MedPart: A Multi-Level Evolutionary Differentiable Hypergraph Partitioner

Rongjian Liang
rliang@nvidia.com
NVIDIA
Austin, TX, USA

Anthony Agnesina
aagnesina@nvidia.com
NVIDIA
Austin, TX, USA

Haoxing Ren
haoxingr@nvidia.com
NVIDIA
Austin, TX, USA

ABSTRACT

State-of-the-art hypergraph partitioners, such as hMETIS, usually adopt a multi-level paradigm for efficiency and scalability. However, they are prone to getting trapped in local minima due to their reliance on refinement heuristics and overlooking global structural information during coarsening. SpecPart, the most advanced academic hypergraph partitioning refinement method, improves partitioning by leveraging spectral information. Still, its success depends heavily on the quality of initial input solutions. This work introduces MedPart, a multi-level evolutionary differentiable hypergraph partitioner. MedPart follows the multi-level paradigm but addresses its limitations by using fast spectral coarsening and introducing a novel evolutionary differentiable algorithm to optimize each coarsening level. Moreover, by analogy between hypergraph partitioning and deep graph learning, our evolutionary differentiable algorithm can be accelerated with deep graph learning toolkits on GPUs. Experiments on public benchmarks consistently show MedPart outperforming hMETIS and achieving up to a 30% improvement in cut size for some benchmarks compared to the best-published solutions, including those from SpecPart—moreover, MedPart's runtime scales linearly with the number of hyperedges.

CCS CONCEPTS

• **Hardware → Methodologies for EDA**; • **Mathematics of computing → Hypergraphs**.

KEYWORDS

hypergraph partitioning, GPU acceleration, gradient descent

ACM Reference Format:
Rongjian Liang, Anthony Agnesina, and Haoxing Ren. 2024. MedPart: A Multi-Level Evolutionary Differentiable Hypergraph Partitioner. In *Proceedings of the 2024 International Symposium on Physical Design (ISPD '24), March 12–15, 2024, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3626184.3633319

1 INTRODUCTION

Hypergraphs are a natural extension of traditional graphs, representing connections among more than two vertices through hyperedges. They are, therefore, particularly adept at modeling complex multi-way relationships, rendering them invaluable across many fields [4]. In particular, a hypergraph problem of critical importance in VLSI is the balanced min-cut netlist partitioning problem [10]. This problem aims to divide the netlist hypergraph into two or more nearly equal-sized parts while minimizing the number of hyperedges (=nets) connecting vertices (=gates/modules) in different partitions. It is a fundamental combinatorial optimization problem with direct applications to floorplanning, placement, and the latest 3D ICs tier-partitioning.

1.1 Related Works

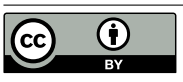
State-of-the-art hypergraph partitioners, such as hMETIS [10], KaHyPar [16], and PaToH [8], usually adopt a multi-level paradigm, progressively coarsening hypergraphs to explore a vast solution space efficiently. These coarser partitions then serve as starting points for finer-level refinement. Such a paradigm tends to be scalable because it focuses on partitioning smaller, more manageable, coarser-level graphs, reducing the computational burden. However, multi-level partitioners may encounter local optima in practice due to two critical limitations outlined in [7]: (i) Hypergraph coarsening predominantly considers local structures, neglecting global hypergraph characteristics, (ii) Refinement heuristics can become trapped in local minima. In response, SpecPart [7] introduces spectral information to refine partitioning, albeit reliant on initial solutions. However, when the initial solution is far from the global optimum, SpecPart may still fall short of achieving global optimality.

Evolutionary algorithms, such as genetic algorithms (GA) [5], have found applications in hypergraph partitioning. While these algorithms excel in systematic exploration in discrete space, they often lack efficiency in local search. Consequently, prior research has commonly resorted to hybrid approaches that combine evolutionary algorithms with local search techniques. In addition, evolutionary partitioners necessitate a substantial number of evolution generations to converge, resulting in heavy evaluation workloads, especially for large hypergraphs.

A graph neural network (GNN)-based graph partitioner introduced in [15] defines a differentiable loss function representing the partitioning objectives. It employs backward propagation to optimize the GNN parameters, enabling the GNN to predict partitioning solutions, even for previously unseen graphs. However, its loss function involves multiplications matrices of size $N \times N$, where N represents the number of vertices, limiting its scalability. Moreover, it is not designed to handle hypergraphs.

1.2 Contributions

In this work, we develop a multi-level evolutionary differentiable hypergraph partitioner named **MedPart**. It follows the multi-level paradigm but addresses its limitations by fast spectral coarsening



This work is licensed under a Creative Commons Attribution International 4.0 License.

ISPD '24, March 12–15, 2024, Taipei, Taiwan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0417-8/24/03.
https://doi.org/10.1145/3626184.3633319

and a novel evolutionary differentiable optimizer with a global view at each level. Moreover, by analogy between hypergraph partitioning and deep graph learning, our evolutionary differentiable optimizer can be accelerated with deep graph learning toolkits on GPUs. Our contributions are summarized as follows:

(1) We introduce a fast spectral hypergraph coarsening algorithm based on emerging graph signals. It can progressively coarsen a graph with hundreds of thousands of nodes in seconds.
(2) We propose an evolutionary differentiable algorithm that integrates GA and gradient descent (GD) for optimization at each coarsening level. In the GD search, we place a probability on assigning a vertex to each partition. With a differentiable and computing-efficient loss function, GD optimizes the assignment probabilities end-to-end. GA is employed to systematically generate good starting points to help the GD escape local optima.
(3) Our framework generalizes to many constraint-driven partitioning problem formulations. As long as the loss function can be optimized with differentiable optimization, both discrete and continuous objectives can be targeted, thereby expanding the space of problems that can be solved beyond traditional min-cut and ratio cut bipartitioning formulations.
(4) We accelerate the evolutionary differentiable algorithm with deep graph learning toolkits on GPUs by drawing an analogy between hypergraph partitioning and deep graph learning.
(5) Experimental results applied to balanced hypergraph bipartitioning on publicly available benchmarks show that MedPart consistently outperforms the leading partitioner hMETIS, and achieves up to a 30% improvement in cut size compared to the best-published solutions for some benchmarks—moreover, MedPart's runtime scales linearly with the number of hyperedges.

2 PRELIMINARY

This section presents some preliminaries necessary for the understanding of MedPart. We first offer the mathematical framework for hypergraph partitioning, our problem at stake. We then introduce the spectral coarsening and genetic algorithms used in MedPart. Finally, we present the mechanism of message passing in graph neural networks, which will serve as the basis for efficient implementations of MedPart routines.

