelerated graph algorithms [8,9].

In step one (lines 1–13), we assign one GPU thread to process each PI. Each thread determines the group ID for its assigned PI (indexed by *PIIdx*) based on the majority count of its successors, stored in the PI's *outputs* array (line 3). For each successor, the thread calculates the modulo of its index under k

Algorithm 1. Majority-based grouping

```
1: parallel for each thread {  /* Step 1: Initial grouping */
2:     /* Assign group ID to PIs */
3:     for each outNodeIdx ∈ outputs of PIIdx
4:         mod = outNodeIdx % k; accum[mod]++
5:     /* Assign a group ID based on the majority count */
6:     Group[PIIdx] = argmax(accum)
7:     /* Initial BFS queue Q */
8:     for each outNodeIdx ∈ outputs of PIIdx
9:         atomicSub(in_degree[outNodeIdx], 1)
10:        /* Enqueue nodes whose parent dependencies are resolved */
11:        if in_degree[outNodeIdx] == 0 then {enqueue outNodeIdx in Q}
12:    level[PIIdx] = 0; totalLevels = 0; accum.clear()
13: }
14: Grp_propagation_kernel {  /* Step 2: Group propagation */
15:    parallel for each thread {
16:        nodeIdx = dequeue Q; level[nodeIdx] = totalLevels
17:        /* Assign group IDs based on majority count */
18:        for each inputIdx ∈ inputs of nodeIdx
19:            groupIdOfInput = Group[inputIdx]; accum[groupIdOfInput]++
20:        /* Maintain BFS queue Q */
21:        for each outNodeIdx ∈ outputs of nodeIdx
22:            atomicSub(in_degree[outNodeIdx], 1)
23:            if in_degree[outNodeIdx] == 0 then {enqueue outNodeIdx in Q_tmp}
24:    }
25: }
26: while Q is not empty do {
27:    Q_tmp ← empty queue
28:    call Grp_propagation_kernel #blocks =⌈Q.size()/1024⌉, #threads =1024
29:    Q ← Q_tmp; totalLevels++; accum.clear()
30: }
```

and increments the corresponding count in an accumulation array, *accum* (lines 3–4). This array stores the counts of modulo results for each group, allowing the thread to assign the PI to the group with the highest count. Each PI's assigned group ID is stored in the *Group* array (line 6). After this, each thread initializes the queue Q for group propagation (lines 8–11) by inserting resolved successors (via updating their *in_degree* count with `atomicSub`). Finally, each thread sets its PI node level to 0 and initializes *totalLevels* to track circuit levels (line 12).

In step two, the function runs a while loop until all nodes receive a group ID (lines 14–30). In each iteration, it launches *Grp_propagation_kernel* to process all nodes in Q using $\lceil Q.size()/1024 \rceil$ blocks each of 1024 threads (the thread count can be configured by applications). Each thread dequeues a node, assigns it a level index in *totalLevels* (line 16), and a group ID based on its predecessors' group majority. It then updates the BFS queue by decrementing each output node's *in_degree* using `atomicSub` and enqueues nodes with zero *in_degree* (lines

21–23). After processing, the main loop updates Q with Q_tmp, increments *totalLevels*, and repeats until all nodes are grouped (lines 26–30).

3.2 DSP-RAP Hybrid Partitioning

The goal of RAP is to perform multiple backward traversals to construct a fanin cone for each group ID starting from the POs, while replicating nodes with different group IDs. However, as this traversal goes deeper towards level 0, the risk of over-replication significantly increases. For example, applying RAP to the middle graph in Fig. 2 will yield two partitions with many replications, as shown in Fig. 3, where all nodes in level 0 are replicated. While RAP can create two fully independent partitions that can be processed by two separate GPU blocks in parallel, over-replicated nodes may oversubscribe threads during simulation. Additionally, circuit graphs typically exhibit a long-tailed distribution after levelization, with more parallel nodes at earlier levels than the later levels, as shown in Fig. 4.

To understand the impact of this property, we assume that pure DSP yields a level-by-level GPU-parallel simulation and pure RAP yields k independent instances of level-by-level GPU-parallel simulation (see Fig. 1(b)). If we roughly characterize the parallel efficiency to the ratio of GPU thread utilization to synchronization overhead, then pure RAP yields low parallel efficiency at earlier levels due to fixed thread count processing potentially many replicated nodes with frequent intra-block synchronizations. As we progress with fewer nodes, the number of intra-block synchronizations decreases, thus increasing its parallel efficiency. On the other hand, pure DSP can yield high parallel efficiency at earlier levels due to unrestricted thread counts and less frequent intra-block synchronization. However, as we progress, the cost of inter-block synchronization dominates the performance and reduces its parallel efficiency.

