K-SpecPart: Supervised Embedding Algorithms and Cut Overlay for Improved Hypergraph Partitioning

Ismail Bustany, *Member, IEEE*, Andrew B. Kahng, *Fellow, IEEE*, Ioannis Koutis, *Member, IEEE*, Bodhisatta Pramanik, *Student Member, IEEE*, and Zhiang Wang, *Student Member, IEEE*

***Abstract*—State-of-the-art hypergraph partitioners follow the multilevel paradigm that constructs multiple levels of progressively coarser hypergraphs that are used to drive cut refinement on each level of the hierarchy. Multilevel partitioners are subject to two limitations: 1) hypergraph coarsening processes rely on local neighborhood structure without fully considering the global structure of the hypergraph and 2) refinement heuristics risk entrapment in local minima. In this article, we describe *K-SpecPart*, a supervised spectral framework for multiway partitioning that directly tackles these two limitations. *K-SpecPart* relies on the computation of generalized eigenvectors and supervised dimensionality reduction techniques to generate vertex embeddings. These are computational primitives that are not only fast, but embeddings also capture global structural properties of the hypergraph that are not explicitly considered by existing partitioners. *K-SpecPart* then converts the vertex embeddings into multiple partitioning solutions. Unlike multilevel partitioners that only consider the best solution, *K-SpecPart* introduces the idea of "ensembling" multiple solutions via a *cut-overlay clustering* technique that often enables the use of computationally demanding partitioning methods such as integer linear programming (ILP). Using the output of a standard partitioner as a supervision hint, *K-SpecPart* effectively combines the strengths of established multilevel partitioning techniques with the benefits of spectral graph theory and other combinatorial algorithms. *K-SpecPart* significantly extends ideas and algorithms that first appeared in our previous work on the bipartitioner *SpecPart* (Bustany et al., ICCAD 2022). Our experiments demonstrate the effectiveness of *K-SpecPart*. For bipartitioning, *K-SpecPart* produces solutions with up to ∼15% cutsize improvement over *SpecPart*. For multiway partitioning, *K-SpecPart* produces solutions with up to ∼20% cutsize improvement for smaller *K*, and maintains ∼2% improvement even when *K* is increased to 128, over leading partitioners *hMETIS* and *KaHyPar*.**

***Index Terms*—Hypergraph partitioning, partitioning algorithms, physical design (EDA), spectral partitioning.**

Manuscript received 3 June 2023; revised 2 October 2023; accepted 3 November 2023. Date of publication 13 November 2023; date of current version 21 March 2024. This work was supported in part by NSF under Grant CCF-2112665, Grant CCF-2039863, and Grant CCF-1813374; and in part by the Defense Advanced Research Projects Agency (DARPA) under Grant HR0011-18-2-0032. This article was recommended by Associate Editor I. H.-R. Jiang. *(Corresponding author: Zhiang Wang.)*

Ismail Bustany is with the Adaptive and Embedded Computing Group (AECG), Advanced Micro Devices Inc., Santa Clara, CA 95054 USA (e-mail: ismail.bustany@amd.com).

Andrew B. Kahng is with the Department of Computer Science and Engineering and the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: abk@ucsd.edu).

Ioannis Koutis is with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102 USA (e-mail: ioannis.koutis@njit.edu).

Bodhisatta Pramanik is with the Department of Electrical Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: bopramanik@ucsd.edu).

Zhiang Wang is with the Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093 USA (e-mail: zhw033@ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2023.3332268

I. Introduction

Balanced hypergraph partitioning is a well-studied, fundamental combinatorial optimization problem with multiple applications in EDA. The objective is to partition vertices of a hypergraph into a specified number of disjoint *blocks* such that each block has bounded size and the *cutsize*, i.e., the number of hyperedges spanning multiple blocks, is minimized [32].

Many hypergraph partitioners have been proposed over the past decades. State-of-the-art partitioners, including *MLPart* [27], *PaToH* [13], *KaHyPar* [32], and *hMETIS* [8], follow the multilevel paradigm [8]. Another thread of work that has been less successful in practice uses variants of unsupervised spectral clustering [35], [36], [38], [39]. All partitioning algorithms that are constrained by practical runtime constraints are inevitably bound to limitations, due to the computational complexity of the problem. However, different types of algorithms may have complementary strengths. For example, multilevel algorithms attempt to directly optimize the combinatorial objective, but they are bound by the local nature of their clustering heuristics and the entrapment in local minima that cannot be circumvented by their greedy refinement heuristics [15], [19]. On the other hand, spectral algorithms by design take into account the global properties of the hypergraph, albeit at the expense of optimizing surrogate objectives that may introduce significant approximation error. Researchers have proposed several approaches to incorporate the global properties of the hypergraph into multilevel algorithms. Notably, Shaydulin et al. [17] proposed a weighting scheme based on algebraic distances to take distant neighborhoods of vertices into account. *KaHyPar* [32] incorporates global information about the community structure into the coarsening process.

K-SpecPart is based on a novel general concept: a partitioning solution is viewed as a hint that can be used as input to supervised algorithms. The idea enables us to combine the strengths of established partitioning techniques with the benefits of supervised methods, and in particular spectral algorithms. Following are our main algorithmic and experimental contributions.

Supervised Spectral K-Way Embedding: Similar to *SpecPart* [34], *K-SpecPart* adapts the supervised spectral algorithm of [1] to generate a vertex embedding by solving a generalized eigenvalue problem. Spectral K-way partitioning usually involves either the computation of K eigenvectors of a single problem, or recursive bipartitioning. In our work, the availability of the K-way hint leads to a "one-versus-rest" approach that involves three fundamental steps: 1) we extract multiple two-way partitioning solutions from a K-way hint partitioning solution, and incorporate these as hint solutions into multiple instances of the generalized eigenvalue problem; 2) we subsequently solve the problem instances to generate multiple eigenvectors; and 3) the (column) eigenvectors from these instances are horizontally stacked to form a large-dimensional embedding. This particular way of generating a supervised K-way embedding is novel and may be of independent interest (Section IV).

Supervised Dimensionality Reduction: K-SpecPart generates embeddings that have larger dimensions than those in *SpecPart*, posing a computational bottleneck for subsequent steps. To mitigate this problem, we use linear discriminant analysis (LDA), a *supervised* dimensionality reduction technique, where we leverage again the K-way hint. This produces a low-dimensional embedding that respects (spatially) the K-way hint solution. Our experimental results show that this step not only reduces the runtime significantly ($\sim 10\times$) but also slightly improves ($\sim 1\%$) the cutsize, relative to using the large-dimensional embedding (Sections IV and VII-E).

Cut Distilling Trees and Tree Partitioning: Converting a vertex embedding to a K-way partitioning is an integral step of *K-SpecPart*. *SpecPart* introduced in this context a novel approach that uses the hypergraph and the vertex embedding to compute a family of weighted trees that in some sense *distill* the cut structure of the hypergraph. This effectively reduces the hypergraph partitioning problem to a K-way tree partitioning problem. Of course, $K > 2$ makes for a significantly more challenging problem, which we tackle in *K-SpecPart*. More specifically, we use recursive bipartitioning by extending the tree partitioning algorithm of [34] and augmenting it with a refinement step using the multiway Fiduccia–Mattheyses (FM) algorithm [15], [19]. This step is essentially an encapsulated use of an established partitioning algorithm, tapping again into the power of existing methods (Section V).

Cut Overlay and Optimization: K-SpecPart is an iterative algorithm that uses its partitioning solution from iteration i as a hint for subsequent iteration $i+1$. Standard multilevel partitioners compute multiple solutions and pick the best while discarding the rest. *K-SpecPart*, however, uses its entire pool of computed solutions in order to find a further improved partitioning solution, via a solution ensembling technique, *cut-overlay clustering* [34]. Specifically, we extract clusters by removing from the hypergraph the union of the hyperedges cut by any partitioning solution in the pool. The resulting clustered hypergraph typically comprises only hundreds of vertices, enabling integer linear program (ILP)-based hypergraph partitioning to efficiently identify the optimal partitioning of the set of clusters. The solution is then subsequently "lifted" to the original hypergraph and further refined with FM (Section VI).

TABLE I
NOTATION

Term	Description
$H(V,E)$	Hypergraph H with vertices V and hyperedges E
$H_c(V_c,E_c)$	Clustered hypergraph H_c where each vertex $v_c \in V_c$ corresponds to a group of vertices in $H(V,E)$
$G(V,E)$	Graph G with vertices V and edges E
$\tilde{G}$	Spectral sparsifier of G
$T(V,E_T)$	Tree T with vertices V and edges E_T
u, v	Vertices in V
e_{uv}	Edge connecting u and v
e_T	Edge of tree T
w_v, w_e	Weight of vertex v, or hyperedge e, respectively
K	Number of blocks in a partitioning solution
S	Partitioning solution, $S = \{V_0, V_1, ..., V_{K-1}\}$
ϵ	Allowed imbalance between blocks in S
$cut(S)$	Cut of S, $cut(S) = \{e \mid e \not\subseteq V_i$ for any $i\}$
$cutsize_H(S)$	Cutsize of S on H, i.e., sum of w_e, $e \in cut(S)$
X_{emb}, X	Vertex embeddings

TABLE II
PARAMETERS OF THE *K-SpecPart* FRAMEWORK

Parameter	**Description** (default setting)
m	Number of eigenvectors ($m = 2$)
δ	Number of best solutions ($\delta = 5$)
β	Number of iterations of *K-SpecPart* (β = 2)
ζ	Number of random cycles ($\zeta = 2$)
γ	Threshold of number of hyperedges ($\gamma = 500$)

Autotuning: We apply autotuning [54] on the hyperparameters of standard partitioners in order to generate a better hint for *K-SpecPart*. Our experiments show that this can further push the leaderboard for well-studied benchmarks (Section VII-H).

An Extensive Experimental Study: We validate *K-SpecPart* on multiple benchmark sets (*ISPD98 VLSI Circuit Benchmark Suite* [4] and *Titan23* [11]) with state-of-the-art partitioners (*hMETIS* [8] and *KaHyPar* [32]). Experimental results show that for some cases, *K-SpecPart* can improve cutsize by more than 50% over *hMETIS* and/or *KaHyPar* for bipartitioning and by more than 20% for multiway partitioning (Section VII-A). We also conduct a large ablation study in Sections VII-D–VII-H that shows how each of the individual components of our algorithm contributes in the overall result. Besides publishing all codes and scripts, we also publish a leaderboard with the best known partitioning solutions for all our benchmark instances in order to motivate future research [52].

K-SpecPart is built as an extension to *SpecPart* but significantly extends the ideas in [34]. This framework includes a variety of novel components that may seem challenging to comply with the strict runtime constraints of practical hypergraph partitioning. However, the choice of numerical solvers [24], [26] along with careful engineering enables a very efficient implementation, with further parallelization potential (Section VII-B). *K-SpecPart*'s capacity to include supervision information makes it potentially even more powerful in industrial pipelines. More importantly, its components are subject to individual improvement possibly leveraging machine learning and other optimization-based techniques (Section VIII). We thus believe that our work may eventually lead to a departure from the multilevel paradigm that has dominated the field for the past quarter-century.

II. Preliminaries

A. Hypergraph Partitioning Formulation

A hypergraph $H(V, E)$ consists of a set of vertices V and a set of hyperedges E where for each $e \in E$, we have $e \subseteq V$. We work with weighted hypergraphs, where each vertex $v \in V$ and each hyperedge $e \in E$ are associated with positive weights w_v and w_e, respectively. Given a hypergraph H, we define the following.

1) *K-Way Partition:* A collection $\mathcal{S} = \cup_i V_i$ of K vertex blocks $V_i \subseteq V$ such that $V_i \cap V_j = \emptyset$ and $\cup_{i=0}^{K-1} V_i = V$.
2) *Vertex Set Weight:* For $U \subseteq V$, $W_U = \sum_{v \in U} w_v$.
3) *ϵ-Balanced K-Way Partition S:* A K-way partition such that for all $V_i \subseteq \mathcal{S}$, we have $0 \leq (1/K) - \epsilon \leq W_{V_i}/W_V \leq (1/K) + \epsilon$.[1]
4) $cut_H(\mathcal{S}) = \{e | e \nsubseteq V_i$ for all $V_i \subseteq \mathcal{S}\}$.
5) $cutsize_H(\mathcal{S}) = \sum_{e \in cut_H(\mathcal{S})} w_e$.

