SpecPart: A Supervised Spectral Framework for Hypergraph Partitioning Solution Improvement

Ismail Bustany  
Advanced Micro Devices  
San Jose, CA, USA  
ismail.bustany@gmail.com

Andrew B. Kahng  
University of California San Diego  
La Jolla, CA, USA  
abk@ucsd.edu

Joannis Koutis  
New Jersey Institute of Technology  
Newark, NJ, USA  
ikoutis@njit.edu

Bodhisatta Pramanik  
Iowa State University  
Ames, IA, USA  
bodhi91@iastate.edu

Zhiang Wang  
University of California San Diego  
La Jolla, CA, USA  
zhw033@ucsd.edu

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 refinements on each level of the hierarchy. Multilevel partitioners are subject to two limitations: (i) Hypergraph coarsening processes rely on local neighborhood structure without fully considering the global structure of the hypergraph. (ii) Refinement heuristics can stagnate on local minima. In this paper, we describe SpecPart, the first supervised spectral framework that directly tackles these two limitations. SpecPart solves a generalized eigenvalue problem that captures the balanced partitioning objective and global hypergraph structure in a low-dimensional vertex embedding while leveraging initial high-quality solutions from multilevel partitioners as hints. SpecPart further constructs a family of trees from the vertex embedding and partitions them with a tree-sweeping algorithm. Then, a novel overlay of multiple tree-based partitioning solutions, followed by lifting to a coarsened hypergraph, where an ILP partitioning instance is solved to alleviate local stagnation. We have validated SpecPart on multiple sets of benchmarks. Experimental results show that for some benchmarks, our SpecPart can substantially improve the cutsize by more than 50% with respect to the best published solutions obtained with leading partitioners hMETIS and KaHyPar.

CCS CONCEPTS
• Hardware → Physical design (EDA); • Theory of computation → Design and analysis of algorithms.

KEYWORDS

Hypergraph Partitioning, Supervised Spectral Partitioning

ACM Reference Format:

1 INTRODUCTION

Hypergraphs are a generalization of graphs where hyperedges, the counterpart of edges in a graph, can connect more than two vertices. A fundamental NP-hard problem related to hypergraphs is to partition all the vertices into balanced blocks such that each block has bounded size and the cutsize, i.e., the number of spanning multiple blocks, is minimized. This balanced hypergraph partitioning has been a well-studied, fundamental combinatorial optimization problem with application throughout VLSI CAD. Balanced partitioning can also enable efficient distributed computations when solving area-constrained hypergraph optimization problems. Many hypergraph partitioners have been proposed over the past decades. State-of-the-art hypergraph partitioners, including MLPart [21], PaToH [9], KaHyPar [24], and hMETIS [6], usually follow the multilevel paradigm [6]. The multilevel paradigm constructs a hierarchy of progressively coarser hypergraphs using local clustering heuristics [24], partitions the coarsest hypergraph, then uncoarsens, and refines the partitioning solution at each level of the hierarchy [11, 14].

Multilevel partitioners are powerful but subject to two limitations. The first stems from the propensity of partition refinement heuristics to become trapped on local minima that persist through levels in the hierarchy. It is reasonable to hypothesize that any given solution obtained by a multilevel partitioner is ‘in the vicinity’ of potentially much better solutions. However, finding such solutions may require some type of global understanding of the hypergraph. That brings us to the second limitation of the multilevel paradigm: the coarsening phase and refinement decisions are usually based on local structure and greedy computational objectives, hence the global structure of the hypergraph is not explicitly taken into account.

We thus consider a cut obtained by a multilevel partitioner as a hint for a better solution and set out to design a solution improvement method that leverages the hint while using global structural information. This kind of global structure of the hypergraph can be exposed by spectral algorithms [26–29] based on the well-known Cheeger inequality [31]. Spectral partitioning algorithms have been generalized by Cucuringu et al. [1] to supervised partitioning instances, e.g., instances where a hint is available. More specifically, the algorithm of [1] formulates supervised partitioning as a generalized eigenvalue problem satisfying a generalized Cheeger inequality. This suggests a clear direction towards obtaining improved partitioning solutions.

We propose SpecPart, the first supervised spectral framework for hypergraph partitioning solution improvement. In this work, we focus on the bipartitioning problem which is often used as a subroutine in k-way partitioners.
Our contributions include:

- A novel method that incorporates pre-computed hint solutions into a generalized eigenvalue problem. The computed eigenvectors yield high-quality vertex embeddings that are superior to those obtained without supervision. Importantly, our carefully engineered code yields a practically fast implementation. [Section 4.1].
- A novel algorithm for converting a vertex embedding into a partitioning solution. The algorithm uses the embedding to construct a family of trees that in some sense distill the cut structure of the hypergraph. Then, fast algorithms can be used on the tree to explore a large space of candidate solutions from which the best can be picked. [Section 4.2].
- A novel cut overlay method for improving a small pool of initial solutions. Specifically, we compute clusters by removing from the hypergraph the union of the hyperedges cut by any of the solutions in the pool. The size of the clustered hypergraph is small, but it nearly always contains an improved solution that can often be computed optimally using an ILP formulation. [Section 3].
- We have validated SpecPart on multiple benchmark sets (ISPD98 VLSI Circuit Benchmark Suite [4], Titan23 [8] and Industrial benchmarks from a leading FPGA company) with state-of-the-art partitioners (hMETIS [6] and KaHyPar [24]). Experimental results show that for some benchmarks, our SpecPart can substantially improve the cutsize by more than 50% with respect to hMETIS and/or KaHyPar. [Section 5.1].
- We apply autotuning to tune the hyperparameters of existing partitioners and generate a better initial solution for SpecPart. Experiments suggest that the autotuning-based SpecPart can further push the leaderboard for these benchmarks. [Section 5.3].

