A Differentiable Approach to Task Graph Partitioning: A Case Study in RTL Simulation

Aditya Das Sarma
University of Wisconsin-Madison
Madison, USA
adassarma@cs.wisc.edu

Wan-Luan Lee
University of Wisconsin-Madison
Madison, USA
wlee329@wisc.edu

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

Boyang Zhang
University of Wisconsin-Madison
Madison, USA
bzhang523@wisc.edu

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

Abstract

Graph partitioning is essential for many EDA applications that leverage task graph parallelism for faster execution. For instance, RTL simulators partition an input RTL design into dependent tasks and schedule them across threads. However, existing partitioners are largely limited to general-purpose heuristics that overlook real threading costs, resulting in suboptimal performance. Consequently, we introduce DiffPart, a differentiable task graph partitioning framework that automatically learns high-quality partitions under real operating conditions. Applied to RTL simulation, DiffPart improves state-of-the-art Verilator's partitioning quality, delivering up to 1.22–55.25× faster simulation runtime across diverse designs.

ACM Reference Format:
Aditya Das Sarma, Wan-Luan Lee, Shui Jiang, Boyang Zhang, and Tsung-Wei Huang. 2026. **A Differentiable Approach to Task Graph Partitioning: A Case Study in RTL Simulation**. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803900

1 Introduction

Graph partitioning is a fundamental technique widely adopted by many electronic design automation (EDA) applications to parallelize their algorithms using *task graph parallelism* (TGP) [7, 8, 31]. For example, in RTL simulation, existing algorithms partition an input design into a *task graph*, where a task encapsulates a set of RTL instructions and an edge represents a data dependency between two tasks [43]. By delegating the task graph to a *scheduler* like Taskflow [18], RTL simulators can benefit from automatic parallelization without worrying about low-level scheduling details, such as concurrency control and load balancing. Similar uses of TGP can be found in task-parallel static timing analysis [20], global routing [35], placement [34], and so on [5, 22].

However, existing task graph partitioners often rely on general-purpose heuristics, such as cut size [11, 24, 25] and path criticality [39], to guide the partitioning process. For instance, the partitioner in the state-of-the-art RTL simulator Verilator [43] uses a hardcoded cost for each instruction and greedily merges tasks that result in the least increase to the critical path cost, inspired from [39]. While this greedy approach is computationally efficient, it fails to account for practical execution costs like scheduling overhead of the resulting partition. As a consequence, this type of heuristic-driven partitioning often results in suboptimal simulation performance. To better understand this issue, we randomly sampled several task graph partitions of five real-world RTL designs and measured their simulation performance. As shown in Figure 1, the default partitioner of Verilator leaves 1.7–6.8× performance on the table, compared to the best-sampled partitions.

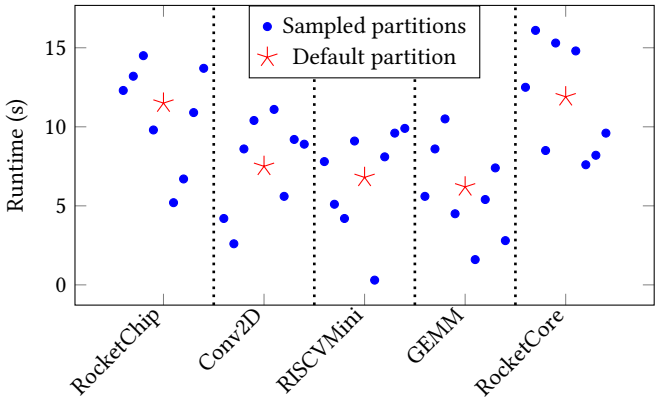

Figure 1: Task graph partitioning has a great impact on the simulation performance of Verilator. The default partition leaves 1.7–6.8× performance on the table.

To address this challenge, we propose DiffPart, a novel task-graph partitioning framework that capitalizes on recent advances in machine learning to learn from the intrinsic features of the input graph. Unlike existing partitioners that count on general-purpose heuristic, DiffPart designs a *differentiable optimization*-based algorithm to automatically learn how to derive a high-quality partition under real operating conditions. We summarize three key contributions as follows:

- We formulate graph partitioning into a continuous learning problem and solve it using differentiable optimization. At the core of our method is a Graph Neural Network (GNN) [41] that learns to map a graph's semantic features to high-quality partitions. This learning-based formulation generalizes effectively across diverse task graph structures.
- We introduce an efficient algorithm to prevent cycles during our graph partitioning process. Our algorithm takes the learned distribution from the GNN and samples node-to-partition assignments in a topological order, which inherently ensures acyclicity.
- We use backpropagation to guide the partitioning directly towards optimizing the real execution cost, instead of searching for a proxy combinatorial objective. This allows DiffPart to be agnostic to the

DAC '26, July 26–29, 2026, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07…$15.00
https://doi.org/10.1145/3770743.3803900

task graph execution backend, whether it is static or dynamic, and thus it is flexible for different task graph-based applications.

We evaluate the effectiveness of DiffPart using RTL simulation as a case study. RTL simulation is a fundamental component in the design flow of SoC as it verifies the functional correctness at the RTL level. Besides, existing RTL simulators heavily rely on TGP to enhance the simulation performance for large designs [33]. Compared to the default partitioner of the industry-grade RTL simulator, Verilator [43], DiffPart shows 1.22-55.25× faster simulation performance across a wide range of hardware designs. We plan to release DiffPart as open-source software to benefit the EDA community.