The hypergraph partitioning problem seeks an ϵ-balanced K-way partition $\mathcal{S}$ that minimizes $\text{cutsize}_H(\mathcal{S})$.

B. Laplacians, Cuts, and Eigenvectors

Suppose $G = (V, E, w)$ is a weighted graph. The Laplacian matrix L_G of G is defined as follows: 1) $L(u, v) = -w_{e_{uv}}$ if $u \neq v$ and 2) $L(u, u) = \sum_{v \neq u} w_{e_{uv}}$. Let x be an indicator vector for the bipartitioning solution $S = \{V_0, V_1\}$ containing 1s in entries corresponding to V_1, and 0s everywhere else (V_0). Then, we have

$$x^T L x = \text{cutsize}_G(S). \tag{1}$$

There is a well-known connection between balanced graph bipartitioning and spectral methods. Let G_C be the complete unweighted graph on the vertex set V, i.e., for any distinct vertices $u \in V$ and $v \in V$, there exists an edge between u and v in G_C. Let L_{G_C} denote the Laplacian of G_C. Using (1), we can express the *ratio cut* $R(x)$ [55] as

$$R(x) \triangleq \frac{\text{cutsize}_G(S)}{|S| \cdot |V - S|} = \frac{x^T L x}{x^T L_{G_C} x}. \tag{2}$$

Minimizing $R(x)$ over 0-1 vectors x incentivizes a small $\text{cutsize}_G(S)$ with a simultaneous balance between $|S|$ and $|V - S|$, hence $R(x)$ can be viewed as a proxy for the balanced partitioning objective. We relax the minimization problem by looking for real-valued vectors x instead of 0-1 vectors x, while ensuring that the real-valued vectors x are orthogonal to the common null space of L and L_{G_C} [1]. A minimizer of (2) is given by the first nontrivial eigenvector of the problem $Lx = \lambda L_{G_C} x$ [1].

C. Spectral Embeddings and Partitioning

A *graph embedding* is a map of the vertices in V to points in an m-dimensional space. In particular, a *spectral embedding* can be computed by computing m eigenvectors $X \in \mathbb{R}^{|V| \times m}$ of a matrix pair (L_G, B), in a generalized eigenvalue problem of the form

$$L_G x = \lambda B x \tag{3}$$

where L_G is a graph Laplacian, and B is a positive semi-definite matrix. An embedding can be converted into a partitioning by clustering the points in this m-dimensional space.

Spectral embeddings have been used for hypergraph partitioning. In this context, the hypergraph H is first transformed to a graph G, and then the spectral embedding is computed using L_G. For example, the eigenvalue problem solved in [35] sets $B = D_w$, where D_w is the diagonal matrix containing positive vertex weights. In this article, we solve more general problems where B is a graph Laplacian. This enables us to handle zero vertex weights as required in practice, and to encode in a natural "graphical" way prior supervision information into the matrix B.[2]

D. Supervised Dimensionality Reduction (LDA)

LDA is a supervised algorithm for dimensionality reduction [29]. The inputs for LDA are: 1) a matrix $X^{N \times M}$ where the ith row x_i is a point in M-dimensional space and 2) a class label from $\{0, \ldots, K-1\}$ for each point x_i. Then, the objective of LDA is to transform $X^{N \times M}$ into $\tilde{X}^{N \times m}$, where m ($m < M$) is the target dimension so that the clusters of points corresponding to different classes are best separated in the m-dimensional space, under the simplifying assumption that the classes are normally distributed and class covariances are equal [12]. From an algorithmic point of view, LDA calculates in $O(NM^2)$ time two matrices $S_B^{M \times M}$ and $S_W^{M \times M}$ capturing between-class-variance and the within-class-variance, respectively. Then, it calculates a matrix $P^{M \times m}$ containing the m largest eigenvectors of $S_W^{-1} S_B$, and lets $\tilde{X} = XP$. Because in our context m is a small constant, LDA can be computed very efficiently.

E. ILP for Hypergraph Partitioning

Hypergraph partitioning can be solved optimally by casting the problem as an ILP [31]. To write balanced hypergraph partitioning as an ILP, for each block V_i we introduce integer $\{0, 1\}$ variables, $x_{v,i}$ for each vertex v, and $y_{e,i}$ for each hyperedge e. Setting $x_{v,i} = 1$ signifies that vertex v is in block V_i, and setting $y_{e,i} = 1$ signifies that all vertices in hyperedge e are in block V_i. We then define the following constraints for each $0 \leq i < K$.

1) $\sum_{j=0}^{K-1} x_{v,j} = 1$, for all $v \in V$.
2) $y_{e,i} \leq x_{v,i}$ for all $e \in E$, and $v \in e$.
3) $([1/K] - \epsilon) \leq \sum_{v \in V_i} w_v x_{v,i} \leq ([1/K] + \epsilon)W$, where $W = \sum_{v \in V} w_v$.

[1]The imbalance definition in *K-SpecPart* differs from that in *KaHyPar*. However, *K-SpecPart* can also accept an argument *eps*, exactly in the same way as *KaHyPar* [32]. Specifically, we set an $eps = K \times \epsilon$ for *KaHyPar* in all our experiments (Section VII-A).

[2]*Technical Remark:* In this work, we assume that G is connected. Then, the problem in (3) is well defined even if B does not correspond to a connected graph, because L_G's null space is a subspace of that of B [16]. The assumption that G is connected holds for practical instances. In the more general case, we can work by embedding each connected component of G separately and work with a larger embedding. The details are omitted.

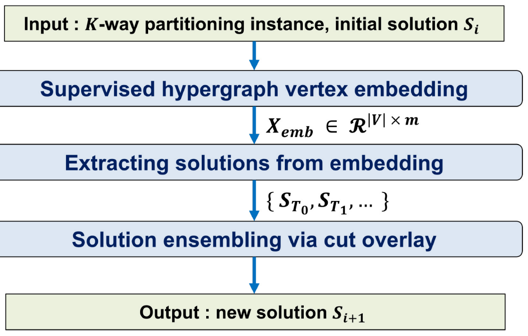

Input : K-way partitioning instance, initial solution S_i
Supervised hypergraph vertex embedding
$X_{emb} \in \mathcal{R}^{|V| \times m}$
Extracting solutions from embedding
$\{S_{T_0}, S_{T_1}, \dots\}$
Solution ensembling via cut overlay
Output : new solution S_{i+1}

Fig. 1. One iteration of *K-SpecPart*. The three modules are expanded in Figs. 3–5, and further described in Sections IV–VI. The initial partitioning solution S_0 can be obtained from any partitioner, but in this work, we use *hMETIS* and *KaHyPar*. During its iterations, *K-SpecPart* collects its outputs $\{S_0, S_1, \dots, S_\beta\}$. *K-SpecPart* then applies the ensembling module on $\{S_0, S_1, \dots, S_\beta\}$ to compute its final output S_{out}.

The objective is to maximize the total weight of the hyperedges that are not cut, i.e.,

$$\text{maximize} \sum_{e \in E} \sum_{i=0}^{K-1} w_e y_{e,i}.$$

III. K-SpecPart Framework

We view *K-SpecPart* as an instantiation of a general framework for improving a given solution to a partitioning instance. The framework involves three modules: 1) vertex embedding module; 2) solution extraction module; and 3) ensembling module, as illustrated in Fig. 1. The details are given in Algorithm 1.

The input is the hypergraph H and a partitioning solution S_i in the form of block labels $\{0, \dots, K-1\}$ for the vertices. The *vertex embedding module* computes a map of each hypergraph vertex to a point in a low-dimensional space. The embedding is computed by a *supervised* algorithm, using S_i as the supervision input (Algorithm 1, lines 7–19). The intuition is that the vertex embedding is incentivized to conform with S_i, thus staying in the "vicinity" of S_i, but simultaneously to respect the global structure of the hypergraph, thus having the potential to improve S_i. The *solution extraction module* computes a pool of different partitioning solutions $\{S_{i,1} \dots, S_{i,\delta}\}$ (lines 20–22). These are then sent to the *ensembling module*, which uses our cut-overlay method to convert the given solutions to a small instance of the K-way partition which can be solved much more reliably by more expensive partitioning algorithms (line 23). The solution to this small problem instance is then lifted (i.e., mapping back to the original hypergraph H) and further refined to the output S_{i+1}. The remainder of this article presents our implementations of these three modules.

Algorithm 1: *K-SpecPart* Framework

```
Input: Hypergraph $H(V, E)$, Number of blocks $K$,
  Initial partitioning solution $S_{init}$,
  Number of supervision iterations $\beta$,
  Allowed imbalance between blocks $\epsilon$
Output: Improved partitioning solution $S_{out}$
1  Construct the clique expansion graph $G$ of $H$ and the Laplacian
     matrix $L_G$ of $G$ (Section IV)
2  Construct the weight-balance graph $G_w$ of $H$ and the Laplacian
     matrix $L_{G_w}$ of $G_w$ (Section IV)
3  Initialize the empty candidate solution list $\{S_{candidate}\}$
4  $\{S_{candidate}\}.push\_back(S_{init})$
5  $S_0 = S_{init}$
6  for $i = 0; i < \beta; i++$ do
       /* Supervised hypergraph vertex embedding
          (Section IV)                                        */
7      if $K = 2$ then
8          Construct the hint graph $G_h$ based on $S_i$ and the Laplacian
             matrix $L_{G_{h_i}}$ of $G_h$ (Section IV-A)
9          Solve the generalized eigenvalue problem
             $L_G x = \lambda(L_{G_w} + L_{G_{h_i}})x$ to obtain the first $m$ nontrivial
             eigenvectors $X_{emb} \in \mathcal{R}^{|V| \times m}$
10     end
11     else
12         Decompose the $K$-way partitioning solution $S_i$ into $K$
             bipartitioning (2-way) solutions $\{S_{b_0}, \dots, S_{b_{K-1}}\}$
             (Section IV-B)
13         for $j = 0; j < K; j++$ do
14             Construct the hint graph $G_{h_j}$ based on $S_{b_j}$ and the
                 Laplacian matrix $L_{G_{h_j}}$ of $G_{h_j}$ (Section IV-B)
15             Solve the generalized eigenvalue problem
                 $L_{G_j} x = \lambda(L_{G_w} + L_{G_{h_j}})x$ to obtain the first $m$ nontrivial
                 eigenvectors $X_j^m \in \mathcal{R}^{|V| \times m}$
16         end
17         $X_{emb} = [X_0^m | X_1^m | \cdots | X_{K-1}^m]$
18         Perform linear discriminant analysis (LDA) to generate the
             vertex embedding $X_{emb} \in \mathcal{R}^{|V| \times m}$.
19     end
       /* Extracting solutions from embedding
          (Section V)                                         */
20     Construct a family of trees $\{T_0, T_1, \dots\}$ leveraging the vertex
         embedding $X_{emb} \in \mathcal{R}^{|V| \times m}$
21     Generate hypergraph partitioning solutions $\{S_{T_0}, S_{T_1}, \dots\}$ through
         cut distilling and tree partitioning
22     Refine $\{S_{T_0}, S_{T_1}, \dots\}$ using multi-way FM
       /* Solution ensembling via cut overlay
          (Section VI)                                        */
23     $S_{i+1} \leftarrow$ perform cut-overlay clustering and ILP-based
         partitioning on the top $\delta$ solutions from $\{S_{T_0}, S_{T_1}, \dots\}$
24     $\{S_{candidate}\}.push\_back(S_{i+1})$
25 end
   /* Solution ensembling via cut overlay                     */
26 $S_{out} \leftarrow$ perform cut-overlay clustering and ILP-based partitioning on
     solutions $\{S_{candidate}\}$
27 Refine $S_{out}$ using multi-way FM
28 return $S_{out}$
```

IV. Supervised Vertex Embedding

The supervised vertex embedding module takes as inputs the hypergraph $H(V, E)$ and a K-way partitioning solution S_{hint}, and outputs an m-dimensional embedding $X^{|V| \times m}$.

