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 K-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.

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 MLParr [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, Shyadihun 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.

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 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: iokinis.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: zhiangw@ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2023.3332268
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 (~10x) but also slightly improves (~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 multway 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).

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>H(V,E)</td>
<td>Hypergraph H with vertices V and hyperedges E</td>
</tr>
<tr>
<td>Hc(Vc, Ec)</td>
<td>Clustered hypergraph Hc, where each vertex vc ∈ Vc corresponds to a group of vertices in H(V,E)</td>
</tr>
<tr>
<td>G(V,E)</td>
<td>Graph G with vertices V and edges E</td>
</tr>
<tr>
<td>G</td>
<td>Spectral sparsifier of G</td>
</tr>
<tr>
<td>T(V, ET)</td>
<td>Tree T with vertices V and edges ET</td>
</tr>
<tr>
<td>u, v</td>
<td>Vertices in V</td>
</tr>
<tr>
<td>εuv</td>
<td>Edge connecting u and v</td>
</tr>
<tr>
<td>εT</td>
<td>Edge of tree T</td>
</tr>
<tr>
<td>wvc, wc</td>
<td>Weight of vertex v, or hyperedge e, respectively</td>
</tr>
<tr>
<td>K</td>
<td>Number of blocks in a partitioning solution</td>
</tr>
<tr>
<td>S</td>
<td>Partitioning solution, S = {V0, V1, ..., VK-1}</td>
</tr>
<tr>
<td>t</td>
<td>Allowed imbalance between blocks in S</td>
</tr>
<tr>
<td>cut(S)</td>
<td>Cut of S, cut(S) = {e ∈ E</td>
</tr>
<tr>
<td>cutsize+(S)</td>
<td>Cutsizes of S on H, i.e., sum of wvc, e ∈ cut(S)</td>
</tr>
<tr>
<td>Xemb, X</td>
<td>Vertex embeddings</td>
</tr>
</tbody>
</table>

Table I: Notation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description (default setting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Number of eigenvectors (m = 2)</td>
</tr>
<tr>
<td>δ</td>
<td>Number of best solutions (δ = 5)</td>
</tr>
<tr>
<td>β</td>
<td>Number of iterations of K-SpecPart (β = 2)</td>
</tr>
<tr>
<td>ζ</td>
<td>Number of random cycles (ζ = 2)</td>
</tr>
<tr>
<td>γ</td>
<td>Threshold of number of hyperedges (γ = 500)</td>
</tr>
</tbody>
</table>

Table II: Parameters of the K-SpecPart Framework

Authorized licensed use limited to: Univ of Calif San Diego. Downloaded on March 22,2024 at 00:57:23 UTC from IEEE Xplore. Restrictions apply.
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 $S = \bigcup_i V_i$ of $K$ vertex blocks $V_i \subseteq V$ such that $V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^{K} V_i = V$.

2) Vertex Set Weight: For $U \subseteq V$, $W_U = \sum_{v \in U} w_v$.

3) $\epsilon$-Balanced $K$-Way Partition $S$: A $K$-way partition such that for all $V_i \subseteq S$, we have $0 \leq (1/K) - \epsilon \leq W_{V_i}/W_V \leq (1/K) + \epsilon$.

4) cut$_H(S) = \{ e \mid e \not\subseteq V_i \text{ for all } V_i \subseteq S \}$.

5) cutsize$_H(S) = \sum_{e \in cut_H(S)} w_e$.

The hypergraph partitioning problem seeks an $\epsilon$-balanced $K$-way partition $S$ that minimizes cutsize$_H(S)$.

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) $L(u, v) = -w_{uv}$ if $u \neq v$ and 2) $L(u, u) = \sum_{v \neq u} w_{uv}$. 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

$$x^T L x = \text{cutsize}_{G}(S).$$

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 $R(x)$ as

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

Minimizing $R(x)$ over 0-1 vectors $x$ incentivizes a small cutsize$_G(S)$ with a simultaneous balance of $|S|$ 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 $L_{G_C}$. A minimizer of (2) is given by the first nontrivial eigenvector of the problem $L x = \lambda L_{G_C} x$.