2 Problem Formulation

DiffPart is inspired by our research on accelerating the RTL simulation performance on a multithreaded platform, where TGP plays an important role in parallelizing the simulation algorithms. We consider Verilator [43], a widely used open-source RTL simulator, as our case study. Although we focus on RTL simulation, we believe that the ideas behind DiffPart naturally extend to other TGP-powered EDA applications [20, 34, 35], as they all rely on effectively partitioning dependent tasks for parallel execution.

As shown in Figure 2, Verilator compiles an RTL design into a directed acyclic graph (DAG) $G = (V, E)$, aka an *RTL graph*, where each vertex $v \in V$ represents a set of RTL instructions and each edge $e = (u, v) \in E$ captures a data dependency between RTL instructions. Each vertex has an associated weight c_v representing its estimated computational cost. Leveraging TGP, Verilator partitions G into dependent *macro tasks* (mTasks) using a hard-coded heuristic that estimates the execution cost of each RTL instruction, and assigns these mTasks to a custom static scheduler (i.e., no dynamic load balancing scheme like [18]) for parallel simulation. Formally, this partitioning divides V into k disjoint subsets $P = \{p_1, p_2, \ldots, p_k\}$, such that $\bigcup_i p_i = V$ and $p_i \cap p_j = \emptyset$ for all $i \neq j$, where each p_i denotes an mTask mapped to a thread. To maximize the parallel efficiency, we need to optimize P based on two competing objectives:

- **Workload Balancing:** To maximize parallelism, computation should be evenly distributed across threads. This is modeled by minimizing the cost of the heaviest macro task, defined as $|p_i| = \sum_{v \in p_i} c_v$, with the objective $\min \max_i |p_i|$.
- **Threading Overhead:** To reduce synchronization costs, the number of cross-partition edges should be minimized. This is equivalent to minimizing the *cut size*, i.e., edges $(u, v) \in E$ where $u \in p_i$, $v \in p_j$, and $i \neq j$.

Meanwhile, these objectives are subject to two key structural constraints:

- **Topological Constraint:** The inter-partition dependency graph G_P, where each node represents a macro task p_i, must preserve all data dependencies from the original graph. That is, if there is an edge from a vertex in p_i to a vertex in p_j in G, then G_P must include a directed edge from p_i to p_j.
- **Acyclicity Constraint:** G_P must remain acyclic to avoid erroneous task graph execution.

3 DiffPart

Figure 3 presents an overview of DiffPart that comprises of three main stages: *Preprocessing*, *Core*, and *Cycle-free Sampling*, followed by the downstream simulation workflow, from scheduling to execution on hardware.

- **Preprocessing:** The goal is to generate structure-aware features for each node in the RTL graph. We employ a GNN to process the entire graph, enabling each node to learn an embedding that encodes its local connectivity patterns and neighborhood structure. The resulting embeddings are used as input for the core partitioning stage.

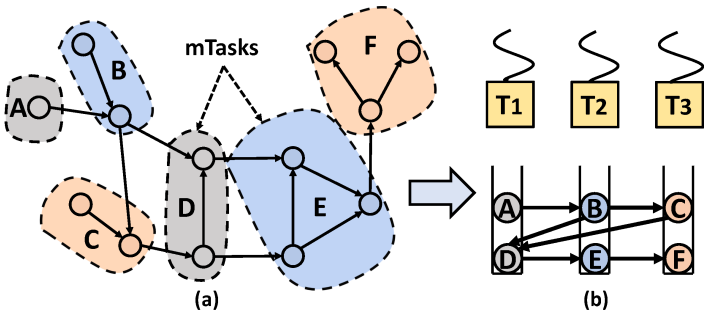

Figure 2: Illustration of Verilator's task graph partitioning for parallel simulation. (a) An input RTL graph is partitioned to a set of macro tasks and dependencies. (b) Partitioned macro tasks are then executed by different threads through a custom static scheduler that assigns each macro task to a thread T_i.

Algorithm 1: Main Backpropagation Loop

Input: Graph G
Output: Best partition assignments A_{best}
Initialize: GNN model M, optimizer Opt
$A_{\text{best}} \leftarrow \emptyset$; $cost_{\text{best}} \leftarrow \infty$; num_steps $\leftarrow$ 400; sample_steps $\leftarrow$ 10
for $step \leftarrow 1$ **to** *num_steps* **do**
  $\tau \leftarrow$ GetTemperature($step$)
  // Compute differentiable loss using Gumbel-Softmax (Eq. 4-8)
  $L \leftarrow$ ComputeLoss(M, G, τ)
  // Backpropagation and parameter update
  Compute gradients ∇L w.r.t. M
  Update M using Opt and ∇L
  if $step$ mod *sample_steps* = 0 **then**
    $P_{\text{eval}} \leftarrow$ Softmax(M.GetLogits(G))
    // Algorithm 3
    $A_{\text{sampled}} \leftarrow$ CycleFreeSampling(P_{eval}, G)
    $cost_{\text{sampled}} \leftarrow$ CalculateSampledCost(A_{sampled})
    if $cost_{sampled} < cost_{best}$ **then**
      $cost_{\text{best}} \leftarrow cost_{\text{sampled}}$
      $A_{\text{best}} \leftarrow A_{\text{sampled}}$
return A_{best}

- **Core:** The goal is to learn an optimal, probabilistic partitioning by minimizing a differentiable cost function. We use an iterative optimization loop, described in Algorithm 1, where a Scorer (explained in 3.2) evaluates all potential parent-child pairings for each node. These scores are used within a cost function that balances cluster size and synchronization overhead. Gradients are backpropagated to update the GNN and Scorer weights, guiding the model toward a low-cost probabilistic solution.
- **Cycle-free Sampling:** The goal is to convert the learned probabilistic solution into a concrete (i.e, deterministic assignments) and valid set of clusters. The output from the core stage is a probability distribution over parent choices for each node. We develop an efficient *Cycle-free Sampling* algorithm that uses these probabilities to sample a concrete partition selection for every node, ensuring the final output is a valid acyclic partition. This final partition is passed to the downstream simulation workflow, as depicted in Figure 3.