In *K-SpecPart*, we use a spectral embedding algorithm that encodes into a generalized eigenvalue problem the supervision information S_{hint}. Fig. 2 illustrates how the inclusion of the hint incentivizes the computation of an embedding that in general respects (spatially) the given solution S_{hint}, but also identifies vertices of contention where improving the solution may be possible.

A. Embedding From Two-Way Hint

The embedding algorithm for two-way hints is identical to that used in *SpecPart* [34], which is shown in Fig. 3. It consists of two major steps.

1) *Graph Construction:* We define the three graphs used by the embedding algorithm: a) clique expansion graph G; b) weight-balance graph G_w; and c) hint graph G_h. The

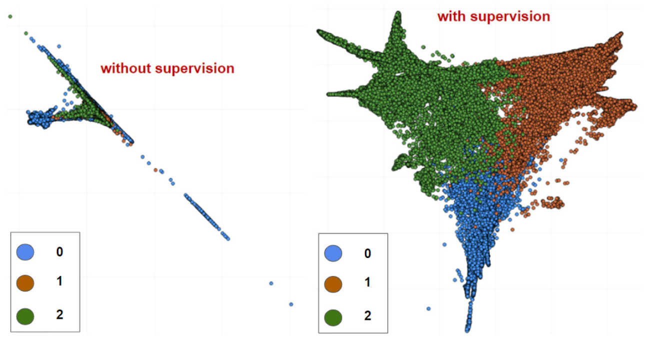


Fig. 2. Vertex embeddings of the ISPD IBM14 benchmark. Point colors indicate block membership in a 3-way partitioning solution with $\epsilon = 5\%$ computed by *hMETIS*. The embedding on the right uses as a hint the same *hMETIS* solution, while the embedding on the left is unsupervised.

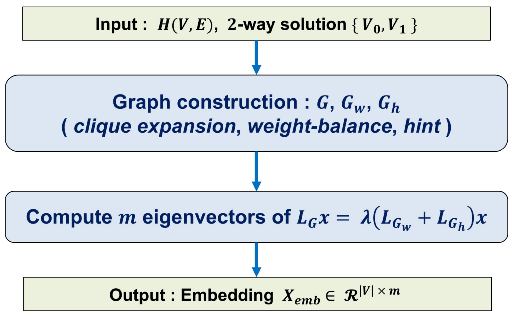

Input : $H(V,E)$, 2-way solution $\{V_0, V_1\}$
Graph construction : G, G_w, G_h
(clique expansion, weight-balance, hint)
Compute m eigenvectors of $L_G x = \lambda(L_{G_w} + L_{G_h})x$
Output : Embedding $X_{emb} \in \mathcal{R}^{|V| \times m}$

Fig. 3. Supervised embedding with a two-way hint.

clique expansion graph G is a superposition of weighted cliques. The clique corresponding to the hyperedge $e \in E$ has the same vertices as e and edge weights $[1/|e| - 1]$. The *weight-balance graph* G_w is a complete weighted graph used to capture arbitrary vertex weights and incentivize balanced cuts. G_w has the same vertices as hypergraph H, and edges of weight $w_u \cdot w_v$ between any two vertices u and v. The *hint graph* G_h is a complete unweighted bipartite graph on the two vertex sets V_0 and V_1 defined by the two-way hint solution S_b.

2) *Generalized Eigenvalue Problem and Embedding:* Given a two-way partitioning solution S_{hint}, we solve the generalized eigenvalue problem $L_G x = \lambda B x$ where $B = L_{G_w} + L_{G_h}$, and compute the first m nontrivial eigenvectors $X \in \mathbb{R}^{|V| \times m}$ whose rows provide the vertex embedding. We solve $L_G x = \lambda B x$ using LOBPCG, an iterative preconditioned eigensolver.

B. Embedding From Multiway Hint

The flow for generating an m-dimensional embedding from a K-way hint is shown in Fig. 4. The steps are described in the following paragraphs.

Embedding by Concatenation: In the K-way case where $K > 2$, the solution hint S_{hint} corresponds to a K-way partitioning solution $\{V_0, \ldots, V_{K-1}\}$. We then extract K different bipartitions, S_{b_i}, for $i = 0, \ldots, K-1$, where

$$S_{b_j} = \left\{ V_j, \bigcup_{i=0, i \neq j}^{K-1} V_i \right\}.$$

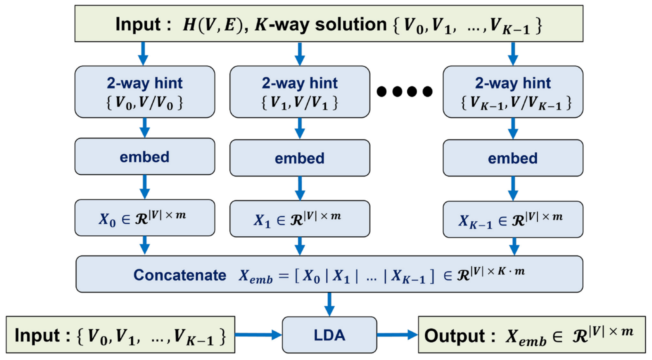

Input : $H(V,E)$, K-way solution $\{V_0, V_1, \ldots, V_{K-1}\}$
2-way hint $\{V_0, V/V_0\}$
2-way hint $\{V_1, V/V_1\}$
2-way hint $\{V_{K-1}, V/V_{K-1}\}$
embed
embed
embed
$X_0 \in \mathcal{R}^{|V| \times m}$
$X_1 \in \mathcal{R}^{|V| \times m}$
$X_{K-1} \in \mathcal{R}^{|V| \times m}$
Concatenate $X_{emb} = [X_0 | X_1 | \ldots | X_{K-1}] \in \mathcal{R}^{|V| \times K \cdot m}$
Input : $\{V_0, V_1, \ldots, V_{K-1}\}$
LDA
Output : $X_{emb} \in \mathcal{R}^{|V| \times m}$

Fig. 4. Vertex embedding generation process for a given multiway ($K > 2$) hint, using the two-way embedding subroutine from Section IV-A.

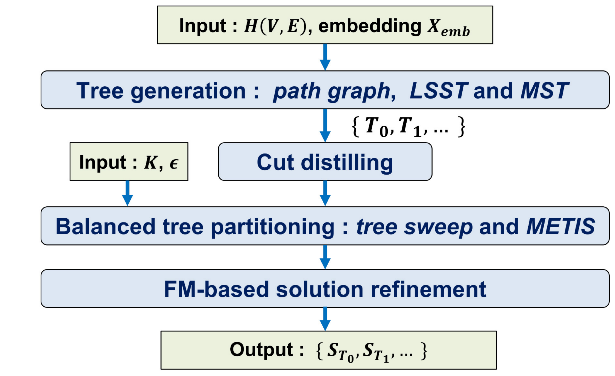

Input : $H(V,E)$, embedding X_{emb}
Tree generation : path graph, LSST and MST
$\{T_0, T_1, \ldots\}$
Input : K, ϵ
Cut distilling
Balanced tree partitioning : tree sweep and METIS
FM-based solution refinement
Output : $\{S_{T_0}, S_{T_1}, \ldots\}$

Fig. 5. Flow of extracting solutions from embeddings.

For each S_{b_j}, we solve an instance of the generalized problem we set up in Section IV-A. This generates K different embeddings $X_j \in \mathbb{R}^{|V| \times m}$. We then concatenate these K embeddings horizontally to get our final embedding $X_{\text{emb}} \in \mathbb{R}^{|V| \times K \cdot m}$, i.e.,

$$X_{\text{emb}} = \left[X_0 | X_1 | \ldots | X_{K-1}\right], \text{ where } X_j \in \mathbb{R}^{|V| \times m}.$$

Supervised Dimensionality Reduction: Note that the above embedding X_{emb} has dimension $K \cdot m$. We then apply on X_{emb} a supervised dimensionality reduction algorithm, specifically LDA (see Section VII-E), as illustrated in Fig. 4. We use LDA primarily to reduce the runtime of subsequent steps, but also because this second application of supervision has the potential to increase the quality of the embedding.

Besides X_{emb}, LDA takes as input a target dimension, and class labels for the points in X_{emb}. We choose m as the target dimension. We assign label i to vertex v if $V \in V_i$. For the computation, we use a Julia-based LDA implementation from the *MultivariateStats.jl* package [57].

V. Extracting Solutions From Embeddings

The inputs of the solution extraction module are the hypergraph H, number of blocks K, balance constraint ϵ, and an embedding $X_{\text{emb}} \in \mathcal{R}^{|V| \times m}$, and the output is a pool of solutions $\{S_0, \ldots, S_{\delta-1}\}$. The main idea of the algorithm is to use the embedding to reduce the K-way hypergraph partitioning problem to multiple K-way balanced partitioning problems on trees whose edge weights "summarize" the underlying cuts of the hypergraph. The steps of the algorithm are shown in Fig. 5 and described in Sections V-C and V-B.

A. Tree Generation

In our algorithm, each S_i in the output comes from a tree that spans the set of vertices V. Here, we define the types of trees we use.

Path Graph: We first define a path graph on the vertices V, which appears in the proofs of Cheeger inequalities for bipartitioning [41], [42]. Let X_{emb_i} be the ith column of X. We sort the values in X_{emb_i} and let $o(j)$ be vertex at the jth position of the sorted X_{emb_i}. Then, we define the path graph on V to be $v_{o(1)}, v_{o(2)}, \ldots, v_{o(|V|)}$.

Clique Expansion Spanning Tree: The path graph is likely not a spanning tree of the clique expansion graph G. To take connectivity directly into account, we work with a weighted graph that reflects both the connectivity of H and the global information contained in the embedding, adapting an idea that has been used in work on K-way Cheeger inequalities [28]. Concretely, we form a graph $\hat{G}$ by replacing every hyperedge e of H with a sum of ζ cycles (as also done in Section IV-A). Suppose that $Y \in \mathbb{R}^{|V|\times d}$ is an embedding matrix. We denote by Y_u the row of Y containing the embedding of vertex u. We construct the weighted graph $\hat{G}_Y$ by setting the weight of each edge $e_{uv} \in \hat{G}$ to $||Y_u - Y_v||_2$, i.e., equal to the Euclidean distance between the two vertices in the embedding. Using $\hat{G}_Y$ we build two spanning trees.

Low Stretch Spanning Tree (LSST): A desired property for a spanning tree $\hat{T}$ of $\hat{G}_Y$ is to preserve the embedding information contained in $\hat{G}$ as faithfully as possible. Thus, we let $\hat{T}$ be an LSST of $\hat{G}$, which by definition means that the weight $w_{e_{uv}}$ of each edge in $\hat{G}$ is approximated *on average*, and up to a small function $f(|V|)$, by the distance between the nodes u and v in $\hat{T}$ [2]. We compute the LSST using the AKPW algorithm of Alon et al. [2]. The output of the AKPW algorithm depends on the vertex ordering of its input. To make it invariant to the vertex ordering in the original hypergraph H, we relabel the vertices of $\hat{G}_Y$ using the order induced by sorting the smallest nontrivial eigenvector computed earlier. Empirically, this order has the advantage of producing LSSTs that contain slightly better cutsizes.

MST: A graph can contain multiple different LSSTs, with each of them approximating to different degrees the weight $w_{e_{uv}}$ for any given e_{uv}. It is known that the AKPW algorithm is suboptimal with respect to the approximation factor $f(|V|)$; more sophisticated algorithms exist but they are far from practical. Hence, we also apply Kruskal's algorithm [3] to compute a Minimum Spanning Tree of $\hat{G}$, which serves as an easy-to-compute proxy to an LSST. The MST can potentially have better or complementary distance-preserving properties relative to the tree computed by the AKPW algorithm.

In summary, we compute m path graphs, and also generate the LSSTs and MSTs by letting Y range over each subset of columns of X_{emb}. This produces a family of $t = 2(2^m - 1) + m$ trees.

B. Cut Distilling

We reweight *each* tree T in the given family of trees to distill the cut structure of H over T, in the following sense: 1) for a given tree $T = (V, E_T)$, observe that the removal of an edge e_T of T yields a partitioning S_{e_T} of V and thus of the original hypergraph H and 2) we reweight each edge $e_T \in E_T$ with the corresponding cutsize$_H(S_{e_T})$.