SpecPart draws strength from recent theoretical and algorithmic progress [1, 18, 20, 22]. In particular, a careful choice of the numerical solvers enables a very efficient implementation. Moreover, SpecPart’s capacity to include supervision information makes it potentially even more powerful in industrial pipelines. 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.

2 PRELIMINARIES

2.1 Hypergraph Partitioning Formulation

In a hypergraph $H(V, E)$, $V$ is a set of vertices with each vertex $v \in V$ associated with a weight $w_v$, and $E$ is a set of hyperedges where a hyperedge $e \in E$ is a subset of $V$. Each hyperedge $e$ can be also associated with a weight $w_e$. Given a positive integer $k$ ($k \geq 2$) and a positive real number $\epsilon$ ($\epsilon \leq \frac{1}{k}$), the $k$-way balanced hypergraph partitioning problem is to partition $V$ into $k$ disjoint blocks $S = \{V_0, V_1, ..., V_{k-1}\}$ such that (letting $W = \sum_{v \in V} w_v$):

- $(1/k - \epsilon)W \leq \sum_{v \in V_i} w_v \leq (1/k + \epsilon)W$, for $0 \leq i \leq k-1$
- \text{cutsize}_H(S) = \sum_{\{v \in V_i, w \notin V_i\} \forall i} w

Here $k$ is the number of blocks in the partitioning solution, $\epsilon$ is the allowed imbalance between blocks, $V_i$ is a partition block and we say that $S$ is an $\epsilon$-balanced partitioning solution.

2.2 Laplacians, Cuts and Eigenvectors

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

$$x^TLx = \text{cutsize}_G(S).$$

Let us now consider an example of how balanced graph bipartitioning relates to spectral methods. Let $K$ be the Laplacian of a complete unweighted graph on $V$. Using expression (1), we have

$$R(x) = \frac{x^TLx}{x^TKx} = \frac{\text{cutsize}_G(S)}{|S|}.\frac{|V - S|}{|V|}.$$ 

Minimizing $R(x)$ over 0-1 vectors $x$ incentivizes a small cutsize($S$) with a simultaneous balance between $|S|$ and $|V - S|$, hence $R(x)$ can be viewed as a proxy for the balanced partitioning objective. We can relax the problem over the real vectors $x$ constrained to be orthogonal to the common null space of $L$ and $K$. It is well understood that the minimum is achieved by the first non-trivial eigenvector of the problem $Lx = \lambda Kx$.

2.3 Spectral Embeddings and Partitioning

Spectral graph partitioning algorithms embed the vertices of an input graph $G$ into a $m$-dimensional space and then cluster the points in this geometric space. The vertex embedding comes from the computation of $m$ non-trivial eigenvectors of an appropriate eigenvalue problem involving the Laplacian $L_G$ of the graph $G$. More specifically, if $X \in \mathbb{R}^{m \times |V|}$ is the matrix containing $m$ (column) eigenvectors, then row $X_u$ of $X$ is the embedding of vertex $u$.

Spectral algorithms have also been used for hypergraph partitioning. In this context, the hypergraph $H$ is first transformed to a corresponding graph $G$, and then the spectral embedding is computed using $L_G$. For example, the eigenvalue problem solved in [26] is

$$L_Gx = \lambda D_wx$$

where $D_w$ is the diagonal matrix containing positive vertex weights. In this paper we solve the more general problem

$$L_Gx = \lambda Bx$$

where $B$ is also a graph Laplacian. In practical instances, hypergraphs are ‘essentially’ connected with possibly a few outstanding vertices and edges that can be processed separately. Thus, since $G$ can be considered connected, the problem 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 $B$ [19]. 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 $B$.

<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>$H_C(V_c, E_c)$</td>
<td>Clustered hypergraph $H_C$ where each vertex $v_i \in V_c$ 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>$T(V, E_T)$</td>
<td>Tree $T$ with vertices $V$ and edges $E_T$</td>
</tr>
<tr>
<td>$u, v$</td>
<td>Vertices in $V$</td>
</tr>
<tr>
<td>$e_{uv}$</td>
<td>Edge or hyperedge connecting $u$ and $v$</td>
</tr>
<tr>
<td>$e_T$</td>
<td>Edge of tree $T$</td>
</tr>
<tr>
<td>$w_v, w_e$</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, $S = {V_0, V_1, ..., V_{k-1}}$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Allowed imbalance (1-49) between blocks in $S$</td>
</tr>
<tr>
<td>$\text{cut}(S)$</td>
<td>Cut of $S$, $\text{cut}(S) = {e \notin V_i \text{ for any } i}$</td>
</tr>
<tr>
<td>$\text{cutSize}_G(S)$</td>
<td>Cutsizes of $S$ on (hyper)graph $H$.</td>
</tr>
<tr>
<td>$\text{ISHP}$</td>
<td>Iterative Supervised Spectral Hypergraph Partitioning</td>
</tr>
</tbody>
</table>