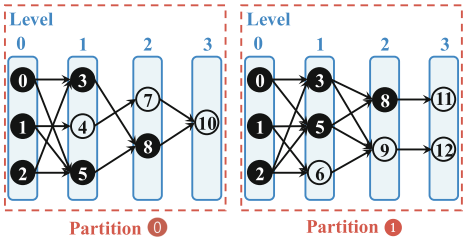

Fig. 3. Unconstrained RAP can result in over-replication (marked in black), which oversubscribes GPU threads during simulation.

To balance parallel efficiency between synchronization cost and GPU thread utilization, we propose a DSP-RAP hybrid strategy that constrains the replication region to a threshold level during the backward traversal. We define γ as the upper bound on the number of nodes at the level where backward traversal stops. We refer to this level as *replication level*. For example, in Fig. 2, with $\gamma = 3$, the replication level is level 1, as it is the first backward level with more than three parallel nodes. A larger γ favors RAP-based simulation with more intra-block synchronizations, which reduces parallel efficiency and increases simulation time. Conversely, a smaller γ leans towards DSP-based simulation, resulting in more costly inter-block synchronization (microseconds).

By default, SimPart sets γ to the GPU block size since it guarantees each replicated partition to be processed within a thread block with minimal synchronization overhead.

Algorithm 2 presents our DSP-RAP hybrid partitioning strategy. First, we determine the replication level l_γ as the first backward level containing more than γ nodes (line 1). Next, we form RAP partitions by constructing a fanin cone for each group ID starting from POs until l_γ (lines 3–4) and replicating nodes with different group IDs from that of POs (lines 5–6). Finally, we assign each of the remaining levels to a DSP partition (lines 7–12).

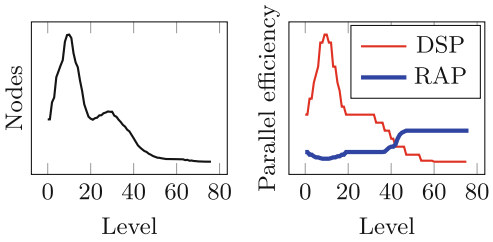

Fig. 4. Node distribution across levels for the real circuit edit_dist [9], and parallel efficiency for two partitioning strategies.

Algorithm 2. DSP-RAP hybrid partitioning

```
1: l_γ = argmax_l{numNodes[l] > γ}
2: for each v ∈ {v | level[v] == totalLevels−1} {  /* RAP */
3:     rap_part[Group[v]] = {x | x ∈ fanin_cone(v) and level[x] > l_γ}
4:     for each y ∈ rap_part[Group[v]]
5:         if Group[y] ≠ Group[v] then {replicate y}
6: }
7: for each l ∈ {l_γ, ···, 1, 0} { /* DSP */
8:     new_part = {}
9:     for each v ∈ {v | level[v] == l}
10:        new_part.insert(node)
11:    dsp_part = dsp_part ∪ new_part
12: }
```

3.3 Conditional CUDA Graph-Based Simulation Framework

A key advantage of SimPart is its simple linear structure for kernel scheduling, starting with DSP-based simulation followed by RAP-based simulation. However, iterative kernel launches can incur non-negligible CPU overhead, especially for large graphs with many levels. Additionally, logic simulators often iterate through multiple inputs, resulting in frequent CPU-GPU interactions for control-flow decisions. Figure 5(a) illustrates these two challenges. To overcome these challenges, we leverage the latest *conditional CUDA Graph* [3] to design a simulation framework that encapsulates both simulation kernels and dynamic control flow within a single GPU-resident graph entity. As shown in Fig. 5(b), we create a conditional CUDA Graph node to represent the while-loop condition, which iteratively invokes our simulation kernels until no more inputs remain.

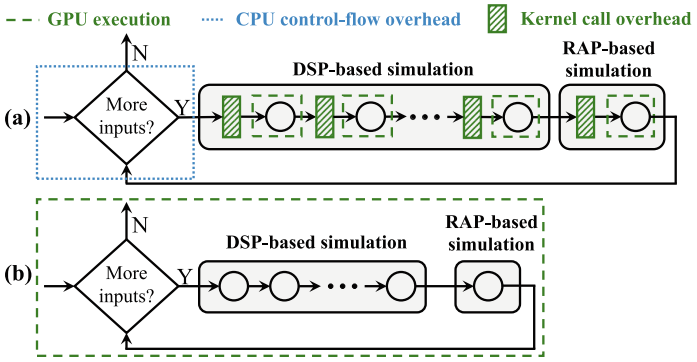

Fig. 5. GPU-parallel simulation framework atop our partition using (a) stream-based and (b) conditional CUDA Graph-based executions.