2.1 Hypergraph Partitioning Formulation

A hypergraph H is defined as a pair $H = (V, E)$ where V represents the set of vertices $v \in V$ with associated weight w_v, and E represents the set of hyperedges where an hyperedge $e \in E$ is a subset of V with associated weight w_e. Given a positive integer $k \geq 2$ and a positive real number $\varepsilon \leq \frac{1}{k}$, letting $W = \sum_{v \in V} w_v$, the k-way balanced hypergraph partitioning problem can be mathematically formulated as:

$$\min_{S=\{V_1,V_2,\ldots,V_k\}} \text{cutsize}_H(S) = \sum_{\{e \mid e \nsubseteq V_i, \forall i\}} w_e \tag{1}$$

$$\text{s.t.} \quad \cup_{i=1}^{k} V_i = V \quad \text{and} \quad V_i \cap V_j = \emptyset, \quad 0 \leq i, j \leq k, \tag{2}$$

$$\left(\frac{1}{k} - \varepsilon\right) W \leq \sum_{v \in V_i} w_v \leq \left(\frac{1}{k} + \varepsilon\right) W, \quad 0 \leq i \leq k, \tag{3}$$

where Eq. (2) ensures that S is a k-way disjoint partitioning solution of V, and ε is the allowed imbalance between partitions (Eq. (3)). We say that S is an ε-balanced partitioning solution.

For simplicity but without loss of generality, this work focuses on traditional bipartitioning scenarios where $k = 2, w_v = 1, \forall v \in V$, and $w_e = 1, \forall e \in E$. Note, however, that our framework readily applies to more general scenarios.

2.2 Spectral Graph Embeddings and Coarsening

Let $G = (V, E, w)$ be a weighted graph. The Laplacian matrix L_G of G is defined as follows:

$$L_G(u, v) = -w_{e_{u,v}}, \text{ if } u \neq v, u, v \in V,$$
$$L_G(u, u) = \sum_{v \neq u} w_{e_{u,v}}, \text{ for } u \in V.$$

Suppose the eigenvalues of L_G are $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_N$ and the corresponding eigenvectors are $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots \boldsymbol{u}_N$, where $\boldsymbol{u}_i, 1 \leq i \leq \text{N}$ is a vector of length of N (# of vertices). An effective way to represent the graph's global structure information is to embed the graph into an n-dimensional space using the first n ($1 \leq n \leq N$) eigenvectors of the graph Laplacian matrix, also known as the spectral graph embedding technique. Next, the graph vertices close to each other in the low-dimensional embedding space can be aggregated to form the coarse-level nodes and, subsequently, the reduced graph. However, calculating the eigenvectors of the original Laplacian graph is very costly, especially for large graphs.

A fast spectral coarsening method is developed in [9] based on emerging graph signal processing techniques. A graph signal $\boldsymbol{g} = \{g_1, g_2, \ldots, \text{g}_N\}$ is defined as a vector that ensembles the individual values on all vertices. A random graph signal $\boldsymbol{g}$ can be expressed with a linear combination of eigenvectors of the graph Laplacian, i.e., $\boldsymbol{g} = \sum_{i=1}^{N} \alpha_i \boldsymbol{u}_i$. Instead of directly using the first few eigenvectors of the original graph Laplacian as the graph embedding, [9] proposes to apply a low-pass graph filtering function to n random graph signal vectors to obtain smoothed vectors for n-dimensional graph embedding, which can be achieved in linear time. Applying the smoothing function on $\boldsymbol{g}$, a smoothed vector $\tilde{\boldsymbol{g}}$ can be obtained as the linear combination of the first few eigenvectors, i.e., $\tilde{\boldsymbol{g}} = \sum_{i=1}^{n} \tilde{\alpha}_i \boldsymbol{u}_i, n \ll N$. It is suggested in [9] that these smoothed graph signals preserve important spectral properties.

2.3 Genetic Algorithms

A general framework of GA is outlined in Alg. 1. The `GenOffspring` function, as indicated in Line 3, generates a fresh set of solutions referred to as "offspring." These offspring are created based on the genetic operations of mutation and crossover applied to the individuals in the current population. The `UpdatePopulation` function in Line 5 incorporates the offspring into the existing population based on the fitness scores of the current population and the offspring.

Maintaining population diversity is critical in GAs to prevent premature convergence and effectively explore the entire search space. Various offspring generation and population update methods have been proposed to ensure diversity. For example, tournament selection [17] is a commonly used method for selecting individuals from a population to serve as parents for the next generation. It

Algorithm 1: Genetic Algorithm

Input: I: number of generations; initial population
Output: best solution
/* Evaluation */
fitness_scores = EvalFitness(population)
/* I generations */
for $i \leftarrow 1$ *to* I **do**
  /* Generate offspring by crossover and mutation */
  offspring = GenOffsppring (population, fitness_score)
  /* Evaluation */
  offspring_fitness_scores = EvalFitness(population)
  /* Update population and fitness score */
  population, fitness_scores = UpdatePopulation(population, fitness_scores, offspring, offspring_fitness_scores)
end
return best solution among population

mimics a tournament-style competition among individuals to determine who will be chosen as parents. Deterministic crowding [13] is a population update mechanism that ensures diversity by replacing a parent with its offspring only if the offspring is more fit and genetically similar to the parent, aiming to explore distinct regions of the solution space. In addition, GAs, while adept at systematic exploration in discrete spaces, often lack efficiency in local search, leading to common hybrid approaches that combine evolutionary algorithms with local search techniques in prior research.

2.4 Message Passing in Graph Neural Networks

Message passing is a core process in GNNs through which vertices in a graph communicate and exchange information with their neighboring nodes to aggregate and update their features. Fig. 1 outlines the message-passing process. Each vertex in the graph is associated with an initial feature vector. The GNN computes a message for each vertex by combining information from neighboring vertices and applying transformations to the aggregated features. Deep graph learning toolkits, such as Deep Graph Library (DGL), accelerate message passing in GNNs by leveraging efficient data structures, advanced caching and memoization techniques, parallelism, and GPU acceleration.

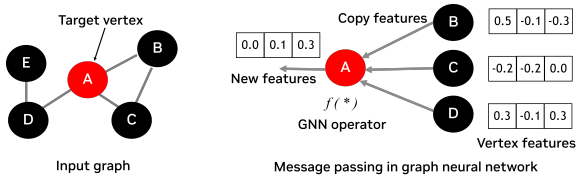


Figure 1: Message passing in graph neural networks. A message is computed for each vertex by combining information from its neighboring vertices and applying transformations to the aggregated features. This mechanism will enable efficient loss/objective computation during partitioning.