Table 1: Notation
2.4 ILP for Hypergraph Partitioning

Hypergraph partitioning can be solved optimally by casting the problem as an integer linear program (ILP) [23]. 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 $e$, and require that:

- $x_{v,i} = 1$ if $v \in V_i$
- $y_{e,i} = 1$ if $e \subseteq V_i$

We then define the following constraints for each $i \in \{0,k-1\}$:

- $(1/k - \epsilon)W \leq \sum_{v \in V_i} w_v x_{v,i} \leq (1/k + \epsilon)W$
- $\sum_{j=0}^{i-1} x_{v,j} = 1$ for $v \in V$
- $y_{e,i} \leq x_{v,i}$ for each $v \in E$, and each $e \in e$

where $W = \sum_{v \in V} w_v$. The objective is:

$$\text{Maximize} \sum_{e \in E} \sum_{0 \leq i \leq k-1} w_e y_{e,i}.$$ 

3 SPECPART: AN OVERVIEW

The architecture of our SpecPart framework is shown in Figure 1. The input is a hypergraph $H(V,E)$, an initial partitioning solution $S_{init}$, and $\epsilon$, the allowed imbalance between blocks in a partitioning solution. The output is an improved partitioning solution $S_{out}$. Here the initial partitioning solution $S_{init}$ can come from any source, including available open-source partitioners.1

![Figure 1: Overview of the SpecPart framework.](image)

The SpecPart framework consists of two major components:

1. **Iterative Supervised Spectral Hypergraph Partitioning.**
   ISSHP constitutes the fundamental algorithmic core of SpecPart. The initial solution $S_{init}$ is incorporated into a generalized eigenvalue problem in order to generate a vertex embedding (Section 4.1). With the hint from $S = S_{init}$, the vertex embedding from the generalized eigenvalue problem is of higher quality relative to that obtained from the standard eigenvalue problem, as illustrated in Figure 2. The embedding is used to compute a family of trees that — in some sense

   - **Cut-Overlay Clustering and Optimal-Attempt Partitioning.**
     In the course of its iterations, ISSHP generates a collection of different solutions. We select the $\delta$ best solutions, denoted as "candidate partitioning solutions" in Figure 1.

     **Cut-Overlay clustering.** Let $E_1, \ldots, E_S \subseteq E$ be the sets of hyperedges cut in the $\delta$ candidate 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 $H_{c}(V_c, E_c)$. A solution on $H_{c}$ can be "lifted" to $H$, and by construction it is guaranteed that $H_{c}$ contains a solution which is at least as good as the best among the cuts $E_i$.

     **Optimal-Attempt Partitioning.** While one would expect that $H_{c}$ has not many more than $2^d$ vertices, empirically we often observe hundreds of vertices and hyperedges (e.g., even for $\delta = 5$). Given such a size for $H_{c}$, we would also expect that it is infeasible to run an ILP-based partitioner on it. Remarkably, due to the special generative process that yields $H_{c}$, it is often the case that the ILP computes within stringent walltime a solution that is better than any of the $\delta$ solutions in the pool. In our current implementation, we include a parameter $\gamma$; in the case when the number of hyperedges in $H_{c}$ is larger than $\gamma$ (default value of $\gamma$ is 300) we run $kMETIS$ on $H_{c}$.

4 THE ISSHP ALGORITHM

The Iterative Supervised Spectral Hypergraph Partitioning (ISSHP) process is described in Algorithm 1, with pointers to subsequent sections that discuss the details.

4.1 Vertex Embedding Generation

In order to generate a vertex embedding, we need to construct the generalized eigenvalue problem and compute the first $m$ nontrivial eigenvectors. Here $m$ is the number of eigenvectors that we use, which is set to 2 by default.

4.1.1 Clique Expansion Graph: We define the clique expansion graph $G$ of the hypergraph $H$, as a sum, i.e., superposition, of weighted cliques; the clique corresponding to edge $e \in E$ has the same vertices as $e$ and edge weights $\frac{w_e}{\sqrt{|e|}}$. Graph $G$ has size

$$\sum_{e \in E} |e|^2$$

where $|e|$ is the size of hyperedge $e$. This is usually quite large relative to the input size $|V| = \sum_{e \in E} |e|$. For this reason we only construct a function $f_{e}$, that evaluates matrix-vector products of

<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>$\tau$</td>
<td>Number of trees ($\tau = 8$)</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Number of best solutions ($\delta = 5$)</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Number of iterations of ISSHP ($\beta = 2$)</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>Number of random cycles ($\zeta = 2$)</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Threshold of number of hyperedges ($\gamma = 300$)</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Number of iterations of eigenvector solver ($\theta = 80$)</td>
</tr>
</tbody>
</table>