This strategy not only reduces iterative kernel call overhead but also decreases CPU involvement in decision-making.

The idea of our GPU-parallel simulation kernels is level-by-level parallel propagation, where multiple GPU threads process parallel nodes within each level simultaneously. While this idea is common in GPU-accelerated CAD algorithms [5,18], we outline our kernels in Algorithm 3 and Algorithm 4 for completeness.

Algorithm 3. DSP-based simulation

```
1: DSP_sim_kernel {
2:     parallel for each thread tid
3:         res[tid] = run_logic_operation_of(nodes[tid])
4: }
5: DSP_based_sim {
6:     for each l ∈ {0, 1, ..., totalLevels-1}
7:         call DSP_sim_kernel #blocks = ⌈#nodesAtLvl_l/1024⌉; #threads = 1024
8:         sync_kernel /* inter-block sync */
9: }
```

The key difference between Algorithm 3 and Algorithm 4 lies in the scope of level-by-level propagation: DSP operates across partitions with inter-block synchronization, while RAP processes individual partitions within a single kernel launch. Specifically, in RAP-based simulation (Algorithm 4), we assign a GPU block to simulate each partition using just one kernel launch (line 13). Within each block, additional passes are required if the number of parallel nodes at a level exceeds the block size (lines 5–newinlinkFigdalg::RAPsim::l99). Threads are synchronized at the end of each iteration (i.e., intra-block synchronization).

Algorithm 4. RAP-based simulation

```
1: RAP_sim_kernel {
2:     parallel for each block {
3:         for each l ∈ {0, 1, ...,totalLevels-1}
4:             passes = ⌈#nodesAtLevel_l_in_Partition/1024⌉
5:             for each pass ∈ {0, 1, ..., passes-1}
6:                 parallel for each thread tid
7:                     res[tid] = run_logic_operation_of(nodes[tid]); tid += 1024
8:                 sync_threads /* intra-block sync */
9:             sync_threads /* intra-block sync */
10:    }
11: }
12: RAP_based_sim {
13:     call RAP_sim_kernel #blocks = k, #threads = 1024
14: }
```

4 Experimental Results

We implemented SimPart in CUDA/C++ and compiled it with nvcc v12.3 using `-O3` and `-std=c++20`. Experiments were conducted on a 64-bit Linux machine with 32 Intel Core i5-13500 cores (4.8 GHz) and an Nvidia RTX A4000 GPU. We evaluated SimPart on logic graphs generated by the RTLFlow simulator [9]; graph statistics are shown in Table 2, with the "_eval" suffix indicating larger variants with added logic elements. Without loss of generality, we simulate a batch of 12 simulation inputs. Since the inputs are independent, increasing the number of inputs does not affect the results but further amplifies the performance gap between SimPart and the baseline.

We consider the state-of-the-art RepCut [15] as our baseline. We use CPU to construct the proxy hypergraph and use Mt-KaHyPar [4] with 16 threads to derive k partitions, as it delivers the best runtime performance on our machine. For comparison, we simulate on both CPU and GPU using OpenMP and our RAP-based kernel, respectively. On CPU, one OpenMP thread runs one partition, following RepCut's original setting. By default, we set the number of partitions (k) to 16, yielding good performance for both RepCut and SimPart. For SimPart, γ is set to 1024 to match the GPU kernel block size, balancing DSP and RAP performance on our machine. Both k and γ are tunable for platform-specific optimization. All results are averaged over 10 runs.

4.1 Overall Performance Comparison

Table 2 compares the overall performance between RepCut and SimPart. We can observe that RepCut spends a large amount of time on solving the proxy hypergraph problem, including building the hypergraph and partitioning the hypergraph. While Mt-KaHyPar reduces partitioning time by 1.9–3.6× by using 16 threads, it still has an average 23× gap compared to SimPart due to our GPU-powered partitioning algorithm. In terms of partition sizes, SimPart outperforms

Table 2. Overall performance comparison between RepCut [15] and SimPart with $k = 16$. Partition quality is evaluated based on the resulting simulation time on CPU (RepCut only) and GPU. The average ratios of replication, partitioning time, and simulation time are measured relative to the original graph size ($|V|$ and $|E|$), T_{Part} in SimPart, and T^{G}_{Sim} in SimPart.