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

$$L_G x = \lambda B x \quad (3)$$

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 $B$.

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 $i$th 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 $\hat{X}_{N \times m}$, 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 $S_{M \times M}^{SM}$ and $S_{M \times M}^{SB}$ 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 $S_{M \times M}^{SB}$, and lets $\hat{X} = XP$. 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 $0 < i < K$.

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) $(\frac{1}{K}) - \epsilon \leq \sum_{i=1}^{K} W_{V_i} y_{e, i} \leq (\frac{1}{K}) + \epsilon W$, where $W = \sum_{i=1}^{K} W_{V_i}$.

\footnote{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.}
The objective is to maximize the total weight of the hyperedges that are not cut, i.e.,
\[
\max \sum_{e \in E} \sum_{i=0}^{K-1} w_e y_{e,i}.
\]

III. K-SPEC PART 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 \( 0, 1, \ldots, K-1 \) 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 \( \{S_{i0}, S_{i1}, \ldots, S_{i\beta}\} \) (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.

IV. SUPERVISED VERTEX EMBEDDING

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

In K-SpecPart, we use a spectral embedding algorithm that encodes into a generalized eigenvalue problem the supervision information \( S_{\text{hint}} \). Fig. 2 illustrates how the inclusion of the hint incentives the computation of an embedding that in

Algorithm 1: K-SpecPart Framework

Input: Hypergraph \( H(V, E) \), Number of blocks \( K \), Initial partitioning solution \( S_{\text{init}} \), Number of supervision iterations \( \beta \), Allowed imbalance between blocks \( \epsilon \)
Output: Improved partitioning solution \( S_{\text{out}} \)
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 \( [S_{\text{candidate}}] \)
4. \( S_0 = S_{\text{init}} \)
5. for \( i = 0; i < \beta; i++ \) do
   /* Supervised hypergraph vertex embedding (Section IV) */
   if \( K = 2 \) then
      Construct the hint graph \( G_h \) based on \( S_i \) and the Laplacian matrix \( L_{G_h} \) of \( G_h \) (Section IV-A)
      Solve the generalized eigenvalue problem \( L_{G_h} x = \lambda(L_{G_h} + L_{G_w}) x \) to obtain the first \( m \) nontrivial eigenvectors \( X_{\text{emb}} \in \mathbb{R}^{\forall \times m} \)
   else
      Decompose the K-way partitioning solution \( S_i \) into \( K \) bipartitioning (2-way) solutions \( \{S_{i0}, \ldots, S_{iK-1}\} \) (Section IV-B)
      for \( j = 0; j < K; j++ \) do
         Construct the hint graph \( G_h \) based on \( S_j \) and the Laplacian matrix \( L_{G_h} \) of \( G_h \) (Section IV-A)
         Solve the generalized eigenvalue problem \( L_{G_h} x = \lambda(L_{G_h} + L_{G_w}) x \) to obtain the first \( m \) nontrivial eigenvectors \( X_{\text{emb}} \in \mathbb{R}^{\forall \times m} \)
      end
      Perform linear discriminant analysis (LDA) to generate the vertex embedding \( X_{\text{emb}} \in \mathbb{R}^{\forall \times m} \)
   end
   /* Extracting solutions from embedding (Section VI) */
   Construct a family of trees \( \{T_0, T_1, \ldots\} \) leveraging the vertex embedding \( X_{\text{emb}} \in \mathbb{R}^{\forall \times m} \)
   Generate hypergraph partitioning solutions \( \{S_{T_0}, S_{T_1}, \ldots\} \) through cut distilling and tree partitioning
   /* Solution ensembling via cut overlay (Section VI) */
   \( S_{i+1} \leftarrow \) perform cut-overlay clustering and ILP-based partitioning on the top 8 solutions from \( \{S_{T_0}, S_{T_1}, \ldots\} \)
   \( [S_{\text{candidate}}] \), push_back \( (S_{i+1}) \)
   end
   /* Solution ensembling via cut overlay */
   \( S_{\text{out}} \leftarrow \) perform cut-overlay clustering and ILP-based partitioning on solutions \( [S_{\text{candidate}}] \)
27. Refine \( S_{\text{out}} \) using multi-way FM
28. return \( S_{\text{out}} \)

general respects (spatially) the given solution \( S_{\text{hint}} \), 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
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 $\frac{1}{|e|} - 1$. The weight-balanced 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 $S_h$.