Table 2: Parameters of SpecPart framework.
We have
\[ \text{where} \]
\[ 1 = \text{vector of vertex weights}. \]
\[ \text{We have the identity} \]
\[ \text{Construct Laplacian} \]
\[ B_{\text{base}} \text{ of weight-balance graph (4.1.2)} \]
\[ \text{for } i = 0; i < \beta; i + \text{do} \]
\[ \text{Construct Laplacian } B_{S_{\text{best}}} \text{ based on hint } S_{\text{best}} (4.1.3) \]
\[ \text{Let } B = B_{\text{base}} + B_{S_{\text{best}}} \]
\[ \text{Solve the generalized eigenvalue problem } L_G x = \lambda B x \text{ to compute } m \text{ nontrivial eigenvectors (4.1.5)} \]
\[ \text{Construct a family of trees } \{ T_i \} \text{ based on computed eigenvectors (4.2)} \]
\[ \text{Generate candidate solutions } \{ S_{i_j} \} \text{ by running tree-sweep and } \text{METIS on trees } \{ T_i \} (4.3) \]
\[ \text{Set } S_{\text{best}} \text{ to the best partitioning solution in } \{ S_{i_j} \} \]
\[ \text{end} \]
\[ \text{Construct } \{ S_{i_j} \} \text{ by picking the best } \delta \text{ solutions from } \{ \{ S_{i_j} \} \} \]
\[ \text{return } \{ S_{i_j} \} \]
\]
\[ \text{Algorithm 1: ISSHP: Iterative Supervised Spectral Hypergraph Partitioning.} \]
\[ \text{Input: Hypergraph } H(V,E), \text{ initial partitioning solution } S_{\text{best}} \]
\[ \text{Output: Candidate partitioning solutions } \{ S_{i_j} \} \]
\[ \text{Construct the Laplacian } L_G \text{ of the clique expansion for } H \]
\[ \text{Generate candidate solutions } \{ S_{i_j} \} \text{ by running tree-sweep and } \text{METIS on trees } \{ T_i \} \]
\[ \text{Set } S_{\text{best}} \text{ to the best partitioning solution in } \{ S_{i_j} \} \]
\[ \text{return } \{ S_{i_j} \} \]
\]
\[ \text{The form } L_G x, \text{ where } L_G \text{ is the Laplacian of } G, \text{ which is all we need to perform the eigenvector computation. In all places where Algorithm 1 mentions the construction of any Laplacian, we construct the equivalent function for evaluating matrix-vector products. This is further justified in Section 4.1.5. The function } f_L \text{ is an application of the following identity that is based on expressing } L_G \text{ as a sum of Laplacians of cliques:} \]
\[ L_G x = \sum_{e \in E} \frac{1}{|e| - 1} \left( x - \frac{x^T e}{e^T e} \right) \cdot 1_e. \]  

(4)

where } 1_e \text{ is the 1-0 vector with 1s in the entries corresponding to the vertices in } e. \text{ By exploiting the sparsity in } 1_e, \text{ the product is run in } O(|V|) \text{ time.} \]

4.1.2 \text{ Weight-Balance Graph:} \text{ The weight-balance graph } G_w \text{ is a complete weighted graph used to capture arbitrary vertex weights and incentivize balanced cuts, as we elaborate in Section 4.1.4. } G_w \text{ has the same vertices as hypergraph } H, \text{ and edges of weight } \omega_u \cdot \omega_v \text{ between any two vertices } u \text{ and } v. \text{ Let } \omega_{V_i} \text{ be the weight of block } V_i \text{ in a partitioning solution } S, \text{ i.e.,} \]
\[ \omega_{V_i} = \sum_{v \in V_i} \omega_v. \]

(5)

We have
\[ \omega_{V_0} \cdot \omega_{V_1} = \sum_{v \in V_0} \omega_v \cdot \sum_{v \in V_1} \omega_v = \sum_{v \in V_0, u \in V_1} \omega_v \cdot \omega_u \]
\[ = \sum_{v \in V_0, u \in V_1} \omega_{eu} = \text{cutsizel}_0(S) \]

(6)

We now discuss how to compute matrix-vector products with the Laplacian matrix of } G_w, \text{ which we denote by } B_{\text{base}}. \text{ Let } w \text{ be the vector of vertex weights. We have the identity} \]
\[ B_{\text{base}} x = w \odot x - \frac{x^T 1}{1^T 1} \cdot w, \]
\[ \text{where } 1 \text{ is the all-ones vector and } \odot \text{ denotes the Hadamard product. Clearly, this can be carried out in time } O(|V|). \]

In general any vector } x \text{ can be written in the form } x = y + c1, \text{ where } y^T 1 = 0. \text{ Substituting this decomposition of } x \text{ into the above equation, we get that } B_{\text{base}} x = w \odot y. \text{ In other words, } B_{\text{base}} \text{ acts like a diagonal matrix on } y \text{ and nullifies the constant component of } x. \]

4.1.3 \text{ Hint Graph:} \text{ The hint graph } G_h \text{ is a complete bipartite graph on the two vertex sets } V_0 \text{ and } V_1 \text{ defined by the hint solution } S_{\text{best}}. \text{ It is used to incentivize the computation of cuts that are similar to } S_{\text{best}}, \text{ as elaborated in Section 4.1.4. If } B_{\text{best}} \text{ denotes the Laplacian of the hint graph,} \]
\[ B_{\text{best}} x = (x - \frac{x^T 1}{1^T 1} 1) - (x - \frac{x^T 1}{1^T V_0} V_0 - 1) (x - \frac{x^T 1}{1^T V_1} V_1) \]

(8)

where } 1_{V_i} \text{ denotes the 1-0 vector with 1s in entries corresponding to the vertices in } V_i. \text{ By exploiting the sparsity in } 1_{V_i}, \text{ the product is implemented in } O(|V|) \text{ time.} \]