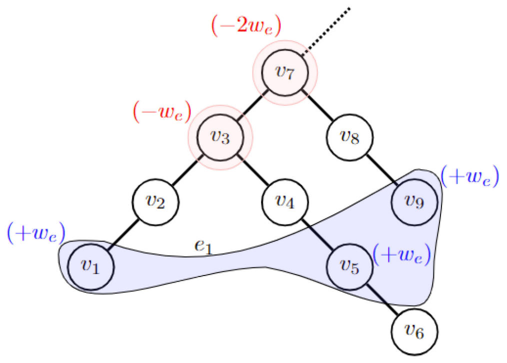


Fig. 6. Hyperedge, junctions, and their numerical labels. The vertices highlighted in red are the junction vertices.

With this choice of weights, we have cutSize$_H(S) \leq$ cutSize$_T(S)$, and owing to the reasoning behind the construction of T, cutSize$_T(S)$ provides a proxy for cutSize$_H(S)$.

Computing edge weights on T can be done in $O(\sum_e |e|)$ time, via an algorithm involving the computation of least common ancestors (LCAs) on T, in combination with dynamic programming on T [9]. We provide pseudocode in Algorithm 2 and give a fast implementation in [52]. We illustrate the idea using the example in Fig. 6.

We consider T to be rooted at an arbitrary vertex. In the example of Fig. 6, consider hyperedge $e = \{v_1, v_5, v_9\}$. The LCA of its vertices is v_7. Then, the weight of e should be accounted for the set $C_e \subset E_T$ of all tree edges that are ancestors of $\{v_1, v_5, v_9\}$ and descendants of v_7. We do this as follows (Algorithm 2, lines 2–13).

1) We compute a set of *junction* vertices that are LCAs of $\{v_1, v_5\}$ and $\{v_1, v_5, v_9\}$.
2) We then "label" these junctions with $-w_e$, where w_e is the weight of e. More generally, for a hyperedge $e = \{v_{i_1}, \ldots, v_{i_p}\}$ ordered according to the post-order depth-first search traversal on T, we calculate the LCAs for the $p-1$ sets $\{v_{i_1}, \ldots, v_{i_j}\}$ for $j = 2, \ldots, p$, and the junctions are labeled with appropriate negative multiples of w_e. We also label the vertices in e with w_e.
3) All other vertices are labeled with 0.

Consider then an arbitrary edge e_T of the tree, and compute the sum-below-e_T, i.e., the sum of the labels of vertices that are *descendants* of e_T. This will be w_e on all edges of C_e and 0 otherwise, thus correctly accounting for the hyperedge e on the intended set of edges C_e (Algorithm 2, lines 14–16). In order to compute the correct total counts of cut hyperedges on all tree edges, we iterate over hyperedges, compute their junction vertices, and aggregate the associated labels. Then, for any tree edge e_T, the sum-below-e_T will equal cutsize$_H(S_{e_T})$. These sums can be computed in $O(|V|)$ time, via dynamic programming on T.

C. Tree Partitioning

We use a linear "tree-sweep" method and *METIS* to partition the trees. In our studies, we have observed that only using *METIS* as the tree partitioner results in an average of 3%,

Algorithm 2: Cut Distilling

```
Input: Hypergraph $H(V, E)$, Tree $T(V, E_T)$
Output: Tree $T$ with updated edge weights
1  Select an arbitrary vertex $v_{root}$ from $V$ and root $T$ at $v_{root}$
2  Perform a post-order depth-first search traversal on $T$ and
     store the sequence of visited vertices in visited_sequence
   /* Label each vertex in T based on the
      hyperedge weight $w_e$ for $e \in E$            */
3  $cuts\_delta[v] \leftarrow 0$ for all $v$ in $V$
4  for each $e$ in $E$ do
5      $cuts\_delta[v] = cuts\_delta[v] + w_e$ for all $v$ in $e$
6      $\{v_{i0}, \dots, v_{i(|e|-1)}\} \leftarrow$ arrange the vertices of $e$ according
         to their positions in visited_sequence
7      $v_{LCA} \leftarrow v_{i0}$
8      for $j = 1; j < |e|; j++$ do
9          $v_{LCA} \leftarrow$ identify the least common ancestor (LCA)
             for $v_{LCA}$ and $v_{ij}$ in $T$
10         $cuts\_delta[v_{LCA}] = cuts\_delta[v_{LCA}] - w_e$
11     end
12     $cuts\_delta[v_{LCA}] = cuts\_delta[v_{LCA}] - w_e$
13 end
   /* Reweight edges of T                             */
14 for each $e_T$ in $E_T$ do
15     $w_{e_T} \leftarrow$ compute the sum-below-$e_T$ (i.e., the sum of the
         labels cuts_delta of vertices that are descendants of $e_T$)
         in the post-order depth-first search ordering
16 end
17 return $T$ with updated edge weights
```

4%, and 3% deterioration in cutsize for $K = 2$, 3, and 4, respectively.

$K = 2$: Given a cut-distilling tree T, and referring back to Fig. 6, an application of dynamic programming can compute the total weight of the vertices that lie below e_T on T. We can thus compute the value for the balanced cut objective for S_{e_T} and pick the S_{e_T} that minimizes the objective among the $n-1$ cuts suggested by the tree. This tree-sweep algorithm generates a good-quality two-way partitioning solution from the tree. Additionally, we use *METIS* [7] to solve a balanced two-way partitioning problem on the edge-weighted tree, with the original vertex weights from H. In some cases, this improves the solution.

$K > 2$: Similar to $K = 2$, we use two algorithms to compute two potentially different K-way partitioning solutions of the tree. The first algorithm is *METIS* [7]. The second algorithm extends the two-way cut partitioning of the tree to K-way partitioning. To this end, we apply the two-way algorithm recursively, for $K - 1$ levels. We use a similar idea as the VILE ("very illegal") method [27] to generate an imbalanced partitioning solution and then refine the solution with the FM algorithm. Specifically, while computing the ith level bipartitioning solution $S^i_{e_T}$ on the tree e_T, the balance constraint for block V_{i0} in the bipartitioning solution $S(V_{i0}, V_{i1})$ is: $([1/K] - \epsilon) \cdot W \leq \sum_{v \in V_{i0}} w_v \leq ([1/K] + \epsilon) \cdot W$.

After obtaining the ith bipartitioning solution $S(V_{i0}, V_{i1})$, we mark all the vertices in V_{i0} as fixed vertices and set their weights to zero. We then proceed with the $(i + 1)$th level bipartitioning solution $S^{i+1}_{e_T}$ on the tree e_T.

D. Refinement on the Hypergraph

The previous step solves balanced partitioning on trees that share the same vertex set V with H. Note that the number of solutions will be larger than the number of trees t, because we apply different partitioning algorithms to each tree. These solutions are then transferred to H, and each is further refined using the FM algorithm [15] on the entire hypergraph H. In particular, we use the FM implementation in [53].

VI. Solution Ensembling Via Cut Overlay

The input of this module is the given K-way partitioning instance and a pool of partitioning solutions. We then perform the following steps.

Cut-Overlay Clustering: In contrast to the widely used V-cycle refinement [8], *cut-overlay clustering* adopts a similar idea as the *Edge-Frequency Multi-Recombine* approach [5]. V-cycle refinement takes the (sole) best solution obtained from the multilevel partitioning algorithm and improves it using multiphase refinement repeatedly; in contrast, *cut-overlay clustering* tries to reveal the high-quality cut structure by combining a pool of partitioning solutions. We first select the δ best solutions. Let $E_1, \dots, E_\delta \subset E$ be the sets of hyperedges cut in the δ solutions. We remove the union of these sets from H to yield a number of connected clusters. Then, we perform a cluster contraction process that is standard in multilevel partitioners, to give rise to a clustered hypergraph $H_c(V_c, E_c)$. By construction, E_c consists of $E_1 \cup \cdots \cup E_\delta$ and hence is guaranteed to contain a solution which is *at least as good* as the best among the cuts E_i.

ILP-Based Partitioning: The coarse hypergraph (H_c) obtained from cut-overlay clustering usually has a few hundreds of vertices and hyperedges (including with the default setting $\delta = 5$). While even this small size would be expected to be prohibitive for applying an exact optimization algorithm, somewhat surprisingly, an ILP formulation can frequently solve the problem optimally. In most cases, our ILP produces a solution better than any of the δ candidate solutions. We solve the ILP with the CPLEX solver [47]. We have found that the open-source OR-Tools package [56] is significantly slower. In our current implementation, we include a parameter γ: in the case when the number of hyperedges in H_c is larger than γ, we run *hMETIS* on H_c. This step generates a K-way solution S' on H_c.

Lifting and Refinement: The solution S' from the previous step is lifted to H, with the standard lifting process that multilevel partitioners use. Finally, we apply FM refinement on H to obtain the final solution S. Here, we again use the FM implementation from [53].

VII. Experimental Validation

The *K-SpecPart* framework is implemented in Julia. We use *CPLEX* [47] and *LOBPCG* [23] as our ILP solver (we provide an OR-Tools-based implementation) and eigenvalue solver, respectively. We run all experiments on a server with an Intel Xeon E5-2650L, 1.70-GHz CPU, and 256-GB memory. We have compared our framework with two state-of-the-art hypergraph partitioners (*hMETIS* [8] and *KaHyPar* [32]) on

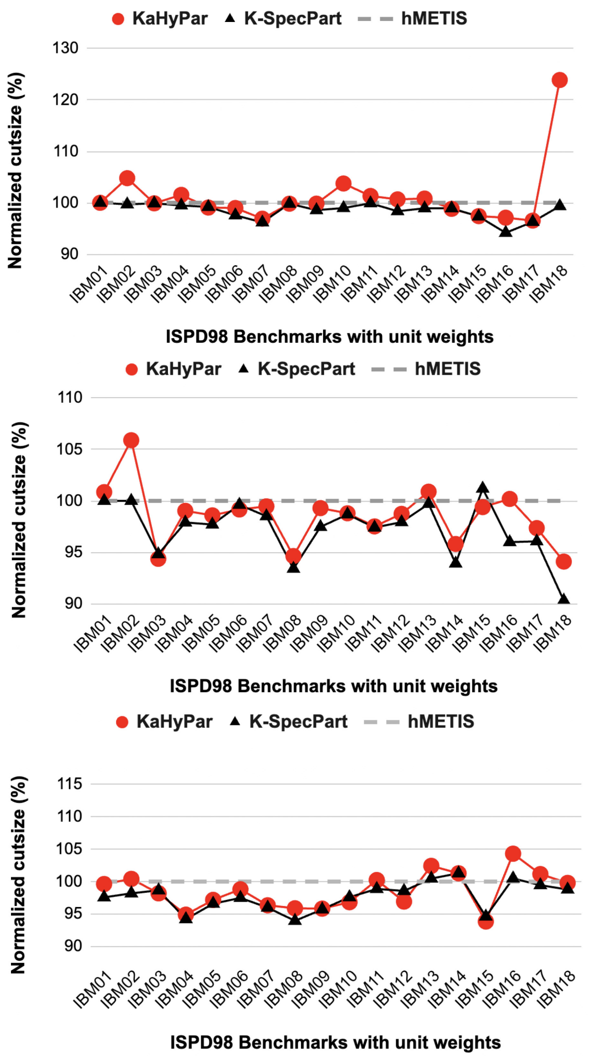


Fig. 7. *K-SpecPart* results on the *ISPD98 Benchmarks* [4] with unit vertex weights for $\epsilon = 2\%$. Top to bottom: $K = 2, 3, 4$.