3 MEDPART MULTI-LEVEL OPTIMIZATION

3.1 MedPart Overview

Fig. 2 depicts the overview of MedPart. MedPart takes as input a hypergraph and two constraint parameters, k (# partitions) and ε (allowed imbalance), and outputs the best partitioning solution. It comprises two key phases: (1) Spectral coarsening on the hypergraph, which progressively reduces the size of the hypergraph and constructs the graph coarsening levels top-down; (2) Coarse-to-fine partitioning, applying bottom-up our evolutionary differentiable

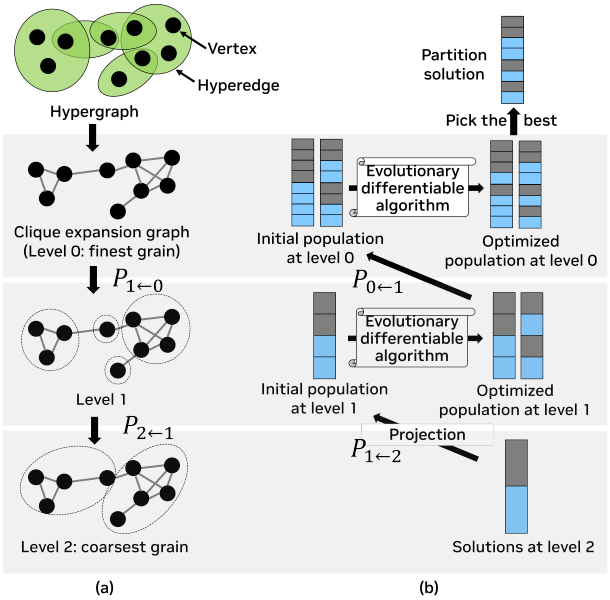


Figure 2: Overview of MedPart. (a) Spectral graph coarsening on a hypergraph. The hypergraph transformed to a clique expansion graph is progressively coarsened into several clusters for scalability. Projection matrices encoding the coarsening for use in (b) are built concurrently. (b) Multi-level optimization framework of MedPart. Partitioning assignments at coarser level l are used as a starting point for evolutionary differentiable optimization at finer level $l-1$.

algorithm from the coarser level to the finer level. There, the coarser partitions serve as starting points for finer-level refinement. We introduce the following notations to simplify the presentation.

- The integers $N_0 > N_1 > \ldots > N_L$ denote the vertex counts at different levels of graph coarsening. Fig. 2 (a) illustrates the notion of graph coarsening levels, where level 0 represents the finest granularity, and level L is the coarsest level.
- $\boldsymbol{x}(l)$ represents a partitioning solution at level l, which is a matrix of size $N_l \times k$. Each row of $\boldsymbol{x}(l)$ is a one-hot vector encoding the partition block assignment of a vertex. Without causing confusion, we will omit the subscript l in $\boldsymbol{x}(l)$.
- $\tilde{\boldsymbol{x}}(l)$, also a matrix of size $N_l \times k$, represents the continuous relaxation of the partitioning solution at level l. In this matrix, each (i, j)-th element corresponds to the probability of assigning the i-th vertex to the j-th partition block. It is important to note that for any vertex $1 \le i \le N_l$, the sum of probabilities across all partition blocks is equal to 1.
- $\boldsymbol{P}_{j\leftarrow i}$ represents the binary projection matrix of size $N_j \times N_i$ that maps a partitioning solution at level i to level j. In Fig. 2, suppose a partitioning solution at level 2 is $\boldsymbol{x}(2) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\boldsymbol{P}_{1\leftarrow 2} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$, then $\boldsymbol{x}(1) = \boldsymbol{P}_{1\leftarrow 2} \cdot \boldsymbol{x}(2) = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$, the corresponding partitioning solution at level 1. Please note that the projection matrix $\boldsymbol{P}_{j\leftarrow i}$ can also be applied to continuous partitioning solution $\tilde{\boldsymbol{x}}(i)$. Furthermore, to reduce the memory overhead, we implement $\boldsymbol{P}_{j\leftarrow i}$ as a sparse matrix.

Algorithm 2: Evolutionary Differentiable Hypergraph Partitioning Algorithm

Input: I: number of generations; M: population size; Th: stagnation threshold
S: number of GD steps; T: checkpoint steps
$\mathbf{X}_0 \leftarrow \{\boldsymbol{x}_1^0, \boldsymbol{x}_2^0, \ldots, \boldsymbol{x}_M^0\}$: initial population
Output: $\boldsymbol{x}^*$: best partitioning solution

/* Evaluation */
scores($\mathbf{X}_0$) = EvalFitness($\mathbf{X}_0$)
/* I generations */
for $i \leftarrow 1$ *to* I **do**
  /* GA iteration */
  /* Generate offspring population by crossover and mutation */
  $\{\boldsymbol{c}_1^i, \boldsymbol{c}_2^i, \ldots, \boldsymbol{c}_M^i\} \leftarrow$ GenOffspring($\mathbf{X}_{i-1}$, scores($\mathbf{X}_{k-1}$))
  /* Evaluation */
  scores($\{\boldsymbol{c}_1^i, \boldsymbol{c}_2^i, \ldots, \boldsymbol{c}_M^i\}$) = EvalFitness($\{\boldsymbol{c}_1^i, \boldsymbol{c}_2^i, \ldots, \boldsymbol{c}_M^i\}$)
  /* in parallel by batching */
  for $m \leftarrow 1$ *to* M **do**
    /* GD epoch */
    Initialize the best solution for the current GD run: $\boldsymbol{c}^{*i}_m \leftarrow \boldsymbol{c}_m^i$, scores($\boldsymbol{c}^{*i}_m$) $\leftarrow$ scores($\boldsymbol{c}_m^i$)
    Select hyper-parameters π_m^i
    Initialize continuous solution: $\tilde{\boldsymbol{c}}_m^i \leftarrow$ Relax($\boldsymbol{c}_m^i$)
    for $s \leftarrow 1$ *to* S **do**
      GD update of $\tilde{\boldsymbol{c}}_m^i$ with π_m^i
      if (s MOD T == 0) *or* (s == S) **then**
        $\boldsymbol{c}_m^i \leftarrow$ Discretize($\tilde{\boldsymbol{c}}_m^i$)
        scores($\boldsymbol{c}_m^i$) = EvalFitness($\boldsymbol{c}_m^i$)
        if scores($\boldsymbol{c}_m^i$) *better than* scores($\boldsymbol{c}^{*i}_m$) **then**
          $\boldsymbol{c}^{*i}_m \leftarrow \boldsymbol{c}_m^i$
          scores($\boldsymbol{c}^{*i}_m$) $\leftarrow$ scores($\boldsymbol{c}_m^i$)
        end
      end
    end
  end
  /* Gather best solutions from GD outcome */
  $\mathbf{C}_i^* \leftarrow \{\boldsymbol{c}^{*i}_1, \boldsymbol{c}^{*i}_2, \ldots, \boldsymbol{c}^{*i}_M\}$
  scores($\mathbf{C}_i^*$) $\leftarrow$ {scores($\boldsymbol{c}^{*i}_1$), scores($\boldsymbol{c}^{*i}_2$), … scores($\boldsymbol{c}^{*i}_M$)}
  /* Update population with deterministic crowding */
  $\mathbf{X}_i$, scores($\mathbf{X}_k$) $\leftarrow$ UpdatePopulation($\mathbf{X}_{i-1}$, scores($\mathbf{X}_{k-1}$), $\mathbf{C}_i^*$, scores($\mathbf{C}_i^*$))
  /* Early stop criterion */
  if *the best fitness score does not improve for over Th generations* **then**
    $\boldsymbol{x}^* \leftarrow$ best solution from $\mathbf{X}_i$
    return $\boldsymbol{x}^*$
  end
end
return best solution $\boldsymbol{x}^*$ among $\mathbf{X}_I$

3.2 Spectral Coarsening and Multi-Level Optimization

MedPart follows the multi-level paradigm and uses a fast spectral coarsening technique to build the graph's coarsening levels. Before coarsening, the input hypergraph $H = (V, E)$ is transformed into a clique expansion graph [2], G_H, by replacing each hyperedge $e \in E$ with an edge for each pair of vertices in the hyperedge. Fig. 2 (a) illustrates the construction of a clique expansion graph from a hypergraph. Note that G_H has precisely the same vertices as H. Hence, the coarsening results on G_H also apply to H.