4.1.4 \text{ Intuition on the constructed graphs:} \text{ We solve the generalized eigenvalue problem } L_G x = \lambda B x, \text{ where } B = B_{\text{base}} + B_{\text{best}}. \text{ From the discussion in Section 2.2 recall that the eigenvalue problem is directly related to solving} \]
\[ \min_x R(x) = \min_x x^T L_G x = \min_x -x^T B_{\text{base}} x + x^T B_{\text{best}} x \]

(9)

over the real vectors } x. \text{ Recall also that this is a relaxation of the minimization problem over 0-1 cut indicator vectors. Let } X_S \text{ be the indicator vector for some set } S \subset V. \text{ Then, using Equation (1) we have:} \]
\[ x^T L_G X_S = \text{cutsizel}_S(S) \text{ which is a proxy for } \text{cutsizel}_H(S). \text{ Thus, the numerator incentivizes smaller cuts in } H. \]
\[ x^T B_{\text{base}} X_S = \text{cutsizel}_w(S). \text{ By Equation (6), this is equal to } \omega_S \cdot \omega_{V_{\neg S}}, \text{ where } \omega_S \text{ is the total weight of the vertices in } S. \text{ Thus the denominator incentivizes a large } \omega_S \cdot \omega_{V_{\neg S}}, \text{ which implies balance}. \]
\[ x^T B_{\text{best}} X_S \text{ is maximized when all edges of } G_h \text{ are cut, thus the denominator incentivizes cutting many edges that are also cut by the hint.} \]

\[ \text{Figure 3: Graphs used in ISSHP, Algorithm 1.} \]

\[ \text{Hypergraph} \]

\[ \text{Clique expansion graph, } G \]

\[ \text{Weight-balance complete graph, } G_w \]

\[ \text{Hint graph (complete bipartite graph), } G_h \]
4.1.5 Computation: We solve the generalized eigenvalue problem \( L_G x = \lambda B x \) using the preconditioned eigensolver LOBPCG [13]. Due to its iterative nature, LOBPCG does not require explicit matrices \( L_G \) and \( B \), but merely functions that evaluate matrix-vector products with them. For fast computation, the solver can utilize a preconditioner for \( L_G \), also in an implicit functional form. To compute the preconditioner we first obtain an explicit graph \( \hat{G} \) that is spectrally-similar with \( G \) and has size at most \( 9|\mathcal{F}| \), where \( |\mathcal{F}| = \sum_{e \in E} |e| \). More specifically, we build \( \hat{G} \) by replacing every hyperedge \( e \in H \) with the sum of uniformly weighted random cycles on the vertices \( V_e \) of \( e \). This is an essentially optimal sparse spectral approximation for the clique on \( V_e \).\(^2\) Since \( G \) is a sum of cliques, and \( \hat{G} \) is a sum of tight spectral approximations of cliques, standard graph support theory [38] implies that \( \hat{G} \) is a tight spectral approximation for \( G \). Finally, we compute a preconditioner of \( L_G \) using the CMG algorithm [20]; by transitivity [38], it is also a preconditioner for \( L_G \).

4.2 Tree Construction

After solving the generalized eigenvalue problem, we have a matrix \( X \) with \( 4.2 \) Tree Construction.

4.2.1 Paths. We first use a standard linear ordering algorithm [39] to obtain a path graph for each eigenvector \( x_i \), by sorting the vertices in \( V \) based on \( x_i \) in non-decreasing order and connecting the sorted vertices in that order. The path graph is implicit in the proof of the Cheeger inequality [31] which shows that a relatively good cut of the graph into two parts can be found by sweeping over the \( n-1 \) tree cuts. We thus use the \( m \) eigenvectors to construct \( m \) path graphs in total. These path graphs naturally arrange together vertices with similar global positioning, but neighboring nodes in the path are not necessarily neighbors in the original hypergraph \( H \). That means the local neighborhood information is not fully preserved in the paths.

4.2.2 Clique Expansion Spanning Trees. To address the issue of preserving local information, we work with a weighted graph that reflects both the connectivity of the hypergraph and the global information contained in the embedding, adapting an idea that has been used in work on \( k \)-way Cheeger inequalities [22]. Concretely, we form a graph \( \hat{G} \) by replacing every edge \( e \) of \( H \) with a sum of \( \zeta \) cycles (as discussed also in Section 4.1.5). Suppose that \( Y \in \mathbb{R}^{|V| \times |\mathcal{F}|} \) is an embedding matrix and 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 length of each edge \( e_u \in \hat{G} \) to \( ||Y_u - Y_v||_2 \), i.e., equal to the Euclidean distance between the two vertices in the embedding. We will be constructing spanning trees of \( \hat{G}_Y \).

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 a Low Stretch Spanning Tree (LSST) of \( \hat{G} \), which by definition means that the length \( l(e_u) \) 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 reorder \( \hat{G}_Y \) using the order induced by sorting the smallest non-trivial eigenvector computed earlier. Empirically, this order has the advantage of producing slightly better LSSTs.

MST: A graph can contain multiple different LSSTs, with each of them approximating to different degrees the length \( l(e_u) \) for any given \( e_u \). It should also be noted that the AKPW algorithm is known to be suboptimal with respect to the approximation factor \( f(|V|) \); more sophisticated algorithms exist but they are far from practical. For these reasons we also compute a Minimum Spanning Tree of \( \hat{G} \). For most weighted graphs an MST can be viewed as an easy-to-compute proxy to an LSST, which potentially has better or complementary distance-preserving properties relative to the tree computed by the AKPW algorithm. We construct the MST using Kruskal’s algorithm [3].