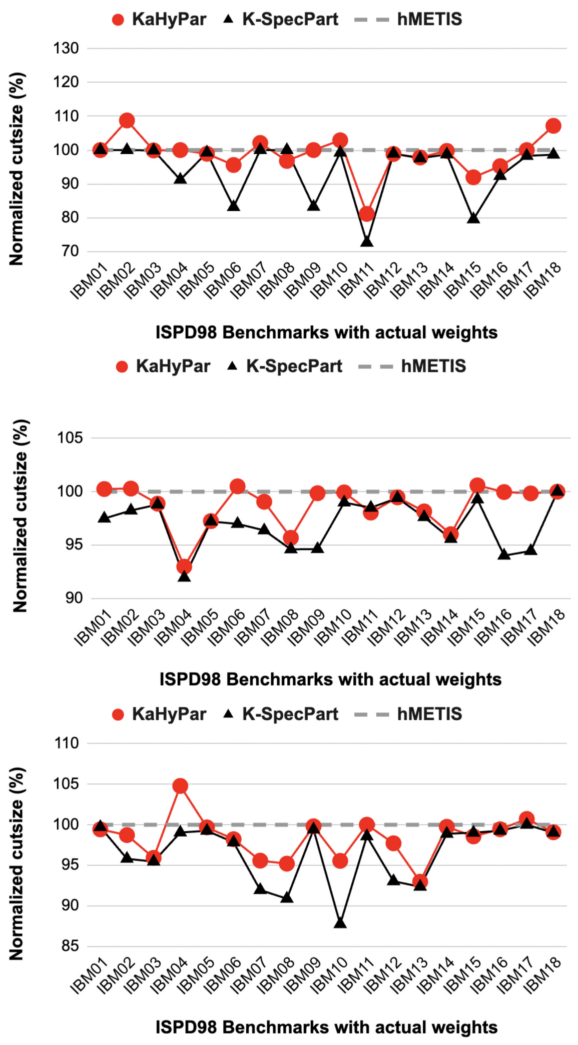


Fig. 8. *K-SpecPart* results on the *ISPD98 Benchmarks* [4] with actual vertex weights for $\epsilon = 2\%$. Top to bottom: $K = 2, 3, 4$. Cutsizes are normalized with respect to those by *hMETIS*$_5$.

the *ISPD98 VLSI Circuit Benchmark Suite* [4] and the *Titan23 Suite* [11]. We make public all partitioning solutions, scripts, and code at [52].

A. Cutsize Comparison

We run *hMETIS* and *KaHyPar* with their respective default parameter settings.[3] We denote by *hMETIS*$_t$ the best cutsize obtained by t runs of *hMETIS*, using t different seeds. We denote by *hMETIS*$_{avg}$ the average (over 50 samples) cutsize of *hMETIS*$_{20}$. We adopt similar notation for *KaHyPar*. In all our experiments we run *K-SpecPart* with its default settings (see Table I) and a hint that comes from *hMETIS*$_1$. We compare *K-SpecPart* against *hMETIS*$_5$ and *KaHyPar*$_5$; this is because 5 runs of *hMETIS* have a similar runtime with *K-SpecPart*. For a more robust and challenging comparison, we also compare *K-SpecPart* against *hMETIS*$_{avg}$ and *KaHyPar*$_{avg}$, which gives to these partitioners at least ∼5× the walltime of *K-SpecPart*.[4] In all our experiments, we adapt the imbalance factor as $eps = K \times \epsilon$ for *KaHyPar*.

ISPD98 Benchmarks With Unit Weights: Comparisons with *hMETIS*$_5$ and *KaHypar*$_5$ are presented in Fig. 7. *K-SpecPart* significantly improves over both *hMETIS*$_5$ and *KaHyPar*$_5$ on numerous benchmarks for both two-way and multiway partitioning. Comparisons with *hMETIS*$_{avg}$ and *KaHyPar*$_{avg}$ are reported in Table III. Each average value is rounded to the nearest tenth (0.1). We observe that *K-SpecPart* generates better partitions (∼2% better on some benchmarks) than *hMETIS*$_{avg}$ and *KaHyPar*$_{avg}$ on the majority of ISPD98 testcases.

ISPD98 Benchmarks With Actual Weights: The inclusion of weights makes the problem more general and potentially more challenging. Fig. 8 compares *K-SpecPart* against *hMETIS*$_5$ and *KaHyPar*$_5$, while Table IV provides comparisons with *hMETIS*$_{avg}$ and *KaHyPar*$_{avg}$. We see that *K-SpecPart* tends to

[3]For *hMETIS* [10], we use the multilevel recursive bisection paradigm (*hmetis*), and the default parameter setting is: Nruns = 10, CType = 1, RType = 1, Vcycle = 1, Reconst = 0, and seed = 0. The default configuration file we use for *KaHyPar* is cut_kKaHyPar_sea20.ini [51].

[4]The hint for *K-SpecPart* comes from *hMETIS*$_1$. Because *hMETIS*$_{avg}$ is the average (over 50 samples) cutsize of *hMETIS*$_{20}$, *K-SpecPart* may perform worse than *hMETIS*$_{avg}$ in some testcases.

TABLE III
COMPARISON OF *hMETIS* (*hMETIS*$_{avg}$), *KaHyPar* (*KaHyPar*$_{avg}$), AND *K-SpecPart* ON ISPD98 BENCHMARKS WITH UNIT VERTEX WEIGHTS FOR MULTIWAY PARTITIONING WITH NUMBER OF BLOCKS (K) = 2, 3, 4 AND IMBALANCE FACTOR (ϵ) = 2%

	Statistics		$K = 2$			$K = 3$			$K = 4$		
Benchmark	$\|V\|$	$\|E\|$	***hM***$_{avg}$	***KHPr***$_{avg}$	***K-SP***	***hM***$_{avg}$	***KHPr***$_{avg}$	***K-SP***	***hM***$_{avg}$	***KHPr***$_{avg}$	***K-SP***
IBM01	12752	14111	203.0	203.0	203	352.0	355.0	352	503.7	503.3	522
IBM02	19601	19584	331.5	350.4	333	339.4	359.1	339	676.5	694.5	706
IBM03	23136	27401	958.3	957.1	957	1544.2	1473.4	1480	1701.6	1682.9	1690
IBM04	27507	31970	581.3	592.6	580	1199.6	1223.6	1212	1669.4	1638.4	1626
IBM05	29347	28446	1728.6	1716.2	1716	2645.2	2644.8	2635	3031.2	2964.2	2946
IBM06	32498	34826	974.3	974.9	976	1306.7	1299.7	1305	1517.9	1496.3	1476
IBM07	45926	48117	910.3	907.9	935	1882.4	1864.8	1846	2201.4	2160.1	2154
IBM08	51309	50513	1141.2	1140.1	1140	2056.2	2042.4	2037	2401.5	2376.2	2328
IBM09	53395	60902	625.3	625.1	620	1404.1	1406.1	1384	1734.2	1676.4	1676
IBM10	69429	75196	1280.3	1326.3	1257	1911.8	1882.7	1880	2445.2	2381.1	2400
IBM11	70558	81454	1052.6	1066.2	1051	1808.5	1789.1	1843	2458.9	2472.6	2452
IBM12	71076	77240	1947.6	1965.3	1937	2817.3	2814.2	2791	3870.4	3799.2	3844
IBM13	84199	99666	844.3	848.1	832	1347.8	1351.9	1335	1913.2	1941.2	1904
IBM14	147605	152772	1875.1	1849.9	1850	2789.9	2603.7	2710	3401.4	3386.5	3475
IBM15	161570	186608	2817.2	2743.2	2741	4200.8	4252.5	4333	4870.7	4642.5	4720
IBM16	183484	190048	1925.6	2012.1	1921	3169.3	3197.6	3062	4045.2	4214.2	4060
IBM17	185495	189581	2364.5	2316.8	2307	4550.1	4305.6	4248	5634.6	5679.2	5583
IBM18	210613	201920	1531.6	1899.6	1523	2543.9	2501.9	2401	2949.4	2948.7	2918

TABLE IV
COMPARISON OF *hMETIS* (*hMETIS*$_{avg}$), *KaHyPar* (*KaHyPar*$_{avg}$), AND *K-SpecPart* ON ISPD98 BENCHMARKS WITH ACTUAL WEIGHTS FOR MULTIWAY PARTITIONING WITH NUMBER OF BLOCKS (K) = 2, 3, 4 AND IMBALANCE FACTOR (ϵ) = 2%

	Statistics		$K = 2$			$K = 3$			$K = 4$		
Benchmark	$\|V\|$	$\|E\|$	***hM***$_{avg}$	***KHPr***$_{avg}$	***K-SP***	***hM***$_{avg}$	***KHPr***$_{avg}$	***K-SP***	***hM***$_{avg}$	***KHPr***$_{avg}$	***K-SP***
IBM01$_w$	12752	14111	215.0	215.0	215	389.1	389.2	387	349.0	348.2	349
IBM02$_w$	19601	19584	324.6	322.3	296	340.9	341.2	334	548.4	540.2	524
IBM03$_w$	23136	27401	958.2	957.5	957	1249.8	1251.1	1230	1496.2	1454.7	1447
IBM04$_w$	27507	31970	584.7	580.9	529	899.6	806.2	797	1572.4	1531.1	1446
IBM05$_w$	29347	28446	1728.9	1715.2	1721	2640.1	2640.8	2642	3035.7	2966.3	3061
IBM06$_w$	32498	34826	969.3	972.8	845	997.2	997.1	963	1263.0	1266.2	1261
IBM07$_w$	45926	48117	812.3	820.9	803	1375.2	1365.4	1328	1902.4	1896.4	1824
IBM08$_w$	51309	50513	1142.4	1140.1	1182	1844.3	1791.6	1749	2407.9	2352.6	2268
IBM09$_w$	53395	60902	626.7	624.8	519	1407.2	1405.6	1334	1530.3	1541.3	1535
IBM10$_w$	69429	75196	1079.8	1066.2	1028	1576.0	1575.1	1560	2447.4	2334.1	2143
IBM11$_w$	70558	81454	845.2	853.6	763	1509.2	1501.3	1508	2069.5	2068.4	2068
IBM12$_w$	71076	77240	1947.3	1951.2	1967	2814.4	2815.2	3185	3859.7	3795.2	3613
IBM13$_w$	84199	99666	847.8	847.2	846	1639.1	1626.7	1636	1795.1	1796.5	1784
IBM14$_w$	147605	152772	1872.7	1858.7	1929	2814.5	2606.7	2780	3374.3	3264.8	3455
IBM15$_w$	161570	186608	2797.1	2746.8	2474	3864.5	3887.2	3836	4805.6	4636.4	4758
IBM16$_w$	183484	190048	1656.4	1664.4	1660	2942.3	2936.5	2742	3676.5	3736.2	3729
IBM17$_w$	185495	189581	2369.5	2326.3	2301	3589.3	3585.8	3580	5630.2	5726.3	5738
IBM18$_w$	210613	201920	1572.1	1926.4	1579	2503.3	2488.4	2836	2947.9	2976.1	3209

yield more significant improvements relative to the unit-weight case. For example, for IBM11$_w$, *K-SpecPart* generates almost 27% improvement over *hMETIS* and *KaHyPar* for $K = 2$. We notice similar improvements for $K > 2$ as seen on IBM04$_w$ for $K = 3$ and IBM10$_w$ for $K = 4$.

Titan23 Benchmarks: The *Titan23* benchmarks are interesting not only because they are substantially larger than the *ISPD98* benchmarks but also because they are generated by different, more modern synthesis processes. In some sense, they provide a "test of time" for *hMETIS*, as well as for *KaHyPar* which does not include *Titan23* in its experimental study [32]. Fig. 9 compares *K-SpecPart* against *hMETIS*$_5$, while Table V compares with *hMETIS*$_{avg}$. Although the *K-SpecPart* runtime is still similar to *hMETIS*$_5$, the runtime of *KaHyPar* on some of these benchmarks is exceedingly long (over 2 h), making it unsuitable for any reasonable industrial setting (for more details on runtime, see [52]). For this reason, we do not compare against *KaHyPar*. We observe that *K-SpecPart* generates better partitioning solutions compared to *hMETIS*$_5$ and *hMETIS*$_{avg}$. On *gsm_switch* in particular, *K-SpecPart* achieves more than 50% better cutsize.

B. Runtime Remarks

Our current Julia implementation of *K-SpecPart* has a walltime approximately 5× that of a single *hMETIS* run. Table V presents a detailed runtime comparison between *K-SpecPart* and *hMETIS*, while the runtime breakdown for *K-SpecPart* is shown in Fig. 10. *K-SpecPart* does utilize multiple cores, but there is still potential for speedup, in the following ways.