The final output partition from DiffPart drives the downstream simulation workflow. A scheduler (e.g., Verilator's custom static scheduler) uses this partition to design a scheduling strategy. This is further compiled into an efficient and multithreaded executable, that is then run on a multicore CPU.

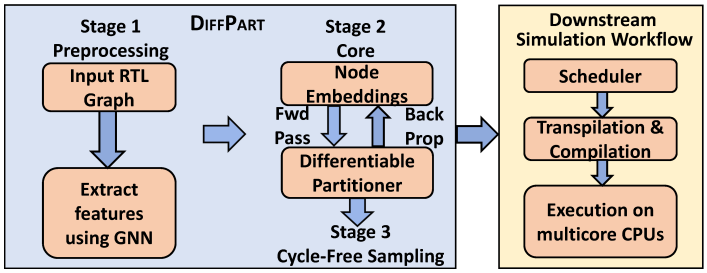

Figure 3: DiffPart framework. First, a GNN extracts features from the input RTL graph. The core stage then learns a probabilistic partitioning via a differentiable optimization loop. Finally, Cycle-free Sampling generates a concrete partition that is passed downstream for execution on a multicore CPU.

3.1 Graph Preprocessing and Feature Engineering

Instead of relying on hard-coded heuristics, we learn directly from the graph representation itself. For each node $v_i \in V$, we construct a feature vector $\mathbf{x}_i \in \mathbb{R}^6$ comprising six important features: (1) computational cost, (2) forward critical path delay, (3) reverse critical path delay, (4) fanout, (5) fanin, and (6) normalized graph depth. Most values are already internally computed by Verilator. Graph depth is determined using Kahn's algorithm [23]. The complete feature matrix $\mathbf{X} \in \mathbb{R}^{|V|\times 6}$ is then min-max normalized to the range $[0, 1]$ for backpropagation stability:

$$\mathbf{x}'_{i,k} = \frac{\mathbf{x}_{i,k} - \min_j(\mathbf{x}_{j,k})}{\max_j(\mathbf{x}_{j,k}) - \min_j(\mathbf{x}_{j,k})} \tag{1}$$

3.2 GNN Architecture for Node Embeddings

Since the input RTL graph can vary arbitrarily depending on the input design, we employ a message-passing GNN to learn expressive node embeddings, $\mathbf{e}_i \in \mathbb{R}^{d_{embed}}$ (d_{embed} is a higher dimensional space), that capture both local and global graph topology. The GNN consists of two Graph Convolution layers. The update rule for a node v_i at layer $(l+1)$ is given by:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathrm{BN}\left(\mathbf{W}^{(l)} \sum_{j\in\mathcal{N}(i)\cup\{i\}} \mathbf{h}_j^{(l)}\right)\right) \tag{2}$$

where $\mathbf{h}_i^{(l)}$ is the embedding of node v_i at layer l, $\mathcal{N}(i)$ is the neighborhood of v_i, $\mathbf{W}^{(l)}$ is a trainable weight matrix, BN is Batch Normalization, and σ is the ReLU activation function.

The node embeddings are then used by the *Scorer*, a two-layer MLP, to evaluate the quality of assigning a given node to a potential partition (represented by another node). For any given node v_i and a candidate option v_j, the Scorer takes their concatenated embeddings, $[\mathbf{e}_i; \mathbf{e}_j]$, and processes them through two dense layers with a ReLU activation to produce a single scalar logit. This logit l_{ij} represents the predicted quality of that specific assignment.

A differentiable loss is computed from the logits and the gradient of this loss is backpropagated to jointly optimize the Scorer MLP and GNN. We initialize weights using the Kaiming uniform method [17] to mitigate vanishing or exploding gradients.

3.3 Continuous Relaxation via Gumbel-Softmax

The deterministic assignment process is non-differentiable due to the `argmax` operation over the logits l_{ij}, which presents a key challenge. We overcome this by using the Gumbel-Softmax technique [37], which provides a continuous approximation for sampling from a categorical distribution. This yields a differentiable probability p_{ij} for assigning node

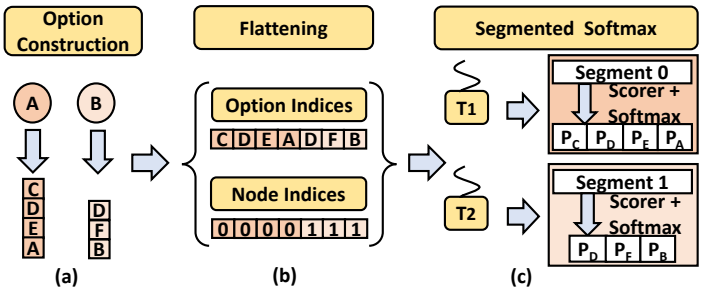

Figure 4: Vectorized Probability Calculation. (a) Option Construction: The initial data structure, where each node has a variable number of options, makes it inefficient for parallel processing. (b) Flattening: To enable vectorization, all options are flattened into an option_indices vector to gather embeddings, and a node_indices vector to map each logit back to its source node (Segment ID). (c) Segmented Softmax: Multiple CPU threads T_i process segments in parallel. Each thread applies a scorer and softmax to its assigned segment's options to compute the logits and final probabilities (e.g., P_A) for the options of each node.