2) Generalized Eigenvalue Problem and Embedding: Given a two-way partitioning solution $S_{hint}$, we solve the generalized eigenvalue problem $L_GX = \lambda LX$ where $L = L_{G_{w}} + L_{G_{h}}$, and compute the first $m$ nontrivial eigenvectors $X \in \mathbb{R}^{V \times m}$ whose rows provide the vertex embedding. We solve $L_GX = \lambda BX$ 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, $S_{b_j}$, for $i = 0, \ldots, K - 1$, where

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

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 $X_{emb} \in \mathbb{R}^{V \times K \times m}$, i.e.,

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

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 $\epsilon$, and an embedding $X_{emb} \in \mathbb{R}^{V \times m}$, and the output is a pool of solutions $\{S_0, \ldots, S_{b_j-1}\}$. 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 $X_{emb}$ be the $i$th column of $X$. We sort the values in $X_{emb}$ and let $o(j)$ be vertex at the $j$th position of the sorted $X_{emb}$. 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 $\xi$ 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 $\hat{G}$ by setting the weight of each edge $e_{uv} \in G$ to $\sum |Y_u - Y_v|^2$, i.e., equal to the Euclidean distance between the two vertices in the embedding. Using $\hat{G}$ we build two spanning trees.

Low Stretch Spanning Tree (LSST): A desired property for a spanning tree $\hat{T}$ of $\hat{G}$ 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 $\hat{G}$ 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$_H(S_{e_T})$.

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 $e = \{v_1, v_3, v_9\}$. 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 $\{v_1, v_3, v_9\}$ 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 $\{v_1, v_3\}$ and $\{v_1, v_3, v_9\}$.
2) We then “label” these junctions with $-w_e$, where $w_e$ is the weight of $e$. More generally, for a hyperedge $e = \{v_{i1}, \ldots, v_{ip}\}$ ordered according to the post-order depth-first search traversal on $T$, we compute the LCAs for the $p - 1$ sets $\{v_{i1}, \ldots, v_{ij}\}$ 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 hyeredges 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 \textit{visited} sequence

   */ Label each vertex in \( T \) based on the
   hyperedge weight \( w_e \) for \( e \in E \) */
3. \( \text{cuts}_\text{delta}[v] \leftarrow 0 \) for all \( v \in V \)
4. for each \( e \) in \( E \) do
5. \( \text{cuts}_\text{delta}[v] = \text{cuts}_\text{delta}[v] + w_e \) for all \( v \) in \( e \)
6. \( \{v_0, \ldots, v_{|e|-1}\} \leftarrow \) arrange the vertices of \( e \) according to their positions in \textit{visited} sequence
7. \( v_{LCA} \leftarrow v_0 \)
8. for \( j = 1; j < |e|; j++ \) do
9. \( v_{LCA} \leftarrow \) identify the least common ancestor (LCA) for \( v_{LCA} \) and \( v_j \) in \( T \)
10. \( \text{cuts}_\text{delta}[v_{LCA}] = \text{cuts}_\text{delta}[v_{LCA}] - w_e \)
11. end
12. \( \text{cuts}_\text{delta}[v_{LCA}] = \text{cuts}_\text{delta}[v_{LCA}] - w_e \)
13. end
14. for each \( e_T \) in \( E_T \) do
15. \( w_{e_T} \leftarrow \text{compute the sum-below-}e_T \) (i.e., the sum of the labels \( \text{cuts}_\text{delta} \) 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

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 \( \delta \) best solutions. Let \( E_1, \ldots, E_\delta \subseteq E \) be the sets of hyperedges cut in the \( \delta \) 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 \( H_c \). 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 \( \delta \) 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 \( \gamma \) in the case when the number of hyperedges in \( H_c \) is larger than \( \gamma \), 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 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 \( i \)-th level bipartitioning solution \( S^i_{e_T} \) on the tree \( e_T \), the balance constraint for block \( V_0 \) in the bipartitioning solution \( S(V_0, V_1) \) is: \((|1/K| - \varepsilon) \cdot W \leq \sum_{v \in V_0} v \leq (|1/K| + \varepsilon) \cdot W \).