1) Most of the computational effort is in the embedding generation module as seen in Fig. 10. For $K > 2$, *K-SpecPart* employs limited parallelism in the embedding generation module, by solving in parallel the K eigenvector problem instances. The eigensolver has much more potential for parallelism since it relies on sequential and unoptimized sparse matrix–vector multiplications. These can be significantly speeded up on multicore CPUs, GPUs, or other specialized hardware.
2) In the tree partitioning module, *K-SpecPart* uses parallelism to handle partitioning of multiple trees. The most time-consuming component of this module is the cut distillation algorithm, where there is scope for runtime improvement, especially for larger instances. This can

TABLE V

COMPARISON OF *hMETIS* (*hMETIS*$_{avg}$) AND *K-SpecPart* ON TITAN23 BENCHMARKS FOR MULTIWAY PARTITIONING WITH NUMBER OF BLOCKS (K) = 2, 3, 4 AND IMBALANCE FACTOR (ϵ) = 2%. *hM*$_{avg}$*-t* AND *K-SP-t*, RESPECTIVELY, DENOTE THE RUNTIMES OF *hMETIS* AND *K-SpecPart* IN SECONDS

	Statistics		$K=2$				$K=3$				$K=4$			
Benchmark	$\|V\|$	$\|E\|$	***hM***$_{avg}$	***K-SP***	***hM***$_{avg}$***-t***	***K-SP-t***	***hM***$_{avg}$	***K-SP***	***hM***$_{avg}$***-t***	***K-SP-t***	***hM***$_{avg}$	***K-SP***	***hM***$_{avg}$***-t***	***K-SP-t***
sparcT1_core	91976	92827	982.2	977	7.2	46.4	2187.9	1889	13.8	36.7	2532.3	2492	16.1	74.3
neuron	92290	125305	245.0	244	7.4	37.2	371.6	396	12.6	60.5	431.5	431	15.7	81.6
stereo_vision	94050	127085	171.0	169	7.5	44.6	332.7	336	13.2	64.1	440.2	475	16.4	79.4
des90	111221	139557	376.8	374	10.3	54.6	536.5	535	14.7	72.2	695.5	747	18.1	92.6
SLAM_spheric	113115	142408	1061.0	1061	10.9	56.2	2797.1	2720	21.4	82.7	3371.1	3241	23.9	97.8
cholesky_mc	113250	144948	282.0	282	17.7	76.9	886.5	864	20.5	86.9	982.8	984	22.4	111.7
segmentation	138295	179051	120.1	120	19.7	92.2	476.1	453	22.2	104.7	496.3	490	28.7	123.8
bitonic_mesh	192064	235328	585.2	584	18.2	96.1	895.0	895	25.9	127.2	1304.4	1311	31.5	143.2
dart	202354	223301	837.0	805	18.5	98.9	1189.9	1243	30.4	132.8	1429.9	1401	30.7	160.4
openCV	217453	284108	435.4	434	22.1	108.5	501.8	525	28.9	142.6	526.2	522	34.6	166.7
stap_qrd	240240	290123	377.4	464	23.2	109.6	501.2	497	26.1	126.4	714.5	674	27.1	142.1
minres	261359	320540	207.0	207	26.4	121.2	309.0	309	31.5	152.6	407.0	407	36.1	176.2
cholesky_bdti	266422	342688	1156.0	1136	27.6	136.4	1769.2	1755	39.3	171.2	1874.4	1865	43.2	221.2
denoise	275638	356848	496.9	418	25.2	139.2	952.8	915	38.4	189.3	1172.1	1001	44.1	236.4
sparcT2_core	300109	302663	1220.7	1188	31.3	147.4	2827.2	2249	60.9	176.3	3323.5	3558	66.6	317.6
gsm_switch	493260	507821	4235.3	1833	28.9	137.2	4148.6	3694	56.2	265.8	5169.2	4404	54.6	296.7
mes_noc	547544	577664	634.6	633	38.1	186.7	1164.3	1125	61.3	312.5	1314.7	1346	71.5	346.1
LU230	574372	669477	3333.3	3363	51.3	236.2	4549.5	4548	81.5	388.4	6325.3	6310	84.3	409.3
LU_Network	635456	726999	524.0	524	53.2	265.9	787.1	882	80.6	412.3	1495.6	1417	87.8	442.4
sparcT1_chip2	820886	821274	914.2	876	71.1	364.6	1453.4	1404	97.4	491.2	1609.8	1601	136.4	625.1
directrf	931275	1374742	602.6	515	75.9	378.1	728.2	762	103.6	512.8	1103.6	1092	137.5	639.2
bitcoin_miner	1089284	1448151	1514.1	1562	80.9	398.7	1944.8	1917	127.2	626.8	2605.4	2737	159.5	797.3

be achieved by implementing the faster LCA algorithm in [9].

3) The CPLEX solver can also be accelerated by leveraging the "warm-start" feature where a previously computed partitioning solution can be used as an initial solution for the ILP. Furthermore, the CPLEX solver often computes a solution prior to its termination where extra time is spent to produce a computational proof of optimality [47]. Using a timeout is an option that can accelerate the solver without significantly affecting the quality of the output, but we have not explored this option in *K-SpecPart*.

To further investigate the scalability of *K-SpecPart* for large values of K, we run *K-SpecPart* with $K = 6, 8, 12, 24, 32, 48, 64, 80, 96, 128$ and $\epsilon = 2\%$ on six Titan23 benchmarks: 1) *sparcT1_core*; 2) *cholesky_mc*; 3) *segmentation*; 4) *denoise*; 5) *gsm_switch*; and 6) *directf*. The results are shown in Fig. 11. Cutsizes are normalized to *hMETIS*$_{10}$ and runtime is normalized to *hMETIS*$_1$. We *denoise* these results by reporting the average (over 20 samples) cutsize of *hMETIS*$_{10}$. The cutsize and runtime of *K-SpecPart* are also averaged over 20 samples, with each sample generated from a different *hMETIS* hint. We see that 1) the runtime of *K-SpecPart* grows linearly for large values of K and 2) for $K = 128$, *K-SpecPart* achieves $\sim 2\%$ improvement with over $60\times$ runtime overhead over *hMETIS*$_1$. The effectiveness of *K-SpecPart* decreases with increasing values of K. We leave improvement of this aspect of *K-SpecPart* as a direction for future work.

C. K-SpecPart Improvements Over SpecPart

We have also compared *K-SpecPart* directly against *SpecPart* [34] for the case $K = 2$. The results are presented in Fig. 12. Although *SpecPart* also improves the hint solutions from *hMETIS* and *KaHyPar*, we observe that *K-SpecPart* generates significant improvement (often in the range of 10%–15%) over *SpecPart* on various benchmarks. This improvement can be attributed to two main factors: 1) *K-SpecPart* refines the partitioning solutions generated from the constructed trees using an FM refinement algorithm and 2) *K-SpecPart* incorporates cut-overlay clustering and ILP-based partitioning in each iteration. To assess the contribution of each of these factors, we have conducted an ablation study on the Titan23 benchmarks and observe the following results: 1) enabling cut-overlay clustering while disabling FM refinement results in a roughly 4% average improvement over *SpecPart*; 2) enabling FM refinement while disabling cut-overlay clustering results in a roughly 3% average improvement over *SpecPart*; and 3) enabling both factors results in a roughly 6% average improvement over *SpecPart*.

D. Parameter Validation

We now discuss the sensitivity of *K-SpecPart* with respect to its parameters, shown in Table II. We define the score value as the average improvement of *K-SpecPart* with respect to *hMETIS*$_{avg}$ on benchmarks *sparcT1_core*, *cholesky_mc*, *segmentation*, *denoise*, *gsm_switch*, and *directf*, for $K = 2$ and $\epsilon = 5\%$. With respect to γ, we have found that using *hMETIS* instead of ILP for partitioning (i.e., setting $\gamma = 0$) worsens the score value by 2.68%. We have also found that settings of $\gamma > 500$ do not improve the score value. For the other parameters, we perform the following experiment. When we vary the value of one parameter (parameter sweep), the remaining parameters are fixed at their default values. The results are presented in Fig. 13. From the results of tuning parameters on *K-SpecPart* we establish that our default parameter setting represents a local minimum in the hyperparameter search space. We also notice that the cutsize does not improve with additional iterations if the number of iterations (β) exceeds two. This may be because further iterations do not increase the diversity of the pool of partitioning solutions used by *cut-overlay clustering* (Section VI).

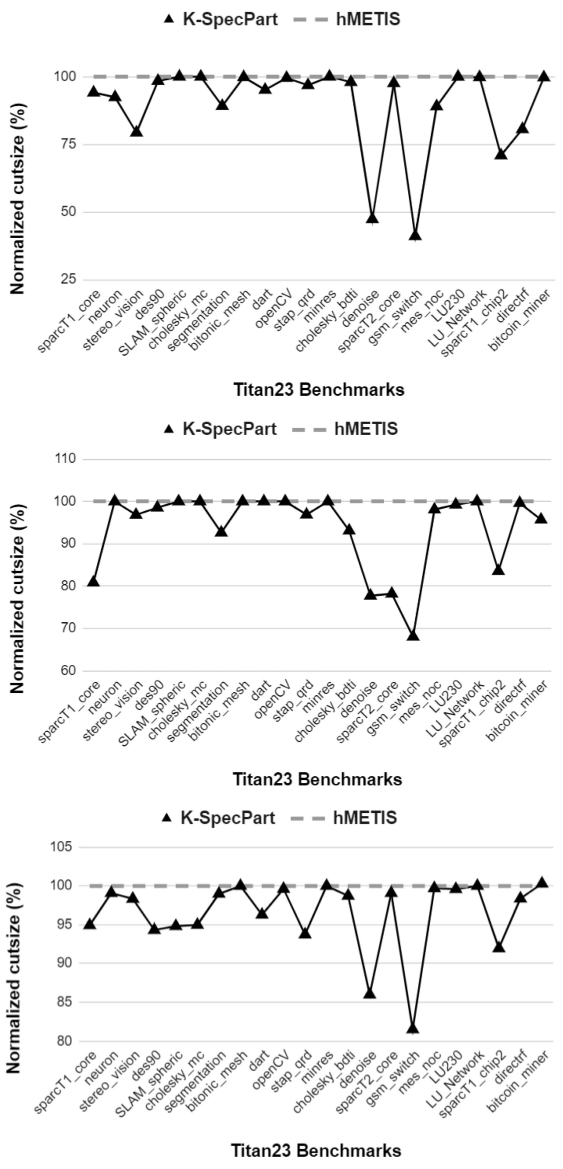


Fig. 9. *K-SpecPart* results on the *Titan23 Benchmarks* [11] for $\epsilon = 2\%$. Top-to-bottom: $K = 2, 3, 4$. Cutsizes are normalized with respect to those by $hMETIS_5$.

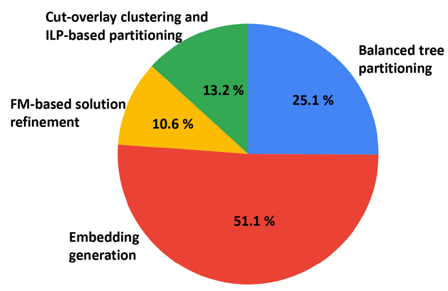


Fig. 10. Runtime breakdown of *K-SpecPart* for $K = 3$ and $\epsilon = 2\%$ on Titan23 benchmarks: *sparcT1_core*, *cholesky_mc*, *segmentation*, *denoise*, *gsm_switch*, and *directf*.

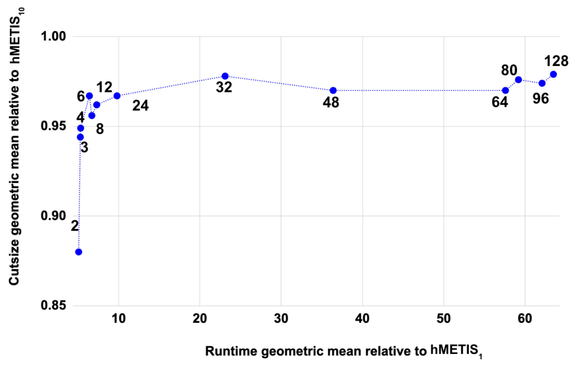


Fig. 11. Normalized cutsize versus runtime comparison of *K-SpecPart* for different values of K and $\epsilon = 2\%$: geometric means taken over six Titan23 benchmarks *sparcT1_core*, *cholesky_mc*, *segmentation*, *denoise*, *gsm_switch*, and *directf*. Numbers next to blue dots indicate the values of K.