v_i to the cluster representated by v_j:

$$p_{ij} = \frac{\exp((l_{ij} + g_{ij})/\tau)}{\sum_{k\in C_i} \exp((l_{ik} + g_{ik})/\tau)} \tag{3}$$

where $g_{ij} \sim \mathrm{Gumbel}(0, 1)$ are i.i.d. Gumbel noise samples and τ is a temperature parameter. As $\tau \to 0$, this distribution approaches a one-hot selection. We anneal τ during backpropagation to gradually shift from an exploratory to an exploitative solution space.

For efficiency, we compute all probabilities in a fully vectorized manner, as illustrated in Figure 4. We construct a `node_indices` vector, which maps each potential assignment back to its source node, and an `option_indices` vector, which flattens all candidate options into a single list. This allows us to gather GNN embeddings and compute a single flat vector of logits in parallel. A highly efficient segmented softmax, grouped by `node_indices`, is then applied to this vector to compute the final probabilities.

3.4 Differentiable Loss Function

The graph partitioning problem seeks to find an assignment function, $\pi : V \to V$, that maps each node $v_i \in V$ to a representative node v_j, under the constraint that v_j must be in the set of ancestors of v_i or be v_i itself ($v_j \in \mathrm{ancestors}(v_i) \cup \{v_i\}$). This mapping defines a set of clusters $K = \{k \mid \exists i, \pi(i) = k\}$. The objective is to find an optimal partition π^* that minimizes a cost function:

$$C(\pi) = \lambda_{\text{cluster}} C_{\text{cluster}}(\pi) + \lambda_{\text{sync}} C_{\text{sync}}(\pi) \tag{4}$$

where $C_{\text{sync}}(\pi) = |K|$ is the number of clusters and $C_{\text{cluster}}(\pi)$ penalizes cost imbalance between clusters. λ_{cluster} and λ_{sync} are the weights of each component of the cost function. As synchronization cost is directly proportional to thread count T, λ_{sync} is modelled to increase with increase in T as follows:

$$\lambda_{\text{sync}}(T) = \lambda_{\text{sync,low}} + \left(\lambda_{\text{sync,high}} - \lambda_{\text{sync,low}}\right) \cdot \tanh\left(\frac{T-1}{k}\right) \tag{5}$$

where $\lambda_{\text{sync,low}} = \lambda_{\text{cluster}}/10$ and $\lambda_{\text{sync,high}} = \lambda_{cluster}/2$ to prevent λ_{sync} from skewing Eqn. 4 and k is the smoothing factor.

We reframe the problem in a continuous domain. Instead of making a deterministic assignment for each node v_i, we learn a probability distribution P_i over all valid candidate representatives $C_i = \{v_i\} \cup \mathrm{parents}(v_i)$.

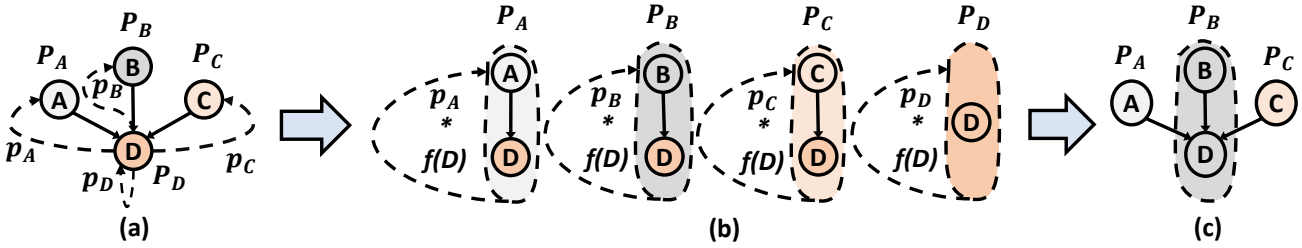

Figure 5: Illustration of the core mechanism of the differentiable cost function. (a) Consider node D which has probabilities p_a, p_b, p_c, p_d of forming a partition with A,B,C and D respectively denoted as P_A, P_B, P_C, P_D. (b) The expected cost of a potential partition is computed by propagating costs of D denoted by $f(D)$ multiplied by the respective probability. (c) Finally, D is assigned to the partition which has the highest probability value which is free of cycles: in this case P_B.

This yields a probability matrix $\mathbf{P} \in [0,1]^{|V|\times|V|}$, where $p_{ij} = P(\pi(i) = j)$ for $j \in C_i$, and $p_{ij} = 0$ otherwise. This transformation allows us to define a differentiable loss function based on the expected values of our objectives.

With the continuous probability matrix $\mathbf{P}$, we define a differentiable loss function based on the expected values of our objectives. The total loss is $L = \lambda_{\text{cluster}} \cdot \hat{L}_{\text{cluster}} + \lambda_{\text{sync}} \cdot \hat{L}_{\text{sync}}$.

