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

Ioannis 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:
Ismail Bustany, Andrew B. Kahng, Ioannis Koutis, Bodhisatta Pramanik, and Zhiang Wang. 2022. SpecPart: A Supervised Spectral Framework for Hypergraph Partitioning Solution Improvement. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549390

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549390

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 ϵ ($\epsilon \leq \frac{1.0}{k}$), 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
- $cutsize_H(S) = \sum_{\{e | e \not\subseteq V_i \text{ for any } i\}} w_e$ is minimized

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

2.2 Laplacians, Cuts and Eigenvectors

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

$$x^T L x = cutsize_G(S). \tag{1}$$

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

$$R(x) \triangleq \frac{x^T L x}{x^T K x} = \frac{cutsize_G(S)}{|S| \cdot |V - S|}.$$

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 $X \in \mathbb{R}^{|V| \times m}$ 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

$$L_G x = \lambda D_w x \tag{2}$$

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

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

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.

Term	Description
$H(V, E)$	Hypergraph H with vertices V and hyperedges E
$H_c(V_c, E_c)$	Clustered hypergraph H_c where each vertex v_c in V_c corresponds to a group of vertices in $H(V, E)$
$G(V, E)$	Graph G with vertices V and edges E
$\tilde{G}$	Spectral sparsifier of G
$T(V, E_T)$	Tree T with vertices V and edges E_T
u, v	Vertices in V
e_{uv}	Edge or hyperedge connecting u and v
e_T	Edge of tree T
w_v, w_e	Weight of vertex v, or hyperedge e, respectively
k	Number of blocks in a partitioning solution
S	Partitioning solution, $S = \{V_0, V_1, ..., V_{k-1}\}$
ϵ	Allowed imbalance (1-49) between blocks in S
$cut(S)$	Cut of S, $cut(S) = \{e \mid e \notin V_i \text{ for any } i\}$
$cutSize_H(S)$	Cutsize of S on (hyper)graph H.
ISSHP	Iterative Supervised Spectral Hypergraph Partitioning

Table 1: Notation

Parameter	Description (default setting)
m	Number of eigenvectors ($m = 2$)
τ	Number of trees ($\tau = 8$)
δ	Number of best solutions ($\delta = 5$)
β	Number of iterations of *ISSHP* ($\beta = 2$)
ζ	Number of random cycles ($\zeta = 2$)
γ	Threshold of number of hyperedges ($\gamma = 300$)
θ	Number of iterations of eigenvalue solver ($\theta = 80$)

Table 2: Parameters of *SpecPart* framework.

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 $i \in [0, k-1]$:

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

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

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

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 ϵ, 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]

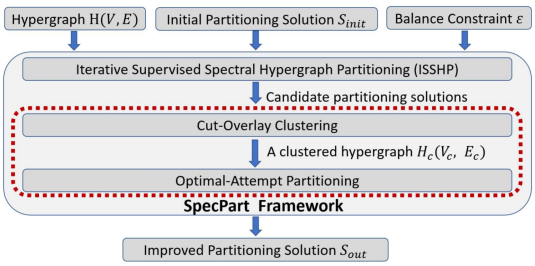


Figure 1: Overview of the *SpecPart* framework.

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 — *distill* the cut structure of the hypergraph (Section 4.2). Then, fast tree-based algorithms are employed to find the best solution S_{best} on those trees. Finally, we set $S = S_{best}$ and the process iterates.

[1] The input initial solution S_{init} may even be a *partial* solution where block membership information is given for only some of the vertices. This may be potentially useful in practical situations but we do not consider it further in this paper.

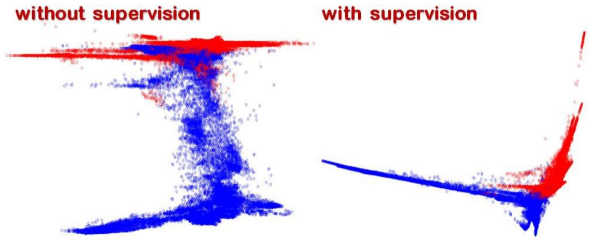


Figure 2: Two vertex embeddings of ISPD IBM14 benchmark. Both are based on the smallest two eigenvectors, computed without supervision (Eq. 2) and with supervision (Eq. 3). The red and blue dots highlight vertices bipartitioned by *hMETIS* with $\epsilon = 2$. With supervision, the distinction between the bipartitioned vertices is cleaner.

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

Cut-Overlay clustering. Let $E_1, \ldots, E_\delta \subset E$ be the sets of hyperedges cut in the δ 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^δ 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 δ solutions in the pool. In our current implementation, we include a parameter γ; in the case when the number of hyperedges in H_c is larger than γ (default value of γ is 300) we run *hMETIS* 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 $\frac{1}{|e|-1}$. 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 $|I| = \sum_{e \in E} |e|$. For this reason we only construct a function f_{L_G} that evaluates matrix-vector products of

Algorithm 1: ISSHP:
Iterative Supervised Spectral Hypergraph Partitioning.