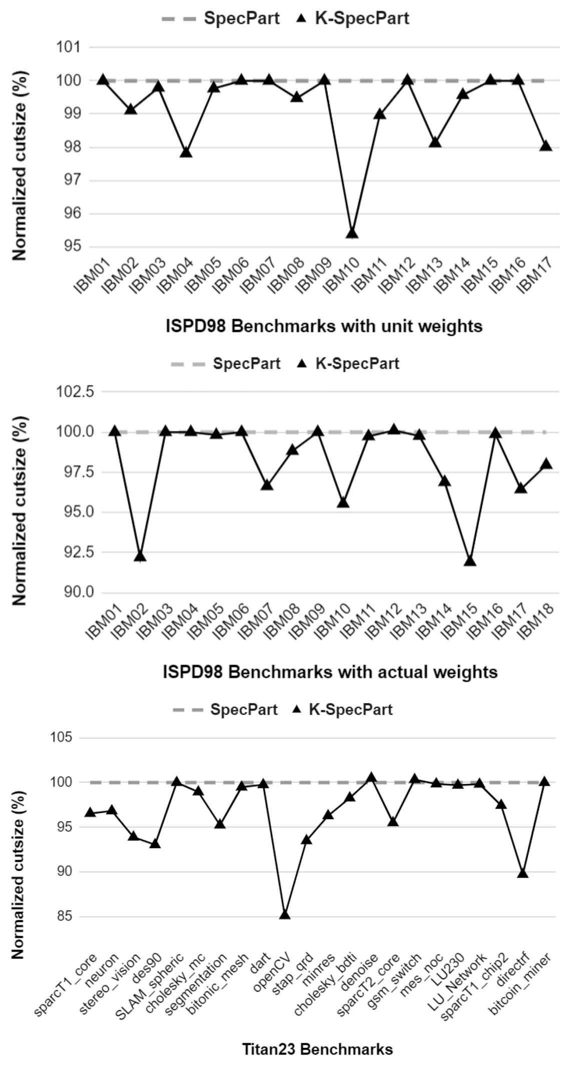


Fig. 12. Comparison of *K-SpecPart* and *SpecPart* on the bipartitioning problem ($\epsilon = 2\%$). Top-to-bottom: ISPD98 with unit weights, ISPD98 with actual weights, and Titan23.

E. Effect of Linear Discriminant Analysis

We have compared the cutsize and runtime of *K-SpecPart* with LDA, and *K-SpecPart* without LDA, i.e., utilizing the horizontally stacked eigenvectors X_{emb}. The result for multi-way partitioning ($K = 4$) is presented in Fig. 14. We observe that *K-SpecPart* with LDA generates slightly better (∼1%) cutsize with significantly faster (∼10×) runtime compared to *K-SpecPart* without LDA. However, for the case of bipartitioning ($K = 2$) we do not observe any significant difference in cutsize when employing LDA.

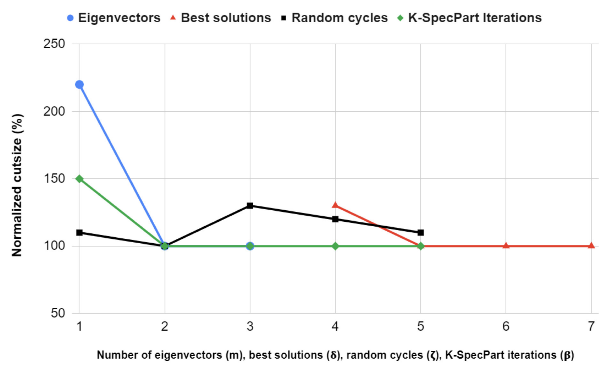


Fig. 13. Validation of *K-SpecPart* parameters. (a) Number of eigenvectors (m) sweep. (b) Number of best solutions (δ) sweep. (c) Number of iterations (β) sweep. (d) Number of random cycles (ζ) sweep.

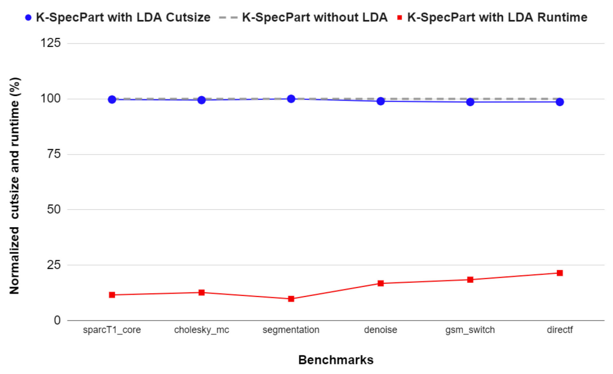


Fig. 14. Comparison of cutsize and runtime of *K-SpecPart* with LDA and *K-SpecPart* without LDA, for $K = 4$ and $\epsilon = 2\%$.

F. VILE Versus Recursive Balanced Tree Partitioning

We additionally compare the "VILE" tree partitioning algorithm (Section V-C) with a balanced tree partitioning baseline, based on a recursive two-way cut distilling and partitioning of the tree, similar to Section V. During each level of recursive partitioning, we dynamically adjust the balance constraint to ensure that the final K-way partitioning solution satisfies the balance constraints (see Section II-A). In particular, while executing the ith $(1 \leq i \leq K-1)$ level bipartitioning, the balance constraints associated with the bipartitioning solution $S(V_{i0}, V_{i1})$ are

$$\left(\frac{1}{K} - \epsilon\right)W \leq \sum_{v \in V_{i0}} w_v \leq \left(\frac{1}{K} + \epsilon\right)W \quad (4)$$

$$\sum_{v \in V_{i0}} w_v \geq \left(\sum_{v \in V_{i0}, V_{i1}} w_v\right) - (K-i)\left(\frac{1}{K} + \epsilon\right)W. \quad (5)$$

After obtaining the bipartitioning solution $S(V_{i0}, V_{i1})$, we proceed with the $(i+1)$th level bipartitioning. A comparison of cutsize obtained with VILE tree partitioning and balanced tree partitioning is presented in Fig. 15. The plots are normalized with respect to the cutsize obtained with balanced tree partitioning. We observe that VILE tree partitioning yields better cutsize (on average 2% better) compared to balanced tree partitioning.

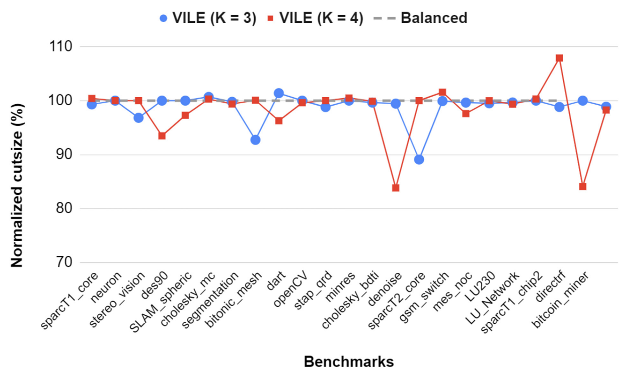


Fig. 15. Comparison of VILE tree partitioning and balanced tree partitioning for $K = 3, 4$ and $\epsilon = 2\%$.

G. Effect of Supervision in K-SpecPart

In order to show the effect of supervision in *K-SpecPart*, we run *solution ensembling via cut overlay* directly on candidate solutions, which are generated by running *hMETIS* multiple times with different random seeds. The flow is as follows.

1) We generate candidate solutions $\{S_1, S_2, \ldots, S_\psi\}$ by running *hMETIS* ψ times with different random seeds, and report the best cutsize *Multistart-hMETIS*. Here, ψ is an integer parameter ranging from 1 to 20.
2) We run *solution ensembling via cut overlay* directly on the best five solutions from $\{S_1, S_2, \ldots, S_\psi\}$ and report the cutsize *Solution-overlay-part*. For each value of ψ, we run this flow 100 times and report the average result in Fig. 16.

Based on the results shown in Fig. 16, we draw the following conclusions.

1) The *Solution-overlay-part* is much better than *Multistart-hMETIS*, which shows that *solution ensembling via cut overlay* produces solutions of higher quality compared to the plain *multistart* approach. Put another way, *solution ensembling via cut overlay* is established as a stronger baseline than the plain *multistart* approach.
2) Even compared against the stronger baseline of *solution ensembling via cut overlay*, *K-SpecPart* generates superior solutions in less runtime. This suggests that supervision is an important component of *K-SpecPart*.

H. Solution Enhancement by Autotuning

hMETIS has parameters whose settings may significantly impact the quality of generated partitioning solutions. We use Ray [54] to tune the following parameters of *hMETIS*: CType with possible values $\{1, 2, 3, 4, 5\}$, RType with possible values $\{1, 2, 3\}$, Vcycle with possible values $\{1, 2, 3\}$, and Reconst with possible values $\{0, 1\}$. The search algorithm we use in Ray [54] is *HyperOptSearch*. We set the number of trials, i.e., the total number of runs of *hMETIS* launched by Ray, to 5, 10, and 40. We set the number of threads to 10 to reduce the runtime (elapsed walltime). Here, we normalize the cutsize and runtime to that of running *hMETIS* once with default random seed. Autotuning increases the runtime for *hMETIS* and computes a better hint S_{init}; it leads to further cutsize improvements of 15% and 2% from *K-SpecPart* on

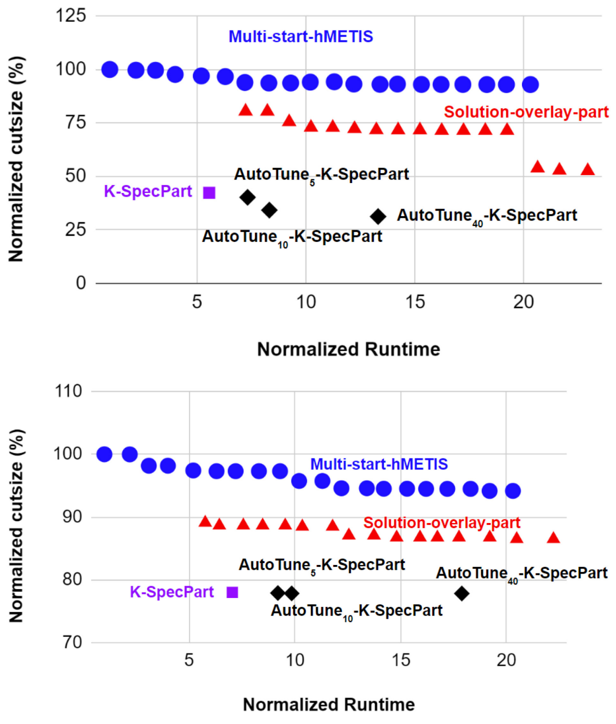


Fig. 16. Cutsize versus runtime on *gsm_switch*, for $\epsilon = 2\%$. Top-to-bottom: $K = 2, 4$.

gsm_switch for $K = 2$ and $K = 4$, respectively.[5] This suggests that the cutsize improvement from *K-SpecPart* could be further augmented if a better hint is provided.

VIII. Conclusion and Future Directions

We have proposed *K-SpecPart*, the first general supervised framework for hypergraph multiway partitioning solution improvement. Our experimental results demonstrate the superior performance of *K-SpecPart* in comparison to traditional multilevel partitioners, while maintaining comparable runtimes for both bipartitioning and multiway partitioning. The findings from *SpecPart* and *K-SpecPart* indicate that the partitioning problem may not be as comprehensively solved as previously believed, and that substantial advancements may yet remain to be discovered. *K-SpecPart* can be integrated with the internal levels of multilevel partitioners; producing improved solutions on each level may lead to further improve solutions. Furthermore, we believe that the *cut-overlay clustering* and LDA-based embedding generation hold independent interest and are amenable to machine learning techniques. Extensions of *K-SpecPart* to handle the connectivity metric will also be of interest.

Acknowledgment

The authors thank Dr. Grigor Gasparyan for sharing his thoughts on *K-SpecPart*.

[5]On *gsm_switch* with $K = 2$, the tuned *hMETIS* parameters are: Nruns = 10, CType = 4, RType = 1, Vcycle = 3, Reconst = 0, and seed = 20. For $K = 4$, the parameters are: Nruns = 10, CType = 2, RType = 2, Vcycle = 0, Reconst = 0, and seed = 34.