- **Expected Synchronization Cost**: The differentiable proxy for the number of clusters is the expected number of clusters, computed as the sum of probabilities of each node becoming its own cluster representative:

$$\hat{L}_{\text{sync}} = \mathbb{E}[|K|] = \sum_{i \in V} P(\pi(i) = i) = \sum_{i \in V} p_{ii} \tag{6}$$

- **Expected Cluster Cost Variance**: To facilitate load balancing, we penalize the variance of cluster costs. We use a differentiable cost propagation algorithm, detailed in Algorithm 2 and illustrated in Figure 5, that iteratively computes the expected accumulated cost $A_i^{(d)}$ for each node v_i in an efficient manner. The update is given by:

$$A_i^{(d)} = \text{cost}(i) + \sum_{j \in \text{parents}(i)} p_{ji} \cdot A_j^{(d-1)} \tag{7}$$

We repeat this update for D steps, where D is the maximum depth of the graph. The loss term is then formulated as the sum of squared expected costs for each potential cluster, encouraging costs $A_i^{(D)}$ to be small and uniform:

$$\hat{L}_{\text{cluster}} = \sum_{i \in V} (p_{ii} \cdot A_i^{(D)})^2 \tag{8}$$

Algorithm 2: Differentiable Cost Propagation

Input: Graph $G(V, E)$, Initial costs $C_0 \in \mathbb{R}^{|V|}$, Probabilities $P \in \mathbb{R}^{|E|}$
Output: Final accumulated costs $A^{(D)}$

$A \leftarrow C_0$ // Initialize accumulated costs $A^{(0)}$
$D \leftarrow \text{MaxGraphDepth}(G)$
$S, R \leftarrow G.\text{senders}, G.\text{receivers}$ // Get senders/receivers indices for all edges
for $d \leftarrow 1$ **to** D **do**
  $A_{\text{senders}} \leftarrow A[S]$ // Gather all sender nodes $A^{(d-1)}$
  $M \leftarrow A_{\text{senders}} \odot P$ // Calculate weighted messages for all edges
  $M_{\text{agg}} \leftarrow \text{SegmentSum}(M, R, |V|)$ // Aggregate messages by receiver nodes
  $A \leftarrow C_0 + M_{\text{agg}}$ // Update total costs $A^{(d)}$
return A // Return final costs $A^{(D)}$

Both loss terms are normalized to improve backpropagation stability. To ensure that the cluster variance and synchronization cost terms are balanced and numerically stable throughout the backpropagation phase, each term is normalized to the range $[0, 1]$ before being weighted by its respective λ value. We normalize the terms using analytically derived bounds. For the cluster variance term, the lower bound is the sum of each node's squared cost (representing a partition where every node is its own cluster), and the upper bound is the squared sum of all node costs (representing a single massive cluster). For the synchronization term, the bounds are simply 1 and the total number of nodes, $|V|$.

3.5 Cycle-free Sampling

After every certain backpropagation iterations, we convert the learned probabilistic partitioning into a deterministic assignment by setting the Gumbel-Softmax temperature $\tau \to 0$, effectively recovering the standard softmax probabilities from the model's logits. However, the concrete partitions may contain cycles as shown in Figure 6, which is invalid. We present an efficient approach in Algorithm 3 which ensures that no cycles are formed.

Algorithm 3: Cycle-free Sampling

Input: Softmax probabilities P, Graph G
Output: Final partition assignments A

$T \leftarrow \text{TopologicalSort}(G)$
$A \leftarrow$ empty map ; $G_{\text{cluster}} \leftarrow$ empty graph
foreach *node u in T* **do**
  $C_u \leftarrow \text{Sorted}(\text{Parents}(u) \cup \{u\})$ by P_u descending
  assigned $\leftarrow$ **False**
  foreach *candidate v in* C_u **do**
    $rep_v \leftarrow A[v]$ **if** $v \in A$ **else** v // Get v's cluster
    is_cyclic $\leftarrow$ **False**
    foreach *parent p of u* **do**
      if $A[p] \neq rep_v$ ***and*** *path exists from* $rep_v \to A[p]$ *in* $G_{cluster}$ **then**
        is_cyclic $\leftarrow$ **True**; **break**
    if ***not*** *is_cyclic* **then**
      $A[u] \leftarrow rep_v$; *assigned* $\leftarrow$ **True**; **break**
  // u is always a candidate in C_u, which denotes self-cluster and is free of cycles
  // Update cluster dependencies for the final assignment
  foreach *parent p of u* ***if*** $A[p] \neq A[u]$ **do**
    Add edge $A[p] \to A[u]$ to G_{cluster}
return A

The algorithm processes nodes in topological order and assigns each node to the highest-probability candidate cluster that does not introduce a cycle in the evolving cluster dependency graph. Because each node is always eligible to form its own singleton cluster, which by construction cannot create a cycle, the procedure is guaranteed to produce a valid, acyclic and executable partition.

4 Experimental Evaluation

We evaluate the performance of DiffPart on a diverse RISC-V benchmark, spanning a range of complexity, from functional units such as encryption, cryptography, deep learning accelerators to general purpose processors and SoCs [1, 2, 21, 42, 44, 47, 48]. Table 1 summarizes key

Table 1: Overall performance comparison between Verilator and DiffPart under 16 threads, where the parallel efficiency saturates. "-" means Verilator fails to find a partition due to its hard-coded partitioning constraints.