After obtaining the \( i \)-th bipartitioning solution \( S(V_0, V_1) \), we mark all the vertices in \( V_0 \) 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 \( e_T \).

A. Cutsize Comparison

We run hMETIS and KaHyPar with their respective default parameter settings. 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 $hMETIS_t$. We compare K-SpecPart against $hMETIS_t$ and KaHyPar$_{avg}$; 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 $\sim 5 \times$ the walltime of K-SpecPart. In all our experiments, we adapt the imbalance factor as $\epsilon = K \times \epsilon$ for KaHyPar.

ISPD98 Benchmarks With Unit Weights: Comparisons with $hMETIS_t$ and KaHyPar$_{avg}$ are presented in Fig. 7. K-SpecPart significantly improves over both $hMETIS_t$ and KaHyPar$_{avg}$ 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 ($\sim 2\%$ 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 $hMETIS_t$ and KaHyPar, while Table IV provides comparisons with $hMETIS_{avg}$ and KaHyPar$_{avg}$. We see that K-SpecPart tends to

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

\[\text{The hint for K-SpecPart comes from } hMETIS_t. \text{ Because } hMETIS_{avg} \text{ is the average (over 50 samples) cutsize of } hMETIS_{20}, \text{ K-SpecPart may perform worse than } hMETIS_{avg} \text{ in some testcases.}\]
TABLE III
COMPARISON OF hMETIS (hMETISavg), KaHyPar (KaHyParavg), AND K-SpecPart ON ISPD98 BENCHMARKS WITH UNIT VERTEX WEIGHTS FOR MULTIWAY PARTITIONING WITH NUMBER OF BLOCKS (K) = 2, 3, AND IMBALANCE FACTOR (ε) = 2%

<table>
<thead>
<tr>
<th>Statistics</th>
<th>K = 2</th>
<th>K = 3</th>
<th>K = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>Vثر</td>
<td>Kثر</td>
<td>hMETISthr</td>
</tr>
<tr>
<td>IBM01</td>
<td>12752</td>
<td>1411</td>
<td>203.0</td>
</tr>
<tr>
<td>IBM02</td>
<td>15654</td>
<td>1648</td>
<td>304.5</td>
</tr>
<tr>
<td>IBM03</td>
<td>26136</td>
<td>2470</td>
<td>583.8</td>
</tr>
<tr>
<td>IBM04</td>
<td>27097</td>
<td>3170</td>
<td>583.1</td>
</tr>
<tr>
<td>IBM05</td>
<td>23478</td>
<td>2845</td>
<td>718.2</td>
</tr>
<tr>
<td>IBM06</td>
<td>47296</td>
<td>4817</td>
<td>910.9</td>
</tr>
<tr>
<td>IBM07</td>
<td>59499</td>
<td>5770</td>
<td>976.1</td>
</tr>
<tr>
<td>IBM08</td>
<td>63890</td>
<td>6939</td>
<td>1260.7</td>
</tr>
<tr>
<td>IBM09</td>
<td>70558</td>
<td>8145</td>
<td>1852.6</td>
</tr>
<tr>
<td>IBM10</td>
<td>71076</td>
<td>7774</td>
<td>1956.3</td>
</tr>
<tr>
<td>IBM11</td>
<td>84139</td>
<td>9956</td>
<td>3441.3</td>
</tr>
<tr>
<td>IBM12</td>
<td>142053</td>
<td>15766</td>
<td>3871.3</td>
</tr>
</tbody>
</table>