Benchmark			RepCut [15]								SimPart			
	$\|V\|$	$\|E\|$	T^{C1}_{Build}	T^{C1}_{Part}	T^{C16}_{Part}	T^{C1}_{Sim}	T^{C16}_{Sim}	T^{G}_{Sim}	$R_{\|V\|}$	$R_{\|E\|}$	T_{Part}	T^{G}_{Sim}	$R_{\|V\|}$	$R_{\|E\|}$
edit_dist	164K	164K	590.3	1492	438.6	439.5	281.8	5.1	1.01765	1.01765	72.3	3.2	1.00528	1.00528
matrix_mult	176K	174K	737.4	1024	384.6	895.6	450.2	6.2	1.00235	1.00235	86.6	3.4	1.03148	1.03148
b19	255K	255K	2058	4377	1203	169.1	40.3	7.5	1.00911	1.00911	77.3	4.0	1.00210	1.00210
leon2	1.6M	1.6M	10054	12478	5501	6846	900.2	35.5	1.00166	1.00166	213.3	21.2	1.00000	1.00000
leon3mp	1.2M	1.2M	6893	12542	4513	4648	528.5	27.3	1.00487	1.00487	146.1	16.1	1.00000	1.00000
netcard	1.5M	1.5M	9120	24578	7209	7589	1184	30.7	1.02060	1.02060	155.9	18.4	1.00000	1.00000
edit_dist_eval	1.3M	1.3M	5193	7432	3148	54791	10555	25.6	1.00209	1.00209	160.5	18.1	1.00000	1.00000
matrix_mult_eval	1.4M	1.4M	6531	5545	2886	76191	13465	29.0	1.00004	1.00004	779.2	17.3	1.00002	1.00002
b19_eval	2.0M	2.0M	15976	37299	10872	4157	811.6	39.4	1.00147	1.00147	925.2	24.3	1.00000	1.00000
leon2_eval	12.9M	12.9M	91500	64697	44711	165011	29598	235.5	1.00021	1.00021	2483	169.5	1.00000	1.00000
leon3_eval	20.0M	20.0M	106012	128188	57139	219415	38024	327.5	1.00004	1.00004	2971	259.0	1.00000	1.00000
netcard_eval	24.0M	24.0M	143731	766218	218582	1530950	283890	371.8	1.00000	1.00000	2866	292.4	1.00000	1.00000
Avg. ratio	**1.000**	**1.000**	-	**68.45**	**23.13**	**1344.76**	**255.04**	**1.58**	**1.005**	**1.005**	**1.000**	**1.000**	**1.003**	**1.003**

$|V|$: Number of nodes
T_{Build}: Hypergraph construction time
T^{Cn}_{Sim}: Simulation time on CPU, with n threads
T_{Part}: Partitioning time
$|E|$: Number of edges
T^{Cn}_{Part}: Partitioning time on CPU, with n threads
T^{G}_{Sim}: Simulation time on GPU
$R_{|V|}$, $R_{|E|}$: Ratio of replicated nodes and edges

RepCut in nearly all graphs except matrix_mult, where hypergraph partitioning achieves a near-optimal solution. On average, SimPart reduces replication by 40% compared to RepCut (0.3% vs. 0.5%). This improvement is due to SimPart's replication constraint, which reduces the risk of over-replication in most cases. Further, the NP-hard nature of hypergraph partitioning makes it challenging for Mt-KaHyPar to find the optimal cut size that minimizes replication for RepCut.

Next, we evaluate the quality of partitions based on the resulting simulation time. Compared to CPU-based simulation on RepCut's partitions, our GPU-based simulation can achieve an average speedup of 1344× over 1 CPU thread (column of T^{C1}_{Sim}) and 255× over 16 CPU threads (column of T^{C16}_{Sim}). When using our GPU simulation kernel to RepCut's partitions, we can still achieve an average speedup of 1.58× (columns of T^{G}_{Sim}). Again, we attribute this speedup to our DSP-RAP hybrid partitioning strategy, which strikes a balanced trade-off between synchronization costs and GPU thread parallelism.