Benchmark				Simulation Runtime (s)			Partitioning Time (s)				Partitioned # mTasks	
Name	# LOC	# mTasks	# Edges	Verilator	DiffPart	Speedup	Verilator	DiffPart			Verilator	DiffPart
								Total	Sampling	BackProp		
RocketChip	81,306	10,376	32,249	11.53	4.76	2.42×	3.31	77.95	6.44%	92.13%	431	1
Conv2D	10,605	6,340	13,781	7.56	2.64	2.86×	0.29	44.93	6.08%	91.17%	128	10
GEMM	7,850	5,079	10,706	6.23	1.12	5.54×	0.14	35.16	7.36%	89.22%	110	2
FMUL	53,380	3,455	7,231	-	3.34	-	-	24.97	9.39%	83.06%	-	1
RocketCore	7,248	1,090	2,982	11.89	4.60	2.58×	0.40	13.68	4.47%	77.37%	620	70
FPU	4,482	511	1,346	6.84	5.61	1.22×	0.15	9.15	2.10%	68.34%	163	454
RISCVMini	3,315	491	1,437	6.78	0.27	24.49×	0.08	9.16	3.02%	69.08%	239	5
SodorCore	3,152	291	798	7.50	4.02	1.87×	0.07	7.94	1.14%	65.39%	261	168
SHA256	1,084	145	376	4.69	0.08	55.25×	0.01	6.498	1.03%	59.21%	138	1

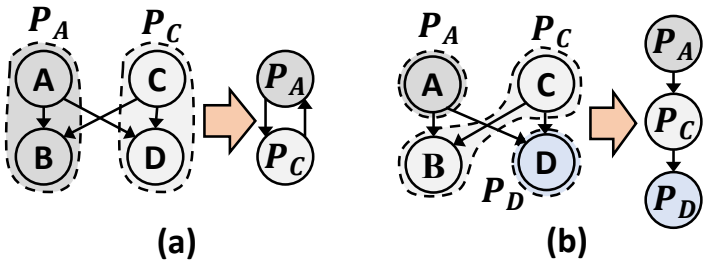

Figure 6: Cycle formation during sampling. (a) An invalid partitioning due to cyclic dependencies between P_A and P_C. (b) A valid partitioning result where P_A includes node A, P_C includes nodes B and C, and P_D includes node D.

graph statistics, while Table 2 categorizes each design. All experiments are conducted on a 64-bit Linux machine with 20-core Intel Core i5-13500 CPU and 125 GB RAM.

Our implementation utilizes JAX [3], a high-performance numerical computing library well-suited for differentiable optimization. The GNN model uses an input dimension of 6, a hidden dimension of 16, and a final embedding size of 8. During the backpropagation phase, the Gumbel-Softmax temperature is annealed from an initial value of $\tau_0 = 50.0$ to a minimum of $\tau_{\min} = 0.8$, using a multiplicative decay factor of 0.985 per step. We use the AdamW optimizer with a learning rate $\eta = 0.01$. The loss function combines clustering and synchronization objectives, with weights $\lambda_{\text{cluster}} = -1.0$, $\lambda_{\text{sync,low}} = -0.1$, and $\lambda_{\text{sync,high}} = -0.5$. The smoothing factor in Eqn 5 is set to $k = 27$. Each design is trained for 400 iterations, with sampling performed every 10 steps to monitor solution quality. All data shown are averaged over ten runs.

Table 2: Types of benchmarks used in the evaluation.

Benchmark	Type	Benchmark	Type
RocketChip	System-on-Chip	FPU	Floating-Point Unit
Conv2D	Accelerator	RISCVMini	Processing Core
GEMM	Matrix Multiplier	SodorCore	Processing Core
FMUL	Float Multiplier	SHA256	Functional Unit
RocketCore	Processing Core		

4.1 Baseline

We use Verilator's [43] built-in partitioner as our baseline. Verilator is a widely used, state-of-the-art RTL simulator that supports multithreading for faster simulation runtime. Internally, Verilator partitions an input design into dependent macro tasks and leverages TGP for parallel simulation. Verilator employs a cost-based heuristic algorithm, inspired by [39], to assign logic blocks to threads. This algorithm aims to balance computational load while preserving data dependencies. However, as discussed in Section 1, this algorithm ignores real threading costs but a proxy combinatorial objective. It also depends on the user to select a number of simulation threads, which, if chosen incorrectly, may cause the partitioner to fail due to hard-coded partitioning constraints (e.g., critical path length must converge to a threshold [39]) or lead to poor workload balancing. Together, these limitations often result in suboptimal simulation performance.

4.2 Overall Performance Analysis

Table 1 compares the overall simulation runtime, partitioning time, and the number of partitioned macro tasks generated between DiffPart and Verilator at 16 threads, where simulation performance saturates on our machine. Note that since Verilator is a cycle-by-cycle simulator, its runtime scales linearly with the number of cycles, which would only proportionally increase the performance gap relative to our method. Hence, we use 1M cycles for all simulations, a reasonable number for large-scale RTL simulation [33].

As shown in Table 1, DiffPart consistently outperforms Verilator in terms of simulation runtime across all designs. The maximum observed speedup is 55.25× for SHA256. In most cases, DiffPart learns to generate a smaller number of partitioned mTasks that minimize the overall threading cost. However, there are cases, such as the FPU, where DiffPart generates more mTasks to achieve faster simulation runtime. This result highlights the effectiveness of our differentiable cost function in capturing the trade-off between available parallelism and threading overhead under real operating conditions.