References

[1] M. Cucuringu, I. Koutis, S. Chawla, G. Miller, and R. Peng, "Simple and scalable constrained clustering: A generalized spectral method," in *Proc. 19th Int. Conf. Artif. Intell. Stat.*, 2016, pp. 445–454.
[2] N. Alon, R. M. Karp, D. Peleg, and D. West, "A graph-theoretic game and its application to the *k*-server problem," *SIAM J. Comput.* vol. 24, no. 1, pp. 78–100, 1995.
[3] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proc. Amer. Math. Soc.*, vol. 7, no. 1, pp. 48–50, Feb. 1956.
[4] C. J. Alpert, "The ISPD98 circuit benchmark suite," in *Proc. ACM/IEEE Int. Symp. Phys. Des.*, 1998, pp. 80–85.
[5] R. Andre, S. Schlag, and C. Schulz, "Memetic multilevel hypergraph partitioning," in *Proc. Genet. Evol. Comput. Conf.*, 2018, pp. 347–354.
[6] L. Gottesbüren, T. Heuer, P. Sanders, C. Schulz, and D. Seemaier, "Deep multilevel graph partitioning," in *Proc. Annu. Eur. Symp. Algorithm*, 2021, pp. 1–17.
[7] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.* vol. 20, no. 1, pp. 359–392, 1998.
[8] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Applications in VLSI domain," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 7, no. 1, pp. 69–79, Mar. 1999.
[9] M. Bender and M. Farach-Colton, "The LCA problem revisited," in *Proc. Latin Amer. Symp. Theor. Informat.*, 2000, pp. 88–94.
[10] G. Karypis and V. Kumar. *hMETIS, A Hypergraph Partitioning Package, Version 1.5.3*. (1998). [Online]. Available: http://glaros.dtc.umn.edu/gkhome/fetch/sw/hMETIS/manual.pdf
[11] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD," in *Proc. 23rd Int. Conf. Field Program. Logic Appl.*, 2013, pp. 1–8.
[12] S. Balakrishnama and A. Ganapathiraju, "Linear discriminant analysis—A brief tutorial," in *Proc. Inst. Signal Inf. Process.*, vol. 18, 1998, pp. 1–8.
[13] Ü. Çatalyürek and C. Aykanat, "PaToH (partitioning tool for hypergraphs)," in *Encyclopedia of Parallel Computing*. Boston, MA, USA: Springer, 2011.
[14] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017.
[15] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. IEEE/ACM Des. Autom. Conf.*, 1982, pp. 175–181.
[16] B. Ghojogh, F. Karray, and M. Crowley, "Eigenvalue and generalized eigenvalue problems: Tutorial," 2019, *arXiv:1903.11240*.
[17] R. Shaydulin, J. Chen, and I. Safro, "Relaxation-based coarsening for multilevel hypergraph partitioning," *Multiscale Model. Simul.*, vol. 17, no. 1, pp. 482–506, 2019.
[18] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM J. Sci. Comput.* vol. 23, no. 2, pp. 517–541, 2001.
[19] T. Heuer, P. Sanders, and S. Schlag, "Network flow-based refinement for multilevel hypergraph partitioning," *ACM J. Exp. Algorithm.*, vol. 24, no. 2, pp. 1–36, Sep. 2019.
[20] A. Henzinger, A. Noe, and C. Schulz, "ILP-based local search for graph partitioning," *ACM J. Exp. Algorithm.*, vol. 25, pp. 1–26, Jul. 2020.
[21] D. Kucar, S. Areibi, and A. Vannelli, "Hypergraph partitioning techniques," *Dyn. Contin., Discret. Impuls. Syst. Ser. A Math. Anal.*, vol. 11, no. 2, pp. 339–367, 2004.
[22] R. Merris, "Laplacian matrices of graphs: A survey," *Linear Algebra Appl.*, vol. 197, pp. 143–176, Jan./Feb. 1994.
[23] A. V. Knyazev, I. Lashuk, M. E. Argentati, and E. E. Ovchinnikov, "Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in hypre and PETSc," *SIAM J. Sci. Comput.*, vol. 25, no. 5, pp. 2224–2239, 2007.
[24] I. Koutis, G. L. Miller, and R. Peng, "Approaching optimality for solving SDD linear system," *SIAM J. Comput.*, vol. 43, no. 1, pp. 337–354, 2014.
[25] J. G. Sun and G. W. Stewart, *Matrix Perturbation Theory*. Boston, MA, USA: Academic, 1990.
[26] I. Koutis, G. L. Miller, and D. Tolliver, "Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing," *Comput. Vis. Image Understand.*, vol. 115, no. 12, pp. 1638–1646, Dec. 2011.
[27] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Improved algorithms for hypergraph bipartitioning," in *Proc. IEEE/ACM Des. Autom. Conf.*, 2000, pp. 661–666.

[28] J. R. Lee, S. O. Gharan, and L. Trevisan, "Multiway spectral partitioning and higher-order cheeger inequalities," *J. ACM*, vol. 61, no. 6, pp. 1–30, Dec. 2014.

[29] S. Mika, G. Rätsch, J. Weston, B. Schölkopf, and K.-R. Müller, "Fisher discriminant analysis with kernels," in *Proc. IEEE Signal Process. Soc. Workshop Neural Netw. Signal Process.*, 1999, pp. 41–48.

[30] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Ann. Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[31] T. Heuer, "Engineering initial partitioning algorithms for direct k-way hypergraph partitioning," B.S. thesis, Dept. Informat., Karlsruhe Inst. Technol., Karlsruhe, Germany, 2015.

[32] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, and P. Sanders, "High-quality hypergraph partitioning," *ACM J. Exp. Algorithm.*, vol. 27, pp. 1–39, Feb. 2023.

[33] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz, "k-way hypergraph partitioning via n-level recursive bisection," in *Proc. Meet. Algorithm Eng. Exp.*, 2016, pp. 53–67.

[34] I. Bustany, A. B. Kahng, Y. Koutis, B. Pramanik, and Z. Wang, "SpecPart: A supervised spectral framework for hypergraph partitioning solution improvement," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2022, pp. 1–9.

[35] J. Y. Zien, M. D. F. Schlag, and P. K. Chan, "Multilevel spectral hypergraph partitioning with arbitrary vertex sizes," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 9, pp. 1389–1399, 1999.

[36] L. Hagen and A. B. Kahng, "Fast spectral methods for ratio cut partitioning and clustering," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1991, pp. 10–13.

[37] T. Heuer and S. Schlag, "Improving coarsening schemes for hypergraph partitioning by exploiting community structure," in *Proc. Int. Symp. Exp. Algorithms*, 2017, pp. 1–19.

[38] N. Rebagliati and A. Verri, "Spectral clustering with more than *K* eigenvectors," *Neurocomputing* vol. 74, no. 9, pp. 1391–1401, Apr. 2011.

[39] C. J. Alpert and A. B. Kahng, "Multiway partitioning via geometric embeddings, orderings, and dynamic programming," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 14, no. 11, pp. 1342–1358, 1995.

[40] R. Horaud (INRIA Grenoble-Rhône-Alpes Res. Centre, Montbonnot-Saint-Martin, France). *A Short Tutorial on Graph Laplacians, Laplacian Embedding, and Spectral Clustering*. (2009). [Online]. Available: https://csustan.csustan.edu/ tom/Clustering/GraphLaplacian-tutorial.pdf

[41] F. R. K. Chung, *Spectral Graph Theory*. Providence, RI, USA: Amer. Math. Soc., 1997.

[42] I. Koutis, G. Miller, and R. Peng, "A generalized cheeger inequality," *Linear Algebra Appl.*, vol. 665, pp. 139–152, May 2023.

[43] M. Kapralov and R. Panigrahy, "Spectral sparsification via random spanners," in *Proc. Innovat. Theor. Comput. Sci. Conf.*, 2012, pp. 393–398.

[44] S. Hoory and N. Linial, and A. Wigderson, "Expander graphs and their applications," *Bull. Trans. Amer. Math. Soc.*, vol. 43, no. 4, pp. 439–561, 2006.

[45] C. Ravishankar, D. Gaitonde, and T. Bauer, "Placement strategies for 2.5D FPGA fabric architectures," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2018, pp. 16–164.

[46] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Ann. History Comput.*, vol. 7, no. 1, pp. 43–57, Jan.–Mar. 1985.

[47] "IBM ILOG CPLEX optimizer version 12.8.0." ibm.com. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[48] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mechan. Theory Exp.*, vol. 2008, no. 10, p. 10008, 2008.

[49] E. G. Boman and B. Hendrickson, "Support theory for preconditioning," *SIAM J. Matrix Anal. Appl.*, vol. 25, no. 3, pp. 694–717, 2003.

[50] C. J Alpert, A. B. Kahng, and S.-Z. Yao, "Spectral partitioning with multiple eigenvectors," *Discret. Appl. Math.*, vol. 90, nos. 1–3, pp. 3–26, 1999.

[51] github.com. 2023. [Online]. Available: https://github.com/kahypar/kahypar/blob/master/config/cut_rKaHyPar_sea20.ini

[52] "Partition solutions, scripts and K-SpecPart." github.com. 2023. [Online]. Available: https://github.com/TILOS-AI-Institute/HypergraphPartitioning

[53] "TritonPart, an open-source partitioner." github.com. 2023. [Online]. Available: https://github.com/ABKGroup/TritonPart_OpenROAD

[54] Ray. github.com. 2023. [Online]. Available: https://docs.ray.io/en/latest/index.html

[55] Y.-C. A. Wei and C.-K. Cheng, "Towards efficient hierarchical designs by ratio cut partitioning," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1989, pp. 298–301.

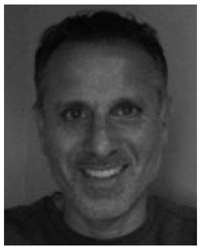
[56] "Google OR-tools version 9.4." github.com. 2023. [Online]. Available: https://developers.google.com/optimization/

[57] "MultivariateStats.jl." github.com. 2023. [Online]. Available: https://github.com/JuliaStats/MultivariateStats.jl

[58] "Combinatorial multigrid solver, an implementation in Julia." github.com. 2023. [Online]. Available: https://github.com/bodhi91/CombinatorialMultigrid.jl

[59] "Experimental results of KaHyPar." github.com. 2023. [Online]. Available: https://github.com/kahypar/kahyparexperimental-results

[60] "KaHIP." github.com. 2023. [Online]. Available: https://kahip.github.io/



Ismail Bustany (Member, IEEE) received the M.S. and Ph.D. degrees in EECS from the University of California at Berkeley, Berkeley, CA, USA, in 1994 and 1999, respectively.

He is a Fellow with Advanced Micro Devices Inc., Santa Clara, CA, USA, working in the Adaptive and Embedded Computing Group on physical design algorithms and MLCAD. His research interests include physical design, computationally efficient optimization algorithms, MLCAD, sparse matrix computations, hypergraph partitioning, and hardware acceleration.

Andrew B. Kahng (Fellow, IEEE) received the Ph.D. degree in computer science from the University of California at San Diego, La Jolla, CA, USA, in 1989.

He is a Distinguished Professor with the Department of Computer Science and Engineering and the Department of Electrical and Computer Engineering, University of California at San Diego. His interests include IC physical design, the design–manufacturing interface, large-scale combinatorial optimization, AI/ML for EDA and IC design, and technology roadmapping.

Ioannis Koutis (Member, IEEE) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 2007.

He is an Associate Professor of Computer Science with the New Jersey Institute of Technology, Newark, NJ, USA. His interests include spectral graph theory and its powerful applications in designing algorithms for problems on graphs and hypergraphs with applications in various contexts, including electronic design automation.

Bodhisatta Pramanik (Student Member, IEEE) received the M.S. degree in computer engineering from Iowa State University, Ames, IA, USA, in 2022. He is currently pursuing the Ph.D. degree with the University of California at San Diego, La Jolla, CA, USA.

His research interests include hypergraph partitioning, graph clustering, placement methodology, and optimization algorithms.

Zhiang Wang (Student Member, IEEE) received the M.S. degree in electrical and computer engineering from the University of California at San Diego, La Jolla, CA, USA, in 2022, where he is currently pursuing the Ph.D. degree.

His current research interests include partitioning, placement methodology, and optimization.