Next, we apply to G_H the graph signal processing-based fast spectral coarsening method discussed in Section 2.2. This process gradually reduces G_H to a graph with only few vertices, potentially just 2 or 3, and concurrently constructs the projection matrices $\boldsymbol{P}_{0\leftarrow 1}, \boldsymbol{P}_{1\leftarrow 2}, \ldots, \boldsymbol{P}_{L-1\leftarrow L}$.

Once the graph coarsening levels have been established, MedPart generates partitioning solutions progressively, starting from the coarsest granularity level and advancing to the finest granularity, as depicted in Fig. 2 (b). For levels where the total number of all possible solutions (=2^{N_l-1}) is small enough, MedPart employs

Algorithm 3: Generation Offspring GenOffspring

Input: $\mathbf{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots\}$: current population;
scores($\mathbf{X}$): fitness scores of the population;
N_cro: offspring size from crossover; N_mut: offspring size from mutation;
N_tour: tournament size; N_cp: number of crossover points;
p_cro: mutation rate for crossover offspring;
p_mut: mutation rate for mutation offspring
Output: $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots\}$: offspring

/* Initialize offspring */
offspring $\leftarrow \{\}$
/* Generate off-springs by crossover */
/* In parallel by tensor operations with PyTorch */
for $i \leftarrow 1$ *to* N_cro **do**
  /* Tournament selections */
  $\boldsymbol{p}_1$ = TournamentSel($\mathbf{X}$, scores($\mathbf{X}$), N_cp)
  $\boldsymbol{p}_2$ = TournamentSel($\mathbf{X}$, scores($\mathbf{X}$), N_cp)
  /* Align parent 2 with parent 1 */
  if $||\boldsymbol{p}_1 - \boldsymbol{p}_2\neg||_1 < ||\boldsymbol{p}_1 - \boldsymbol{p}_2||_1$ **then**
    $\boldsymbol{p}_2 \leftarrow \boldsymbol{p}_2\neg$
  end
  /* Generate an offspring by crossover */
  $\boldsymbol{c} \leftarrow$ Crossover($\boldsymbol{p}_1, \boldsymbol{p}_2$, N_cp)
  /* Mutation */
  $\boldsymbol{c} \leftarrow$ Mutate($\boldsymbol{c}$, p_cro)
  Append $\boldsymbol{c}$ to offspring
end
/* Generate off-springs by mutation */
/* In parallel by tensor operations with PyTorch */
for $i \leftarrow 1$ *to* N_mut **do**
  /* Tournament selection */
  $\boldsymbol{p}$ = TournamentSel($\mathbf{X}$, scores($\mathbf{X}$), N_cp)
  /* Mutation */
  $\boldsymbol{c} \leftarrow$ Mutate($\boldsymbol{p}$, p_mut)
  Append $\boldsymbol{c}$ to offspring
end
return offspring

enumeration to create the optimal partitioning solution. In cases where the number of potential solutions is too large, our evolutionary differentiable algorithm is employed to generate a population of high-quality solutions. These coarser-level solutions are then mapped to solutions at finer levels using the projection matrices $\boldsymbol{P}_{i\leftarrow j}$ and act as starting points for the optimization process at those finer levels. Upon completing the optimization at level 0, the finest granularity level, we report the best-found partitioning solution.

Note that solution $\boldsymbol{x}(l)$ at level l can be mapped to solution $\boldsymbol{x}(0)$ at level 0 by $\boldsymbol{x}(0) = \boldsymbol{P}_{0\leftarrow 1}\boldsymbol{P}_{1\leftarrow 2}\cdots\boldsymbol{P}_{l-1\leftarrow l}\cdot\boldsymbol{x}(l)$. Thus, an evaluation framework developed for the original hypergraph H can be seamlessly applied to assess solutions at any coarsening level.

4 EVOLUTIONARY DIFFERENTIABLE HYPERGRAPH PARTITIONING

We develop an evolutionary differentiable algorithm that combines a genetic algorithm with gradient descent to optimize partitioning at each graph coarsening level. Thanks to their fast convergence and robust scalability, GD methods find widespread application in optimizing extensive continuous problems. By introducing continuous relaxation to the partitioning space, where each vertex's assignment to a partition block is associated with a probability, we effectively re-frame the partitioning problem to align with the GD optimization framework. Subsequently, after obtaining continuous partitioning solutions via GD, we convert them into discrete partitions by assigning each vertex to the block with the highest probability. It is essential to highlight that the initial solutions greatly influence the performance of GD, and poor initialization can result in getting stuck in local minima. Recognizing this, we

utilize GA to generate favorable starting points, enabling GD to escape local optima more effectively.

Alg. 2 presents our evolutionary differentiable algorithm. It starts with an initial population of partitioning solutions, evaluating their fitness scores. It leverages GA and GD to improve these solutions iteratively. GA generates offspring solutions by crossover and mutation in each generation, which are fine-tuned using GD. GD involves the transformation of solutions into a continuous space and their iterative refinement to search for better solutions. At predefined checkpoints, solutions are discretized and evaluated for fitness. After the GD epoch, GA updates the population according to the GD outcome. The algorithm reports the best-discovered solutions upon reaching the generation limit or the stagnation threshold.

4.1 Genetic Algorithm for Partitioning

In this subsection, we delve into the details of the GA component within Alg. 2, specifically focusing on the `GenOffspring` function in Line 3 and the `UpdatePopulation` function in Line 23. We utilize tournament selection in `GenOffspring` and a deterministic crowding technique in `UpdatePopulation` to address the diversity concerns discussed in Section 2.3.