4.2 Performance Under Different Numbers of Partitions

Figure 6 compares the performance between RepCut and SimPart for different partition counts (k), which directly impact the parallelism available for simulation. We can observe that as k increases, RepCut's runtime to solve the proxy hypergraph problem also increases. Take leon2 for example, when k increases from 16 to 128, RepCut's runtime increases from 14.3 to 20.2 s. This runtime cost makes RepCut challenging to use in a dynamic environment, such as simulation-driven hardware fuzzing [10], where the partitioner is iteratively applied to a

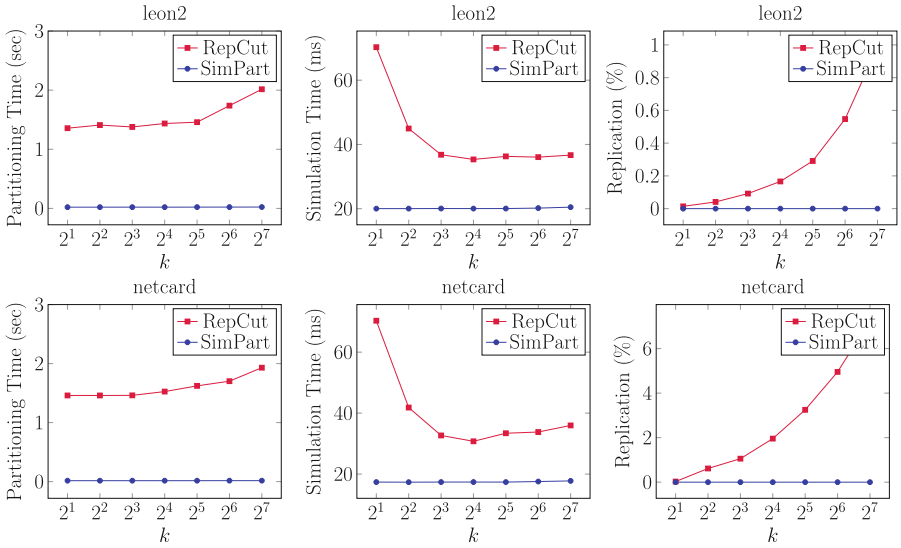

Fig. 6. Comparisons of partitioning time (left), partition quality (middle), and partitioned graph size (right) between RepCut and SimPart at different k.

simulation graph that changes dynamically. On the other hand, SimPart has very low partitioning times regardless of k (all $\leq$ 230 ms). We attribute this efficiency to our GPU-powered partitioning algorithm.

In terms of the partition quality, measured at the resulting simulation time on GPU, SimPart is always faster than RepCut (middle plot). As k increases, RepCut generates more parallel partitions to accelerate the simulation, yet eventually reaching a saturation point at 16 partitions.

Beyond 16 partitions, we begin to see diminishing performance returns in RepCut, primarily due to over-replication, which can be revealed in the bottom plot. On the contrary, the simulation time on our partitions consistently outperforms RepCut across all values of k. We attribute this speedup to SimPart's DSP-RAP hybrid partitioning strategy, which avoids over-replication while maximizing the parallel efficiency across all levels of the circuit graph.

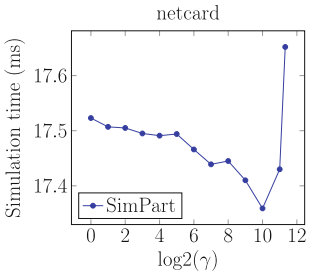

Fig. 7. Partition quality vs. γ.

4.3 Analysis of Replication Threshold

We study the impact of the replication threshold, γ, on the partition quality. As shown in Fig. 7, the curve forms a U-shape as γ increases. As discussed in Sect. 3.2, a larger γ favors RAP-based simulation with more intra-block synchro-

nization, which reduces parallel efficiency and thus increases simulation time. Conversely, a smaller γ leans towards DSP-based simulation, resulting in more costly inter-block synchronizations. At the two extremes of $\gamma = 0$ and $\gamma = \infty$, the simulation reduces to pure level-by-level DSP and pure RAP, neither of which achieves optimal performance. With $\gamma = 1024$, SimPart can balance the parallel efficiency between DSP and RSP to achieve optimal simulation performance.

4.4 Ablation Analysis of Conditional CUDA Graph

To highlight the advantage of our conditional CUDA Graph-based simulation framework, we analyze the simulation time with and without conditional CUDA Graph. As shown in Fig. 8, SimPart with conditional CUDA Graph outperforms RepCut across all circuits. Without CUDA Graph, SimPart still achieves better performance than RepCut in all but two small circuits (edit_dist and b19), which highlights the effectiveness of our partitioning algorithm. For SimPart itself, conditional CUDA Graph achieves an average speedup of 27%, with the highest speedup of 2.23× observed in b19. We attribute this improvement to the integration of simulation kernels and control-flow decisions into a single GPU-resident graph, which reduces kernel call overhead and decreases CPU involvement in decision-making.