4.2.3 Family of Trees. Recall now that we have a matrix \( X \) of \( m \) eigenvectors. We construct the LSST and MST for the graphs \( \hat{G}_X \) for \( i = 1, \ldots, m \), and for the graph \( \hat{G}_X \). Along with the path graphs, these comprise a family \( \mathcal{F} \) of trees. In total, we have \( \tau = m + 2 \times (m+1) \) trees, comprised of \( m \) path graphs, \( m + 1 \) MSTs, and \( m + 1 \) LSSTs. In the default setting, \( \tau = 8 \).

4.3 Cut Distilling and Partitioning on a Tree

We will use each tree \( T \) in the family of trees to distill the cut structure of \( H \) over \( T \), in the following sense: For any fixed tree \( T = (V, E_T) \), observe that the removal of an edge \( e_T \) of \( T \) yields a partitioning \( S_{e_T} \subset V \) and thus of the original hypergraph \( H \). We would thus like to reweight each edge \( e \in E_T \) with the corresponding cutsizes \( f(S_{e_T}) \).

Computing these edge weights on \( T \) can be done in \( O(|\mathcal{F}| \log |\mathcal{F}|) \) time, via an elaborate algorithm involving the computation of least common ancestors (LCA) on \( T \), in combination with dynamic programming on \( T \). We now describe the main idea by example; the omitted details can be found in our code.

---

\(^2\)The construction relies on theory about the asymptotic properties of random \( d \)-regular expanders (e.g., see [32] or Theorem 4.16 in [33]). For the hyperedges in our context, the near-optimality of our construction can also be verified numerically.

4.3.1 Tree Cuts. As a main idea, consider an arbitrary edge \( e \) of \( T \), and compute the sum of all cuts that include \( e \), i.e., the sum of the labels of vertices that are descendants of \( e \) or \( e \) itself.
KalHyPar on $H$ to generate an initial partitioning solution $S_{init}$, which is leveraged by $hMETIS$ and KalHyPar with their respective default parameter settings. To avoid any possible confusion, we adopt these conventions: SpecPart$_k$ and SpecPart$_s$ represent the cutsizes of SpecPart with the initial solutions generated by $hMETIS$ and KalHyPar respectively; SpecPart represents the best cutsize between SpecPart$_k$ and SpecPart$_s$; and $hMETIS_k$ and KalHyPar$_k$ represent the best cutsizes generated by running $hMETIS$ and KalHyPar times with different random seeds respectively.

5 EXPERIMENTAL VALIDATION

The SpecPart framework is implemented in Julia and we provide both Julia and Python interfaces. We use CPLEX [36] and LOBPCG [17] as our ILP solver and eigenvalue solver respectively. We run all experiments on a server with 56 Xeon E5-2650 L.1.7GHz processors and 256 GB memory. We have compared our framework with two-state-of-the-art hypergraph partitioners\(^3\) ($hMETIS$ [6] and KalHyPar [24]) on three different sets of benchmarks (ISPD98 VLSI Circuit Benchmark Suite [4], Titan23 Suite [8] and Industrial Benchmark Suite from a leading FPGA company).\(^4\) The statistics of these benchmarks are summarized in Table 3, Table 4 and Table 5 respectively.

\(^3\)We do not compare our results with PaToH since it generates weaker cuts compared to $hMETIS$ and KalHyPar on the ISPD98, Titan23 and industrial benchmarks.

\(^4\)We make public with permissive open-source license all partition solutions, scripts and code at [41].

\(^5\)The default values for parameters (\(\delta, \beta, \psi, \xi, \theta, \text{ and } \kappa\)) are shown in Table 1.

5.1 Experimental Results

In this section, we present the experimental results of SpecPart with default parameter settings.\(^5\) We run SpecPart as follows. Given a hypergraph $H$ and an imbalance factor $\epsilon$, we first run $hMETIS$ and/or

\[^{\epsilon \gamma} \text{ This will be we on all edges of } C_e \text{ and } 0 \text{ otherwise, thus correctly accounting for the hyperedge } e \text{ on the intended set of edges } C_e.\]

In order to compute the correct total counts on all tree edges, we iterate over hyperedges, compute their junctions and tally the associated labels. Then, for any tree edge $e_T$, the sum-$\gamma$-$\epsilon_T$ will equal $\text{cutSize}_{\epsilon_T}(S_{e_T})$. These sums can be computed in $O(|V|)$ time, via dynamic programming on $T$. A similar 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.

For a partition $S \subseteq V$ that cuts more than one edge on $T$ we have $\text{cutSize}_{\epsilon_T}(S) \leq \text{cutSize}_{\epsilon_T}(S)$, and owing to the spectral origin of $T$ we hope that $\text{cutSize}_{\epsilon_T}(S)$ can provide a good proxy for $\text{cutSize}_{\epsilon_T}(S)$ the cuts of $H$. Therefore, we use METIS [5] to solve a balanced partitioning problem on the rewighted tree, with the original vertex weights from $H$. This can potentially return a partition $S \subseteq V$ that cuts more than one edge on $T$. In some cases we do get $\text{cutSize}_{\epsilon_T}(S) \leq \text{cutSize}_{\epsilon_T}(S)$, thus further improving the solution.