Alg. 3 outlines the process of offspring generation by crossover and mutation for partitioning. The tournament selection function $\texttt{TournamentSel}(\mathbf{X},\ \text{scores}(\mathbf{X}),\ \texttt{N\_cp})$ in Line 3 selects individuals from a population to serve as parents for the next generation. Each tournament involves random selection of `N_cp` individuals from the population $\mathbf{X}$, and then outputs the one with the best fitness score. $\texttt{Crossover}(\boldsymbol{p}_1, \boldsymbol{p}_2, \texttt{N\_cp})$ in Line 8 represents `N_cp`-point crossover of $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, while $\texttt{Mutate}(\boldsymbol{p}, \texttt{p})$ in Lines 9 and 14 represents random mutation of $\boldsymbol{p}$ with probability `p`.

It is essential to consider the concept of "permutation symmetry" in partitioning, where rearranging vertices between two partition blocks maintains overall quality. For instance, in a 4-vertex graph, assigning vertices 1 and 2 to block 1 and vertices 3 and 4 to block 2 is equivalent to assigning vertices 1 and 2 to block 2 and vertices 3 and 4 to block 1. This property is important when performing crossover operations on two partitioning solutions. As a result, we align the solutions before crossover. The alignment process is outlined in Lines 5-7 in Alg. 3, where $||*||_1$ is the L1-norm and $\neg$ inverts the bits (=rearranges vertices between two partition blocks).

In `UpdatePopulation` (Line 23 in Alg. 2), we employ the deterministic crowding replacement technique. If an offspring $\boldsymbol{c}$ results from mutating its parent $\boldsymbol{p}$ and exhibits a superior fitness score, $\boldsymbol{p}$ is substituted with $\boldsymbol{c}$. When an offspring $\boldsymbol{c}$ is generated through the crossover of parents $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, $\boldsymbol{p}_1$ is replaced by $\boldsymbol{c}$ only if $\boldsymbol{p}_1$ is more similar to $\boldsymbol{c}$ (as measured by the L1-norm) than $\boldsymbol{p}_2$, and $\boldsymbol{c}$ possesses a better fitness score. Similarly, $\boldsymbol{p}_2$ is substituted with $\boldsymbol{c}$ if $\boldsymbol{p}_2$ exhibits greater similarity to $\boldsymbol{c}$ and $\boldsymbol{c}$ boasts a superior fitness score. An offspring will replace its parents if and only if the offspring is both more fit and genetically similar to the parent. Compared with greedy replacement methods, deterministic crowding prioritizes diversity and can lead to more robust solutions over time.

Tournament selection, crossover, mutation, and deterministic crowding replacement operations can seamlessly and efficiently be implemented using tensor operators in PyTorch [11]. This capability paves the way for harnessing the power of GPU acceleration in GA.

4.2 Differentiable Hypergraph Partitioning

Here, we provide an in-depth explanation of the gradient descent component within Alg. 2 (specifically, Lines 5 to 20).

4.2.1 Continuous Relaxation and Differentiable Costs. In k-way partitioning, each vertex within a hypergraph is assigned to one of the k distinct partition blocks. To make the search space continuous, we relax the categorical allocation of vertex v to a partition block using a softmax function over all partition blocks:

$$p_{v,i} = \frac{\exp\left(w_{v,i} \cdot T\right)}{\sum_{1 \le j \le k} \exp\left(w_{v,j} \cdot T\right)},$$

where T represents a hyper-parameter named temperature. Values $p_{v,i}, (v \in V, 1 \le i \le k)$, parameterized by $w_{v,i} \in \mathbb{R}$, can be interpreted as the probability of assigning the vertex v to partition block i. Consequently, the partitioning task reduces to learning a set of continuous variables $W_p = \{w_{v,i}\}$. Furthermore, to bridge the gap between continuous and discrete solutions during GD optimization, we progressively increase the temperature T to enforce convergence to a unique partition decision for each vertex.

The expectation of total vertex weight on a partition block i is:

$$W_i = \sum_{v \in V} p_{v,i} \cdot w_v. \tag{4}$$

Then the ε-balanced constraint (Eq. (3)) can be relaxed into a differentiable objective:

$$\sum_{1 \le i \le k} \text{ReLU}\left(W_i - \left(\frac{1}{k} + \varepsilon\right) W\right) + \text{ReLU}\left(\left(\frac{1}{k} - \varepsilon\right) W - W_i\right), \tag{5}$$

where $\text{ReLU}(x) = \max\{0, x\}$.

We also devise four differentiable proxies for the cut size, as depicted in Fig. 3 (b). These proxies are based on a matrix, denoted as P, which represents the probabilities of partition block assignments for all vertices within a hyperedge. The dimensions of matrix P are n by k, where n corresponds to the number of vertices in the hyperedge. In the example provided in Fig. 3 (b), the matrix P is as follows: $P = \begin{bmatrix} 0.1 & 0.9 \\ 0.3 & 0.7 \\ 0.2 & 0.8 \end{bmatrix}$. Each row in P represents the probability distribution of partition block assignment for a vertex. Our cut size proxies are primarily designed to assess the similarity among these probability distributions for all vertices within a hyperedge. Greater similarity corresponds to a smaller cut size. Our four proposed cut size proxies are as follows:

$$\texttt{ProdSum}(P) = \texttt{sum}(\texttt{prod}(P, dim = 0)) \tag{6}$$

$$\texttt{MeanEntropy}(P) = \texttt{sum}(\texttt{entropy}(\texttt{Mean}(P, dim = 0))) \tag{7}$$

$$\texttt{MeanMSE}(P) = \texttt{MSE}(P, \texttt{mean}(P, dim = 0)) \tag{8}$$

$$\texttt{MaxSum}(P) = \texttt{sum}(\texttt{max}(P, dim = 0)) \tag{9}$$

Here, we adopt notations from the PyTorch library. The operations denoted by `prod()`, `mean()`, `sum()`, `max()`, and `entropy()` can be realized using corresponding tensor operators in PyTorch bearing the same names. Additionally, `MSE` refers to the calculation of the mean squared error. Fig. 3 (b) provides an illustrative example for the calculation of our cut size proxies. A larger value of `ProdSum` corresponds to a smaller cut size, while smaller values of the other proxies also indicate a smaller cut size.