TABLE IV
COMPARISON OF hMETIS (hMETISavg), KaHyPar (KaHyParavg), AND K-SpecPart ON ISPD98 BENCHMARKS WITH ACTUAL WEIGHTS FOR MULTIWAY PARTITIONING WITH NUMBER OF BLOCKS (K) = 2, 3, AND IMBALANCE FACTOR (ε) = 2%

<table>
<thead>
<tr>
<th>Statistics</th>
<th>K = 2</th>
<th>K = 3</th>
<th>K = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark</td>
<td>Vثر</td>
<td>Kثر</td>
<td>hMETISthr</td>
</tr>
<tr>
<td>IBM11</td>
<td>142053</td>
<td>15766</td>
<td>3871.3</td>
</tr>
</tbody>
</table>

yield more significant improvements relative to the unit-weight case. For example, for IBM11w, K-SpecPart generates almost 27% improvement over hMETIS and KaHyPar for K = 2. We notice similar improvements for K > 2 as seen on IBM04w for K = 3 and IBM10w 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, which does not include Titan23 in its experimental study [32]. Fig. 9 compares K-SpecPart against hMETIS5, while Table V compares with hMETISavg. Although the K-SpecPart runtime is still similar to hMETIS5, 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 hMETIS5 and hMETISavg. On gsm_switch in particular, K-SpecPart achieves more than 50% better cutszie.

B. Runtime Remarks

Our current Julia implementation of K-SpecPart has a walltime approximately 5x 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 \( (\varepsilon) = 2\% \). \( \Delta hMETIS^{-t} \) and \( \Delta K-SPEC^{-t} \), respectively, denote the runtimes of \( hMETIS \) and \( K-SpecPart \) in seconds.

| Benchmark          | \( |V| \) | \( |E| \) | \( hMETIS_{avg} \) | \( K-SPEC^{-t} \) | \( hMETIS_{avg}^{-t} \) | \( K-SPEC^{-t} \) | \( hMETIS_{avg}^{-t} \) | \( K-SPEC^{-t} \) | \( hMETIS_{avg}^{-t} \) | \( K-SPEC^{-t} \) |
|--------------------|--------|--------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| \( sparcl1 \_core \) | 91976  | 92527  | 982.2         | 777            | 7.2            | 46.4          | 2187.9        | 1889          | 13.8          | 36.7          |
| \( neuron \)       | 92290  | 125035 | 243.0         | 244            | 7.4            | 37.2          | 371.6         | 396           | 12.6          | 60.5          |
| \( stereo \_vision \) | 94050  | 127080 | 171.0         | 169            | 7.5            | 41.6          | 332.7         | 336           | 13.2          | 64.1          |
| \( deoxy \)       | 111221 | 135657 | 376.8         | 372            | 10.3           | 54.6          | 526.5         | 535           | 14.7          | 72.2          |
| \( SLAM \_spheric \) | 111115 | 142048 | 1091.0        | 1061           | 10.9           | 56.2          | 2797.1        | 2720          | 21.4          | 827.7         |
| \( cholesky \_mc \) | 113250 | 134948 | 282.0         | 282            | 17.7           | 76.9          | 886.5         | 864           | 20.5          | 89.6          |
| \( segmentation \) | 134295 | 179051 | 120.1         | 120            | 19.7           | 92.2          | 476.1         | 453           | 22.2          | 104.7         |
| \( bitonic \_mesh \) | 204094 | 233328 | 383.2         | 384            | 18.2           | 98.1          | 839.0         | 983           | 25.9          | 127.2         |
| \( dart \)        | 207254 | 223901 | 537.0         | 461            | 11.0           | 84.8          | 1299.1        | 1243          | 20.4          | 132.5         |
| \( openCV \)      | 217422 | 284108 | 433.4         | 434            | 22.1           | 108.5         | 501.8         | 525           | 28.9          | 142.6         |
| \( sparc2 \_core \) | 300109 | 320663 | 1220.7        | 1188           | 31.3           | 147.4         | 2827.2        | 2249          | 60.9          | 176.3         |
| \( gsm \_switch \) | 342690 | 507821 | 1323.5        | 1333           | 28.9           | 137.2         | 4148.6        | 4694          | 56.2          | 268.5         |
| \( met \_nec \)   | 547543 | 707994 | 839.6         | 633            | 88.1           | 186.7         | 1164.3        | 1215          | 61.3          | 412.5         |
| \( LT \_Network \) | 574372 | 699479 | 3334.3        | 3363           | 11.1           | 238.6         | 4549.5        | 4548          | 81.5          | 388.4         |
| \( L1 \_Network \) | 465470 | 726999 | 524.0         | 524            | 33.2           | 2065.9        | 787.1         | 882           | 80.6          | 413.2         |
| \( sparcl1 \_chip \) | 820886 | 821274 | 914.2         | 876            | 71.1           | 364.6         | 1453.4        | 1404          | 97.4          | 491.2         |
| \( directff \)    | 931275 | 137477 | 602.6         | 515            | 75.9           | 378.1         | 728.2         | 762           | 103.6         | 512.8         |
| \( bitcon \_inner \) | 1089384| 1448151| 1514.1        | 1662           | 80.9           | 398.7         | 1944.8        | 1917          | 127.2         | 626.8         |