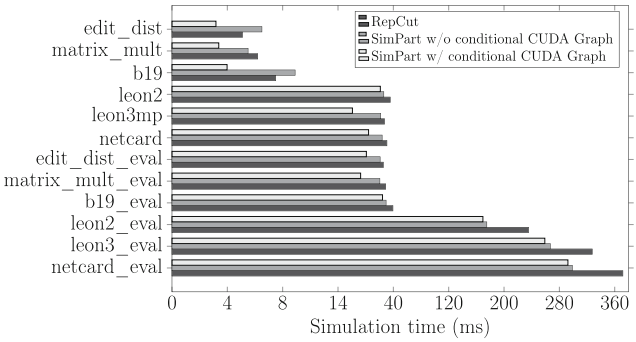

Fig. 8. Simulation time of RepCut vs. SimPart with and without conditional CUDA Graph.

5 Conclusion

In this paper, we have introduced *SimPart*, a simple yet highly effective and efficient GPU-parallel replication-aided partitioner to facilitate the design of parallel RTL simulation algorithms. SimPart addresses the partitioning problem directly, without relying on a proxy problem, and proposes a hybrid strategy that makes the most of GPU parallelism when simulating the circuit on our partition. Compared to RepCut, SimPart achieves an average of 23× speedup in partitioning and 1.58× speedup in GPU-parallel simulation, while increasing the original graph size by only 0.3%.

Acknowledgment. This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141. This research was also conducted by ACCESS – AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the HKSAR Government.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Beamer, S., et al.: Efficiently exploiting low activity factors to accelerate RTL simulation. In: ACM/IEEE DAC, pp. 1–6 (2020)
2. Emami, S.: Highly Parallel RTL Simulation. Ph.D. thesis, EPFL (2024)
3. Gaiser, J., et al.: Dynamic control flow in cuda graphs with conditional nodes (2024)
4. Gottesbüren, L., et al.: Scalable shared-memory hypergraph partitioning. In: 23rd Workshop on ALENEX 2021, pp. 16–30. SIAM (2021)
5. Huang, T.W., et al.: OpenTimer v2: a new parallel incremental timing analysis engine. IEEE TCAD **40**, 776–789(2021)
6. Huang, T.W., et al.: Taskflow: a lightweight parallel and heterogeneous task graph computing system. IEEE TPDS **33**, 1303–1320 (2022)
7. LaSalle, D., et al.: Multi-threaded graph partitioning. In: IEEE IPDPS, pp. 225–236 (2013)
8. Lee, W.L., et al.: G-kway: multilevel GPU-accelerated k-way graph partitioner. In: ACM/IEEE DAC (2024)
9. Lin, D.L., et al.: From RTL to CUDA: a GPU acceleration flow for RTL simulation with batch stimulus. In: Proceedings of the 51st ICPP, pp. 1–12 (2022)
10. Lin, D.L., et al.: GenFuzz: GPU-accelerated hardware fuzzing using genetic algorithm with multiple inputs. In: ACM/IEEE DAC (2023)
11. Lin, D.L., et al.: TaroRTL: accelerating RTL Simulation using coroutine-based heterogeneous task graph scheduling. In: Euro-Par (2024)
12. Olmedo, I.S., et al.: Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In: IEEE RTAS, pp. 213–225 (2020)
13. Snyder, W.: Verilator 4.0: open simulation goes multi- threaded (2018)
14. Tong, J., et al.: BatchSim: parallel RTL simulation using inter-cycle batching and task graph parallelism. In: IEEE ISVLSI (2024)
15. Wang, H., et al.: RepCut: superlinear parallel RTL simulation with replication-aided partitioning. In: ACM ASPLOS, pp. 572–585 (2023)
16. Wang, H., et al.: Don't Repeat Yourself! ACM ASPLOS, Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation. In (2024)
17. Wong, H., et al.: Demystifying GPU microarchitecture through microbenchmarking. In: 2010 IEEE ISPASS, pp. 235–246. IEEE (2010)
18. Zhang, Y., et al.: GL0AM: GPU logic simulation using 0-delay and re-simulation acceleration method. In: IEEE/ACM ICCAD (2024)
19. Zhou, K., et al.: Khronos: fusing memory access for improved hardware RTL simulation. In: IEEE/ACM Micro, pp. 180–193 (2023)