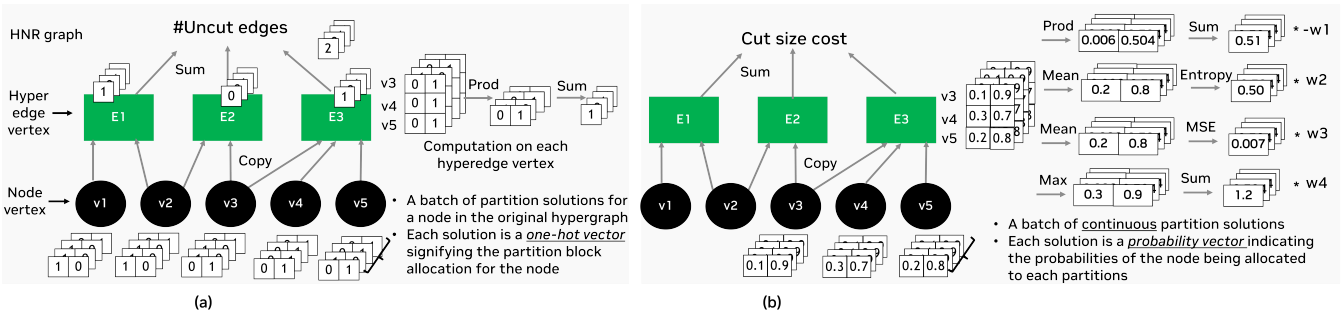


Figure 3: Batch cut size evaluation and optimization on the Hypergraph-Node Relationship graph. A batch of candidate assignments for each node is aggregated into the hyperedges to calculate objectives. (a) Batch cut size evaluation with discrete node to partition assignments. (b) Batch differentiable cut size optimization with soft probabilistic node to partition assignments. By analogy with deep graph learning, both cut-size evaluation and optimization can be accelerated with deep graph learning toolkits on GPUs.

The final cost function is defined as a weighted sum of the ϵ-balanced objective (Eq. (5)) and the four cut size proxies (Eqs. (6–9)), where the weights are hyper-parameters.

4.2.2 Interaction with GA. As illustrated in Line 8 of Alg. 2, our GD optimization starts with initial solutions derived by applying the `Relax()` function to the offspring generated through GA. The $\tilde{\boldsymbol{x}} = \texttt{Relax}(\boldsymbol{x})$ operator, transforming a binary solution $\boldsymbol{x}$ to a continuous counterpart $\tilde{\boldsymbol{x}}$, is defined as follows:

$$\tilde{\boldsymbol{x}}(i,j) = 0.5 - \alpha, \text{ if } \boldsymbol{x}(i,j) = 0,$$
$$\tilde{\boldsymbol{x}}(i,j) = 0.5 + \alpha, \text{ if } \boldsymbol{x}(i,j) = 1,$$

where $0 \leq \alpha \leq 0.5$ is a hyper-parameter.

Inversely, the $\boldsymbol{x} = \texttt{Discretize}(\tilde{\boldsymbol{x}})$ operator in Line 12 of Alg. 2 transforms a continuous solution $\tilde{\boldsymbol{x}}$ to a discrete solution $\boldsymbol{x}$ by applyting argmax to each row of $\tilde{\boldsymbol{x}}$. Additionally, the `MOD()` function in Line 11 represents the modulus operator.

4.3 Acceleration By Deep Graph Learning Toolkits

When executed on GPUs, our evolutionary differentiable optimization process can be significantly accelerated using deep graph learning toolkits, such as DGL [18]. This acceleration capitalizes on the analogy between hypergraph partitioning and deep graph learning and is facilitated by a specialized heterogeneous graph named the Hypergraph-Node Relationship (HNR) graph. As depicted in Fig. 3, the HNR graph comprises two distinct vertex types: node vertices, corresponding to nodes within the given hypergraph, and hyperedge vertices, representing the hyperedges in the original hypergraph. Edges are introduced between a node vertex and a hyperedge vertex only if the corresponding nodes and hyperedges are affiliated in the original hypergraph. We have identified two critical analogies between deep graph learning and hypergraph partitioning using the HNR graph framework:

(1) Cut size evaluation as forward propagation on HNR graph: In deep graph learning, forward propagation resembles a sequence of message passing steps, as illustrated in Fig. 1. This process computes messages for each vertex by aggregating information from neighboring vertices and applying feature transformations. In intermediate GNN layers, these messages update vertex features. In contrast, message aggregation occurs in the final GNN layers, potentially followed by additional transformations for loss computation. For cut size evaluation, each node vertex within the HNR graph is associated with a one-hot vector encoding the partition block allocation of the corresponding node in the original hypergraph, as shown in Fig. 3 (a). Subsequently, every hyperedge vertex combines the partitioning solution vectors from all incoming node vertices, forming the matrix P in Eq. (6). After applying the `ProdSum` operator to P, the result is 1 if all corresponding nodes within the original hypergraph's hyperedges are assigned to the same partition block; otherwise, it is 0. Ultimately, the total count of uncut hyperedges is obtained by summing the outcomes of the hyperedge vertices. If we consider the one-hot solution vectors on node vertices as features, then the computation of the uncut hyperedge count can be viewed as message passing within a single-layer GNN.
(2) Cut size optimization as backward propagation on HNR: Backward propagation in graph learning is the process of computing gradients with respect to the loss function and back-propagating these gradients through the layers of the GNN to update the model parameters. In the context of differentiable partitioning discussed in Section 4.2, we can relax the one-hot solution vectors on node vertices to continuous and treat them as trainable parameters, as depicted in Fig. 3 (b). By implementing differentiable operators as defined in Eqs. (5–9), the continuous partitioning solutions can be optimized using backward propagation techniques, akin to the standard GNN training process. This enables the refinement and learning of continuous partitioning solutions in an end-to-end manner.

Leveraging the insights outlined above, we effectively implement the cut size evaluation steps depicted in Lines 4 and 13 of Alg. 2, as well as the GD optimization iterations in Line 10, by harnessing deep graph learning toolkits, such as DGL.

Furthermore, to leverage the parallel computational capabilities of GPUs effectively, we have devised batch-based approaches for both cut-size evaluation and differentiable optimization, as illustrated in Fig. 3. Rather than processing a single partitioning solution at a time, we now handle a batch of solutions concurrently: all M GD trials in Line 5 of Alg. 2 are executed simultaneously. This innovative approach significantly enhances computational efficiency and harnesses the full power of parallel GPU processing.

5 EXPERIMENTAL VALIDATION

Our graph coarsening is implemented with the fast graph Laplacian linear solver LAMG [12] in Matlab, while the remaining components of MedPart are implemented in Python, leveraging the deep learning toolkits PyTorch and DGL. All experiments are conducted on a server with AMD EPYC 7742 processors and an NVIDIA A100 GPU with 80GB memory. In our evolutionary differentiable algorithm, we configure the number of generations I to be proportional to the logarithm of the vertex count N_l at the current graph coarsening level. We set the population size M as the maximum population size that our GPU memory can accommodate for a given test case, since all M gradient descent trials will be executed concurrently on the GPU. We set the number of GD steps S to 60. We employ the widely recognized gradient descent solver with momentum, Adam [1], in our differentiable optimization.

We compare MedPart with a leading hypergraph partitioner hMETIS [10] and a state-of-the-art partitioning solution refinement method SpecPart [7] on two sets of publicly-available benchmarks (ISPD98 VLSI Circuit Benchmark Suite [3] and Titan23 Suite [14]). The statistics of these benchmarks are summarized in Table 1 and Table 2, respectively. MedPart operates in two distinct modes: "from-scratch optimization" and "refinement." In the first mode, MedPart conducts optimization from scratch. In the refinement mode, initial partitioning solutions derived from running hMETIS five times, each with different random seeds (as provided in [6]), serve as the starting points for MedPart, which then enhances and refines these solutions. To avoid any possible confusion, we adopt these conventions: $MedPart$ and $MedPart_h$ represent the cutsizes of MedPart in "from-scratch optimization" mode and "refinement" mode, respectively. The cut sizes of SpecPart are provided in [7], which refines the solutions obtained from hMETIS [10] and/or KaHyPar [16].