Table 3: Statistics of ISPD98 VLSI circuit benchmark suite [4]. Best and Best$_w$ represent the best published cutsizes for unit weights and actual weights respectively. SpecPart and SpecPart$_w$ represent the cutsizes generated by SpecPart for unit weights and actual weights respectively.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Statistics</th>
<th>Best</th>
<th>Best$_w$</th>
<th>SpecPart</th>
<th>SpecPart$_w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitonic_mesh</td>
<td>$\gamma$</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
</tr>
<tr>
<td>bitonic_mesh</td>
<td>$\delta$</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
</tr>
<tr>
<td>bitonic_mesh</td>
<td>$\psi$</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
</tr>
<tr>
<td>bitonic_mesh</td>
<td>$\xi$</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
</tr>
<tr>
<td>bitonic_mesh</td>
<td>$\theta$</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
</tr>
<tr>
<td>bitonic_mesh</td>
<td>$\kappa$</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
<td>2051</td>
</tr>
</tbody>
</table>

Table 4: Statistics of Titan23 suite [8]. $hMETIS_k$ and $hMETIS_s$ represent the best cutsizes generated by running $hMETIS$ 5 and 20 times with different random seeds. SpecPart$_k$ represents the cutsizes generated by SpecPart where the hint is obtained from running $hMETIS$ once with default random seed. SpecPart$_s$ represents the cutsizes generated by SpecPart where the hint is the solution corresponding to $hMETIS_s$.

Table 5: Statistics of industrial benchmark suite from a leading FPGA company. $hMETIS_k$ and $hMETIS_s$ represent the best cutsizes generated by running $hMETIS$ 5 and 20 times with different random seeds. SpecPart$_k$ represents the cutsizes generated by SpecPart where the hint is obtained from running KalHyPar once and 10 times respectively. SpecPart$_s$ represents the cutsizes generated by SpecPart where the hint is obtained from running KalHyPar once with default random seed.

5.1.1 ISPD98 benchmarks with unit weights: Here we present results for the ISPD98 VLSI Circuit Benchmark Suite with unit vertex weights. In Table 3 we present the solutions generated by SpecPart and compare them with the corresponding best previously published solutions, with references to the corresponding publications. Figures 5(a)-(b) reports the solutions sizes obtained from SpecPart,
KaHyPar$^5$, and $hMETIS^5$, normalized by the best published solution sizes. While $hMETIS^5$ and (mostly) KaHyPar$^5$ also improve upon these previous solutions, it can be seen that SpecPart generates a significant improvement over both KaHyPar and $hMETIS$ on a number of instances. The reasoning behind picking $hMETIS^5$ is motivated by an “iso” (similar) runtime comparison. For these relatively small instances SpecPart has approximately a 50% runtime overhead over $hMETIS$, which is subject to significant improvement. This illustrates that SpecPart can improve very quickly upon solutions computed under stringent walltime requirements.\footnote{Of course, $hMETIS$ and KaHyPar can be run for more random starts. We include such an experimental study for the larger and more interesting Titan23 and Industrial benchmarks, but we omit them for ISPD98.}

5.1.2 ISPD98 benchmarks with actual weights: We further verify our framework on the vertex-weighted ISPD98 benchmarks. Mirroring the considerations of section 5.1.1, the results are presented in Table 3 and Figures 5(c)-(d). The inclusion of weights makes the problem more general and potentially more difficult. Here, we see a tendency of SpecPart to yield bigger improvements.

The Titan23 and Industrial benchmarks are interesting not just because they are significantly larger than ISPD98, but also because they are generated by different, more modern synthesis processes. They hence provide a ‘test of time’ for $hMETIS$, i.e., the best cut size generated by running $hMETIS$ five times with different random seeds. It can be seen that SpecPart generates significantly better partitioning solutions. The improvements are even more than 50% for benchmarks gsm_switch and denoise. To further examine the performance of SpecPart, we add these experiments:

(i) run $hMETIS$ twenty times with different random seeds and report the best cut size $hMETIS^{20}$; and (ii) set the solution corresponding to $hMETIS^{20}$ as the initial solution to SpecPart$^5$ and generate the cutsize SpecPart$^{20}$. We observe that SpecPart$^5$ is still much better even compared to $hMETIS^{20}$ for almost all the benchmarks. SpecPart$^{20}$ is also better than SpecPart$^5$ for some benchmarks. This suggests that SpecPart can achieve better performance even when standard partitioners are allowed significantly more running time (see also Section 5.3).

5.1.4 Industrial benchmarks from a leading FPGA company: Table 5 presents the results of Industrial Benchmark Suite from a leading FPGA company. Here we present results for imbalance factors ($\epsilon = 2$ and $20$) as per guidance from our industrial collaborator. We do not compare against $hMETIS$ because it fails with a segmentation fault on these benchmarks. KaHyPar remains impractically slow on these large benchmarks; SpecPart adds less than 5% overhead to single benchmarks is very large (more than two hours), too high for any reasonable industrial setting (for more details on runtime see \cite{41}). For this reason we do not compare against KaHyPar. It should be noted that because we could not find previous published results on Titan23, Figure 5 reports cut sizes normalized by those obtained by $hMETIS$, i.e., the best cut size generated by running $hMETIS$ five times with different random seeds. It can be seen that SpecPart generates significantly better partitioning solutions. The improvements are even more than 50% for benchmarks gsm_switch and denoise. To further examine the performance of SpecPart, we add these experiments:

(i) run $hMETIS$ twenty times with different random seeds and report the best cut size $hMETIS^{20}$; and (ii) set the solution corresponding to $hMETIS^{20}$ as the initial solution to SpecPart$^5$ and generate the cutsize SpecPart$^{20}$. We observe that SpecPart$^5$ is still much better even compared to $hMETIS^{20}$ for almost all the benchmarks. SpecPart$^{20}$ is also better than SpecPart$^5$ for some benchmarks. This suggests that SpecPart can achieve better performance even when standard partitioners are allowed significantly more running time (see also Section 5.3).
Comparison on benchmark sparcT2_core (e = 10).