Another strength of DiffPart is that it consistently produces valid partitions without requiring the user to explicitly specify or fine-tune the number of threads. In contrast, we observed that Verilator sometimes fails to generate a valid partition for certain benchmarks, such as FMUL,

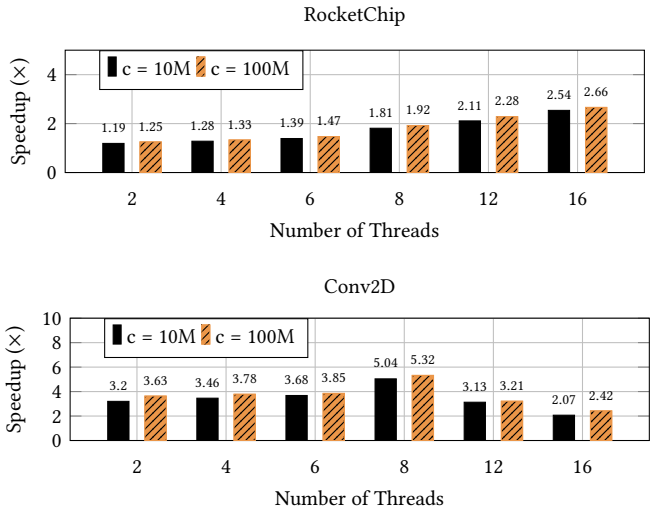

Figure 7: Speedup for RocketChip and Conv2D across different thread counts at 10 million and 100 million simulation cycles(c). In general, speedup increases slightly with the number of cycles.

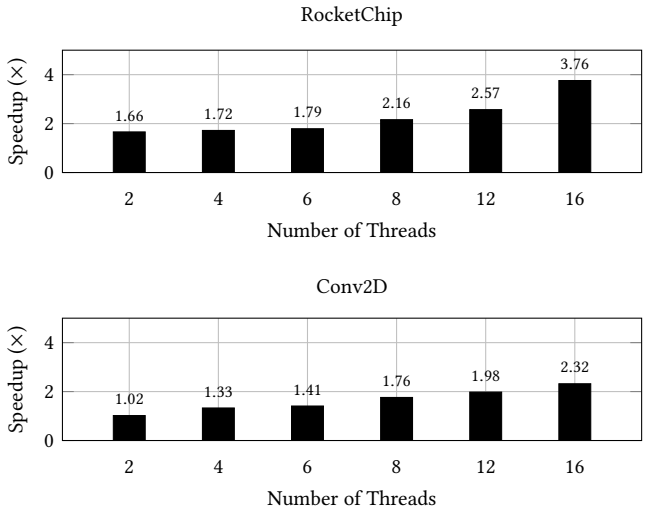

Figure 8: Speedup for RocketChip and Conv2D across thread counts for 1 million simulation cycles when using a dynamic scheduler in place of Verilator's default backend. DiffPart consistently improves simulation performance, highlighting its robustness to different downstream simulation workflows.

and prompts the user to select a different number of threads for partitioning. This behavior highlights Verilator's reliance on time-consuming parameter tuning, which adds extra burden to the user.

While DiffPart incurs higher partitioning time compared to Verilator, this overhead is expected. Verilator employs a fast greedy heuristic, whereas DiffPart iteratively optimizes toward a solution through gradient-based updates. We observe that the vast majority of this time is productively used in the core backpropagation optimization loop, with this phase consistently consuming between 59% and 92% of the total budget. In Table 1, under partitioning time, the Sampling and BackProp percentages for DiffPart do not sum to 100% because the remainder is attributed to constant setup costs, such as graph construction and array initialization. These costs represent a larger share of the runtime for smaller designs (e.g., SHA256) but become negligible for larger ones (e.g., RocketChip). Notably, our Cycle-free Sampling algorithm is highly efficient, consistently accounting for less than 10% of the total partitioning time. That being said, the partitioning process incurs only a one-time compilation cost, which is a favorable trade-off given the significant simulation speedups achieved.

4.3 Partition Quality Analysis

We evaluate the performance of DiffPart at a larger scale by increasing the number of simulation cycles. This is critical, as industrial-scale verification often requires hundreds of millions of cycles for thorough coverage [33]. Figure 7 presents the speedup of DiffPart over Verilator for the two largest designs, RocketChip and Conv2D, evaluated at 10 million and 100 million cycles across different thread counts, $T = \{2, 4, 6, 8, 12, 16\}$.

A key observation is that for a fixed number of threads, the speedup from DiffPart improves as the number of cycles increases. For instance, with Conv2D at $T = 8$, the speedup grows from 4.76× at 1 million cycles to 5.32× at 100 million cycles. We attribute this observation to the high-quality partitions produced by DiffPart, which more effectively reduce the cumulative runtime overhead from thread management, a factor that becomes increasingly significant in longer simulations. For Conv2D, we observe diminishing returns at higher thread counts, with speedup peaking at $T = 8$. Nonetheless, DiffPart consistently outperforms the baseline partitions. We attribute this to our algorithm, which, unlike hard-coded heuristics, uses gradient-based updates to minimize a differentiable cost function and explicitly optimize for a lower execution runtime under any operating conditions.