5.1 Results on ISPD98 Benchmarks

Table 1 compares the cut sizes obtained by MedPart on the ISPD98 VLSI circuit benchmark with those from hMETIS, SpecPart, and the best-published results. Regardless whether ε is 2% or 10%, running MedPart once from scratch consistently outperforms running hMETIS five times with different random seeds. The best-published results for the ISPD98 VLSI circuit benchmark are well-established baselines. MedPart's results exhibit an average gap of approximately 5% for $\varepsilon = 2\%$ and 3.4% for $\varepsilon = 10\%$, demonstrating their optimality. Additionally, MedPart in its "refinement" mode yields results comparable to the state-of-the-art refinement method, SpecPart.

5.2 Results on Titan23 Benchmarks

Table 2 shows results on the Titan23 benchmarks. These are challenging due to many high-degree hyperedges. MedPart significantly outperforms hMETIS, with a 10% improvement for $\varepsilon = 2\%$ and an impressive 25% for $\varepsilon = 20\%$. In some cases, like sparcT1_core, MedPart even achieves solutions surpassing the best-published results by up to 30%. Generally, MedPart excels in smaller Titan23 test cases, mainly because GPU memory constraints necessitate a small population size for large hypergraphs, potentially causing premature convergence. To address this, we plan to explore multi-GPU optimization and mixed-precision gradient descent techniques in the future.

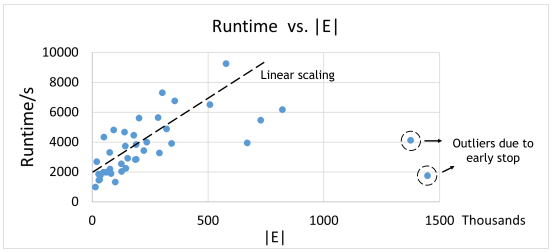


Figure 4: MedPart runtime on hypergraphs with different # of edges.

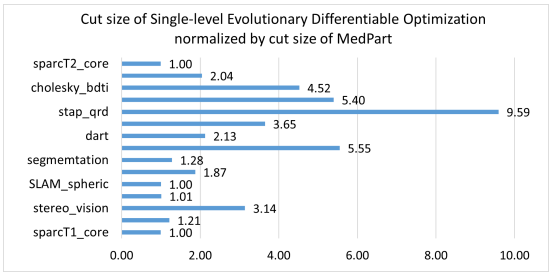


Figure 5: Impact of multi-level optimization on MedPart. The experiments are conducted on the top 15 benchmarks from the Titan23 benchmark suite, with ε set to 10%.

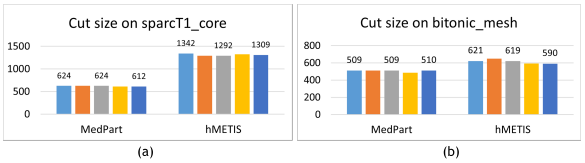


Figure 6: Cut sizes from MedPart and hMETIS on (a) sparcT1_core and (b) bitonic_mesh, each across 5 runs with different random seeds.

5.3 Runtime Scalability

Fig. 4 shows the runtime scalability of MedPart in "from-scratch optimization" mode. In general, MedPart runtime scales linearly with the number of hyperedges. The relatively shorter runtime observed in the two largest test cases can be attributed to the premature convergence caused by the use of a small population size. Premature convergence will trigger an early stop in Alg. 2. MedPart, primarily implemented in Python, has substantial room for runtime improvement. The evolutionary differentiable algorithm dominates the runtime of MedPart, while the graph coarsening takes only 11 seconds on the largest benchmark.

5.4 Impact of Multi-Level Optimization

We evaluate the impact of the multi-level optimization framework using the top 15 benchmarks from the Titan23 benchmark suite as representative examples. The results, illustrated in Fig. 5, reveal that exclusively running the evolutionary differentiable algorithm at the finest granularity level can result in cut sizes up to 9.6× larger than MedPart. This finding underscores the significant role played by our multi-level optimization framework.

5.5 Robustness of MedPart

Fig. 6 displays the cut sizes obtained from running MedPart (in from-scratch-optimization mode) and hMETIS on two sample test cases: sparcT1_core (with around 100K edges) and bitonic_mesh (with

Table 1: Statistics of ISPD98 VLSI circuit benchmark suite and cut sizes of different approaches. *SOTA* represents the best-published cut sizes summarized in [6]. *Spec* denotes the Specpart result presented in [7], which is obtained by employing SpecPart to enhance partitioning solutions generated by hMETIS and/or KaHyPar. $hMETIS_5$ signifies the best cut size obtained from running hMETIS five times with different random seeds (provided in [6]). *MedPart* and $MedPart_{h5}$ represent the cut sizes resulting from running MedPart once from scratch and using MedPart to refine the solutions from $hMETIS_5$, respectively. The best and the second-best results among all the methods are highlighted in red and blue, respectively.

	Statistics		$\varepsilon = 2\%$					$\varepsilon = 10\%$				
Benchmark	|V|	|E|	*SOTA*	*Spec*	$hMETIS_5$	*MedPart*	$MedPart_{h5}$	*SOTA*	*Spec*	$hMETIS_5$	*MedPart*	$MedPart_{h5}$
IBM01	12752	14111	200	202	213	202	205	166	171	190	166	166
IBM02	19601	19584	307	336	339	352	339	262	262	262	264	262
IBM03	23136	27401	951	959	972	955	957	950	952	960	955	954
IBM04	27507	31970	573	593	617	583	584	388	388	388	389	388
IBM05	29347	28446	1706	1720	1744	1748	1744	1645	1688	1733	1675	1668
IBM06	32498	34826	962	963	1037	1000	1012	728	733	760	788	760
IBM07	45926	48117	878	935	975	913	916	760	760	796	773	772
IBM08	51309	50513	1140	1146	1146	1158	1146	1120	1140	1145	1131	1135
IBM09	53395	60902	620	620	637	625	623	519	519	535	520	520
IBM10	69429	75196	1253	1318	1313	1327	1295	1244	1261	1284	1259	1257
IBM11	70558	81454	1051	1062	1114	1069	1067	763	764	782	774	765
IBM12	71076	77240	1919	1920	1982	1955	1949	1841	1842	1940	1914	1872
IBM13	84199	99666	831	848	871	850	850	655	693	721	697	696
IBM14	147605	152772	1842	1859	1967	1876	1884	1509	1768	1665	1639	1605
IBM15	161570	186608	2730	2741	2886	2896	2855	2135	2235	2262	2169	2166
IBM16	183484	190048	1827	1915	2095	1972	2095	1619	1619	1708	1645	1651
IBM17	185495	189581	2270	2354	2520	2336	2338	1989	1989	2300	2024	2028
IBM18	210613	201920	1521	1535	1587	1955	1587	1520	1537	1550	1829	1550
Avg gap to *SOTA*			0%	2.30%	6.20%	5.00%	3.70%	0%	2.10%	5.30%	3.40%	1.80%