Figure 6: (a): Validation of SpecPart parameters discussed in Section 5.2. (b,c): QoR vs. runtime overhead of Multi-start-hMETIS, Solution-overlay-part, SpecPart, and Autotune-SpecPart. Multi-start-hMETIS = best cutsizes from running hMETIS multiple times with different random seeds. Solution-overlay-part = cutsizes from running Cut-Overlay Clustering and Optimal-Attempt Partitioning directly on candidate solutions. SpecPart = cutsizes from SpecPart when the initial solution is from autotuning of hMETIS with i trials.

run of KaHyPar. Nevertheless, we allow the very large runtime and report a comparison with a single run of KaHyPar and KaHyPar10 in Table 5. It can be seen that even when the hint is based on a fairly expensive computation (a single run of KaHyPar), SpecPart can still generate significant improvements even over KaHyPar10 on some of the benchmarks, especially industrial05 where the improvement is more than 50%. We speculate that the improvements would have been greater if based on a hint provided by hMETIS, which is in general much faster than KaHyPar.

5.2 Validation of Parameters

We now discuss the effect of tuning parameters on SpecPart. The parameters we explore are the number of best solutions (δ), the number of iterations of ISSHP (β), the number of random cycles (ζ), and the threshold of the number of hyperedges in the clustered hypergraph $H_c(γ)$. We define the score value as the average improvement of SpecPart with respect to hMETIS on benchmarks sparcT1_core, cholesky mc, segmentation, denoise, gsm_switch and directcf. When we sweep (i.e., vary the value of) one parameter, the remaining parameters are fixed at their default values (Table 2) and $ɛ$ is set to 20. The results appear in Figure 6(a). Sweeping for γ did not change the score value in our experiments. Using $m > 2$ did not generate further improvement. We also note that using hMETIS instead of ILP for Optimal Attempt Partitioning, worsens the score value by 2.43%. From the results of tuning parameters on SpecPart we establish that our default parameter setting is a local minimum in the hyperparameter search space.

5.3 Effect of ISSHP and Solution Enhancement

5.3.1 Effect of ISSHP: In order to show the effect of ISSHP in the SpecPart framework, we run Cut-Overlay Clustering and Optimal-Attempt Partitioning directly on candidate solutions, which are generated by running hMETIS multiple times with different random seeds. The flow is as follows. (i) We generate candidate solutions $\{S_1, S_2, \ldots, S_n\}$ by running hMETIS $\bar{\gamma}$ times with different random seeds, and report the best cutsizes Multi-start-hMETIS. Here $\bar{\gamma}$ is an integer parameter ranging from one to twenty. (ii) We run Cut-Overlay Clustering and Optimal-Attempt Partitioning directly on the best five solutions from $\{S_1, S_2, \ldots, S_n\}$ and report the cutsizes Solution-overlay-part. For each value of $\bar{\gamma}$, we run the above flow 100 times and report the average result in Figures 6(b,c). We observe that Solution-overlay-part is much better than Multi-start-hMETIS, and that SpecPart generates superior solutions in less runtime compared to Multi-start-hMETIS and Solution-overlay-part. This suggests that ISSHP is an important component of SpecPart.

5.3.2 Solution enhancement: hMETIS has parameters whose setting may significantly impact the quality of generated partitioning solutions. We use Ray [42] 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 [42] is HyperOptSearch. We set the number of trials to five, ten and forty, i.e., Ray will launch five, ten and forty runs of hMETIS with different parameters respectively. We set the number of threads to ten to reduce the runtime. The results appear in Figures 6(b,c). 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 $St_{aut}$, yet we see a further 2% and 4% cutsize improvement from SpecPart for sparcT2_core and gsm_switch, respectively, lending further support to the observation in Section 5.1.3.

6 CONCLUSION AND FUTURE DIRECTIONS

We have proposed SpecPart, the first general supervised framework for hypergraph partitioning solution improvement. Experiments confirm its outstanding performance compared to traditional multilevel partitioners with similar runtime. The code, scripts, and best known solution vectors are available through [41]. SpecPart opens multiple future research directions, with its K-way generalization being a priority. SpecPart can be integrated with the internal levels of multilevel partitioners; producing improved solutions on each level may lead to further improved solutions. We also believe that the Cut-Overlay and Optimal-Attempt Partitioning are of independent interest and amenable to machine learning techniques.

Acknowledgments. Bodhisatta Pramanik thanks Dr. Chris Chu for his early guidance. We thank Dr. Grigor Gasparyan for providing testcases and sharing his thoughts on SpecPart. This work was partially supported by NSF grants CCF-2112665, CCF-2039863 and CCF-1813374, and by DARPA HR0011-18-2-0032.