4.4 Robustness Study

Since DiffPart targets framework-level innovation, we evaluate its robustness by testing it across alternative downstream simulation workflows. Specifically, we replace Verilator's hard-coded static scheduler with a widely-used dynamic scheduler, Taskflow [18], to execute the partitions generated by DiffPart. This setup allows us to assess the robustness of our partitions independently of any specialized execution engine that may require non-trivial implementation or integration.

Figure 8 shows the speedup of DiffPart over Verilator for RocketChip and Conv2D across different thread counts $T = \{2, 4, 6, 8, 12, 16\}$ for 1 million simulation cycles. Across all thread counts, DiffPart consistently outperforms Verilator, with the performance gap widening as T increases, demonstrating that its benefits are not tied to a specific scheduler. On average, DiffPart achieves speedups of 2.35× for RocketChip and 1.6× for Conv2D, with similar trends observed for smaller designs. These results highlight the robustness of DiffPart to different simulation backends.

5 Conclusion

In this paper, we have introduced DiffPart, a differentiable task graph partitioning framework that automatically learns how to derive high-quality partitions under real operating conditions. Using RTL simulation as a case study, DiffPart improves state-of-the-art Verilator's partitioning quality by achieving up to 1.22–55.25× faster runtime across various hardware designs. Although DiffPart is initially targeted at RTL simulation, the results are very encouraging. We believe the ideas behind DiffPart naturally extend to other TGP-powered EDA applications, since they all rely on efficiently partitioning dependent tasks for parallel execution. Inspired by our research [4, 6, 9, 10, 12–16, 19, 26–30, 32, 36, 38, 40, 45, 46, 49, 50], we plan to enhance the performance of DiffPart using GPU.

Acknowledgment

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

[1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* (2020). https://doi.org/10.1109/MM.2020.2996616

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. *The Rocket Chip Generator*. Tech. Rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Berkeley, CA, USA.

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

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

[5] Chih-Chun Chang and Tsung-Wei Huang. 2025. Late Breaking Results: Statistical Timing Graph Scheduling Algorithm for GPU Computation. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC63849.2025.11133291

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

[7] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2024. An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. https://doi.org/10.1109/ISVLSI61997.2024.00149

[8] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *Euro-Par 2021: Parallel Processing Workshops – Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30–31, 2021, Revised Selected Papers*. Springer-Verlag. https://doi.org/10.1007/978-3-031-06156-1_37

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

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

[11] Charles M. Fiduccia and Richard M. Mattheyses. 1982. A Linear-Time Heuristic for Improving Network Partitions. *Proceedings of the 19th Design Automation Conference* (1982).

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

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

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

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

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

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision (ICCV)*.

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

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

[20] Tsung-Wei Huang and Martin DF Wong. 2015. OpenTimer: A high-performance timing analysis tool. In *ICCAD*. IEEE.

[21] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2021. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE.

[22] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. *BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3676641.3715984

[23] Arthur B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* (1962).

[24] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* (1998).

[25] Brian W Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal* (1970).

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

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

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

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

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

[31] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation Using Task Graph Parallelism. In *Euro-Par 2021: Parallel Processing – 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings*. Springer-Verlag. https://doi.org/10.1007/978-3-030-85665-6_27

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

[33] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ICPP*.

[34] Yibo Lin, Wuxi Li, Jiaqi Gu, Haoxing Ren, Brucek Khailany, and David Z. Pan. 2020. ABCDPlace: Accelerated Batch-Based Concurrent Detailed Placement on Multithreaded CPUs and GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[35] Siting Liu, Yuan Pu, Peiyu Liao, Hongzhong Wu, Rui Zhang, Zhitang Chen, Wenlong Lv, Yibo Lin, and Bei Yu. 2023. FastGR: Global Routing on CPU–GPU With Heterogeneous Task Graph Scheduler. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2023).

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

[37] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In *International Conference on Learning Representations (ICLR)*.

[38] Chedi Morchdi, Cheng-Hsiang Chiu, Wan-Luan Lee, Tsung-Wei Huang, and Yi Zhou. 2025. Enhancing Graph Partitioning with Reinforcement Learning-based Initialization. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[39] Vivek Sarkar. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press.

[40] Aditya Das Sarma, Shui Jiang, Wan Luan Lee, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 266–272. https://doi.org/10.1109/ASP-DAC66049.2026.11420570

[41] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* (2009). https://doi.org/10.1109/TNN.2008.2005605

[42] Anish Singhani. n.d.. Cryptography accelerator core (for AES128/AES256 and SHA256) designed in CHISEL3, primarily targeting ASIC platforms. https://github.com/asinghani/crypto-accelerator. Accessed: 2025-11-01.

[43] Wilson Snyder. 2018. Verilator 4.0: Open Simulation Goes Multithreaded. In *ORConf*.

[44] Sebastian Strempfer, Kazutomo Yoshii, Mike Hammer, Dawid Bycul, and Antonino Miceli. 2021. Designing a Streaming Data Coalescing Architecture for Scientific Detector ASICs with Variable Data Velocity. In *Proceedings of the 2021 3rd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*.

[45] Jie Tong, Wan-Luan Lee, Umit Yusuf Ogras, and Tsung-Wei Huang. 2025. Scalable Code Generation for RTL Simulation of Deep Learning Accelerators with MLIR. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

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

[47] UCB BAR. n.d.. riscv-mini: Simple RISC-V 3-stage Pipeline in Chisel. https://github.com/ucb-bar/riscv-mini. Accessed: 2025-11-01.

[48] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO56248.2022.00080

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

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