Table 2: Statistics of Titan23 benchmark suite and cut sizes of different approaches. *SOTA* represents the best-published cut sizes. $Spec_{h20}$ denotes the SpectPart cut size presented in [7], which is obtained by employing SpecPart to enhance partitioning solutions generated by running hMETIS 20 times. $hMETIS_5$ signifies the best cut size obtained from running hMETIS five times (provided in [6]). *MedPart* and $MedPart_{h5}$ represent the cut sizes resulting from running MedPart once from scratch and refining the solutions from $hMETIS_5$, respectively. We utilize underlining to emphasize the cut sizes achieved by *MedPart* and $MedPart_{h5}$ that outperform the *SOTA*.

	Statistics		$\varepsilon = 2\%$					$\varepsilon = 20\%$				
Benchmark	|V|	|E|	*SOTA*	$Spec_{h20}$	$hMETIS_5$	*MedPart*	$MedPart_{h5}$	*SOTA*	$Spec_{h20}$	$hMETIS_5$	*MedPart*	$MedPart_{h5}$
sparcT1_core	91976	92827	977	1012	1073	1067	1073	903	903	1290	**624**	**624**
neuron	92290	125305	239	252	276	262	271	206	206	270	271	270
stereo_vision	94050	127085	169	180	213	176	184	91	91	143	93	93
des90	111221	139557	372	402	372	390	372	358	358	441	**349**	357
SLAM_spheric	113115	142408	1061	1061	1061	1061	1061	1061	1061	1061	1061	1061
cholesky_mc	113250	144948	285	285	301	**283**	**283**	285	345	667	**281**	**281**
segmemtation	138295	179051	118	126	183	137	**114**	78	78	141	78	78
bitonic_mesh	192064	235328	584	587	667	594	595	483	483	590	511	493
dart	202354	223301	788	807	849	805	810	540	540	603	593	549
openCV	217453	284108	481	510	635	751	635	481	518	554	617	554
stap_qrd	240240	290123	398	399	399	**386**	**386**	295	295	295	297	**287**
minres	261359	320540	215	215	215	295	215	189	189	189	**181**	189
cholesky_bdti	266422	342688	1156	1156	1161	1172	1161	947	947	1024	1148	1024
denoise	275638	356848	416	416	916	695	516	224	224	478	228	224
sparcT2_core	300109	302663	1227	1244	1410	1329	1319	1227	1245	1972	1148	**1081**
gsm_switch	493260	507821	1827	1827	5974	1722	**1714**	1407	1407	5352	1503	1541
mes_noc	547544	577664	634	634	699	1320	699	617	617	633	1141	633
LU230	574372	669477	3273	3273	4070	3452	3480	2677	2677	3276	2720	2741
LU_Network	635456	726999	525	525	550	597	550	524	524	528	567	528
sparcT1_chip2	820886	821274	899	899	1524	1169	1129	783	783	1029	877	889
directrf	931275	1374742	574	574	646	771	646	295	295	379	317	337
bitcoin_miner	1089284	1448151	1297	1297	1570	1562	1570	1225	1225	1255	1282	1255
Avg gap to SOTA			0%	1.9%	31.0%	19.1%	7.6%	0%	1.4%	44.0%	8.3%	2.60%

about 200K edges). In each case, both methods are executed with different random seeds five times. It can be found from the results that MedPart consistently produces robust outcomes, irrespective of the random seeds used.

6 CONCLUSIONS AND FUTURE DIRECTIONS

This study presents MedPart, a multi-level evolutionary differentiable hypergraph partitioning framework. Our experiments on public benchmarks consistently show MedPart outperforming the leading partitioner hMETIS, and achieving up to a 30% improvement in cut size compared to the best-published solutions for some benchmarks. We plan to apply our framework to other constraint-driven partitioning problems beyond traditional min-cut and ratio cut bipartitioning formulations.

REFERENCES

[1] Kingma DP Ba J Adam et al. 2014. A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* 1412 (2014).

[2] Sameer Agarwal, Kristin Branson, and Serge Belongie. 2006. Higher order learning with graphs. In *Proceedings of International Conference on Machine learning*. 17–24.

[3] Charles J Alpert. 1998. The ISPD98 circuit benchmark suite. In *Proceedings of International Symposium on Physical Design*. 80–85.

[4] Alain Bretto. 2013. Hypergraph theory - An introduction. *Mathematical Engineering* 1 (2013).

[5] Thang Nguyen Bui and Byung Ro Moon. 1996. Genetic algorithm and graph partitioning. *IEEE Trans. Comput.* 45, 7 (1996), 841–855.

[6] Ismail Bustany, Andrew Kahng, Yiannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. 2023. Partition solutions, scripts and SpecPart. https://github.com/TILOS-AI-Institute/HypergraphPartitioning

[7] Ismail Bustany, Andrew B Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. 2022. SpecPart: A supervised spectral framework for hypergraph partitioning solution improvement. In *Proceedings of International Conference on Computer-Aided Design*. 1–9.

[8] Umit V Catalyurek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 10, 7 (1999), 673–693.

[9] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. 2019. Graphzoom: A multi-level spectral approach for accurate and scalable graph embedding. *arXiv preprint arXiv:1910.02370* (2019).

[10] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1997. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of Design Automation Conference*. 526–529.

[11] Jonatan Kłosko, Mateusz Benecki, Grzegorz Wcisło, Jacek Dajda, and Wojciech Turek. 2022. High performance evolutionary computation with tensor-based acceleration. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 805–813.

[12] Oren E Livne and Achi Brandt. 2012. Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver. *SIAM Journal on Scientific Computing* 34, 4 (2012), B499–B522.

[13] Ole J Mengshoel and David E Goldberg. 1999. Probabilistic crowding: Deterministic crowding with probabilistic replacement. (1999).

[14] Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2013. Titan: Enabling large and complex benchmarks in academic CAD. In *Proceedings of International Conference on Field programmable Logic and Applications*. 1–8.

[15] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. 2019. Gap: Generalizable approximate graph partitioning framework. *arXiv preprint arXiv:1903.00614* (2019).

[16] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. K-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments*. 53–67.

[17] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. 2015. Comparative review of selection techniques in genetic algorithm. In *Proceedings of International Conference on Futuristic Trends on Computational Analysis and Knowledge Management*. 515–519.

[18] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *Proceedings of International Conference on Learning Representations Workshop on Representation Learning on Graphs and Manifolds*.