<table>
<thead>
<tr>
<th>Statistics</th>
<th>( K = 2 )</th>
<th>( K = 3 )</th>
<th>( K = 4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( hMETIS )</td>
<td>( \Delta hMETIS^{-t} )</td>
<td>( \Delta K-SPEC^{-t} )</td>
<td>( \Delta hMETIS_{avg}^{-t} )</td>
</tr>
<tr>
<td>( K-SPEC )</td>
<td>( hMETIS_{avg}^{-t} )</td>
<td>( K-SPEC^{-t} )</td>
<td>( hMETIS_{avg}^{-t} )</td>
</tr>
</tbody>
</table>

The cutsize and runtime of \( hMETIS \) are normalized to \( hMETIS_{avg} \). The runtime is normalized to \( hMETIS_{avg} \). The runtime overhead grows linearly for large values of \( K \) and for \( K = 128 \), \( K-SpecPart \) achieves \( 2\% \) improvement with over 60\% runtime overhead over \( hMETIS \). 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 observed 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 \( sparcl1 \_core \), \( cholesky \_mc \), \( segmentation \), \( denoise \), \( gsm \_switch \), and \( directff \), for \( K = 2 \) and \( \varepsilon = 5\% \). With respect to \( \gamma \), 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 (\( \beta \)) 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.

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.

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 ($\sim 1\%$) cutsizes with significantly faster ($\sim 10\times$) runtime compared to K-SpecPart without LDA. However, for the case of bipartitioning ($K = 2$) we do not observe any significant difference in cutsizes when employing LDA.

Fig. 11. Normalized cutsizes 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.
balance constraints associated with the bipartitioning algorithm (Section V-C) with a balanced tree partitioning baseline, $F$. VILE Versus Recursive Balanced Tree Partitioning

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 $i$th ($1 \leq i \leq K - 1$) level partitioning, the balance constraints associated with the bipartitioning solution $S(V_{0i}, V_{i1})$ are

$$
\left( \frac{1}{K} - \epsilon \right) W \leq \sum_{v \in V_{0i}} w_{v} \leq \left( \frac{1}{K} + \epsilon \right) W
$$

(4)

$$
\sum_{v \in V_{0i}} w_{v} \geq \left( \sum_{v \in V_{0i}} w_{v} \right) - (K - i) \left( \frac{1}{K} + \epsilon \right) W.
$$

(5)

After obtaining the bipartitioning solution $S(V_{0i}, V_{i1})$, we proceed with the $(i + 1)$th level partitioning. 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.

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 $hMETIS$ $\psi$ times with different random seeds, and report the best cutsize Multistart-$hMETIS$. Here, $\psi$ 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 $\psi$, 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
gsxswi4ch for K = 2 and K = 4, respectively. 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.

REFERENCES


Ismail Bustany (Member, IEEE) received the M.S. and Ph.D. degrees in ECECS 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.