Input: Hypergraph $H(V, E)$, Initial partitioning solution S_{best}
Output: Candidate partitioning solutions $\{S_{c_j}\}$

```
Construct the Laplacian L_G of the clique expansion for H (4.1.1)
Construct the Laplacian B_base of weight-balance graph (4.1.2)
for i = 0; i < β ; i++ do
    Construct Laplacian B_{S_best} based on hint S_best (4.1.3)
    Let B = B_base + B_{S_best}
    Solve the generalized eigenvalue problem L_G x = λBx to
      compute m nontrivial eigenvectors (4.1.5)
    Construct a family of trees {T_{i_j}} based on computed
      eigenvectors (4.2)
    Generate candidate solutions {S_{i_j}} by running tree-sweep and
      METIS on trees {T_{i_j}} (4.3)
    Set S_best to the best partitioning solution in {S_{i_j}}
end
Construct {S_{c_j}} by picking the best δ solutions from {{S_{i_j}}}
return {S_{c_j}}
```

the form $L_G x$, where L_G is the Laplacian of G, 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_G} is an application of the following identity that is based on expressing L_G as a sum of Laplacians of cliques:

$$L_G x = \sum_{e \in E} \frac{1}{|e| - 1} \left(x - \frac{x^T \mathbf{1_e}}{\mathbf{1_e}^T \mathbf{1_e}} \cdot \mathbf{1_e} \right), \tag{4}$$

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

4.1.2 ***Weight-Balance Graph:*** The weight-balance graph G_w 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 has the same vertices as hypergraph H, and edges of weight $w_u \cdot w_v$ between any two vertices u and v. Let w_{V_i} be the weight of block V_i in a partitioning solution S, i.e.,

$$w_{V_i} = \sum_{v \in V_i} w_v. \tag{5}$$

We have

$$w_{V_0} \cdot w_{V_1} = \sum_{v \in V_0} w_v \cdot \sum_{v \in V_1} w_v = \sum_{v \in V_0, u \in V_1} w_v \cdot w_u$$

$$= \sum_{v \in V_0, u \in V_1} w_{e_{vu}} = cutsize_{G_w}(S) \tag{6}$$

We now discuss how to compute matrix-vector products with the Laplacian matrix of G_w, which we denote by B_{base}. Let $\mathbf{w}$ be the vector of vertex weights. We have the identity

$$B_{base} x = \mathbf{w} \circ x - \frac{x^T \mathbf{1}}{\mathbf{1}^T \mathbf{1}} \cdot \mathbf{w}, \tag{7}$$

where $\mathbf{1}$ is the all-ones vector and $\circ$ denotes the Hadamard product. Clearly, this can be carried out in time $O(|V|)$.

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

4.1.3 ***Hint Graph:*** The hint graph G_h is a complete bipartite graph on the two vertex sets V_0 and V_1 defined by the hint solution S_{best}. It is used to incentivize the computation of cuts that are similar to S_{best}, as elaborated in Section 4.1.4. If $B_{S_{best}}$ denotes the Laplacian of the hint graph,

$$B_{S_{best}} x = (x - \frac{x^T \mathbf{1}}{1^T 1} \cdot 1) - (x - \frac{x^T \mathbf{1_{V_0}}}{1_{V_0}^T 1_{V_0}} \cdot \mathbf{1_{V_0}}) - (x - \frac{x^T \mathbf{1_{V_1}}}{1_{V_1}^T 1_{V_1}} \cdot \mathbf{1_{V_1}}) \tag{8}$$

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

4.1.4 ***Intuition on the constructed graphs:*** We solve the generalized eigenvalue problem $L_G x = \lambda B x$, where $B = B_{base} + B_{S_{best}}$. From the discussion in Section 2.2 recall that the eigenvalue problem is directly related to solving

$$\min_x R(x) = \min_x \frac{x^T L_G x}{x^T B x} = \min_x \frac{x^T L_G x}{x^T B_{base} x + x^T B_{S_{best}} x} \tag{9}$$

over the real vectors x. Recall also that this is a relaxation of the minimization problem over 0-1 cut indicator vectors. Let x_S be the indicator vector for some set $S \subset V$. Then, using Equation (1) we have:

- $x_S^T L_G x_S = cutsize_G(S)$ which is a proxy for $cutsize_H(S)$. Thus, the *numerator* incentivizes *smaller* cuts in H.
- $x_S^T B_{base} x_S = cutsize_{G_w}(S)$. By Equation (6), this is equal to $w_S \cdot w_{V-S}$, where w_S is the total weight of the vertices in S. Thus the *denominator* incentivizes a *large* $w_S \cdot w_{V-S}$, which implies balance.
- $x_S^T B_{S_{best}} x_S$ is maximized when all edges of G_{hint} are cut, thus the *denominator* incentivizes cutting *many* edges that are also cut by the *hint*.

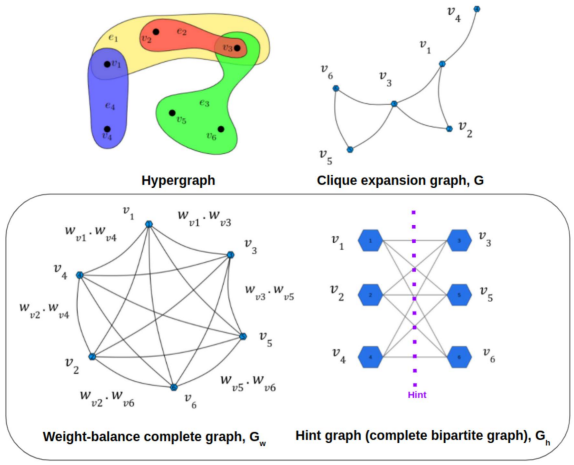


Figure 3: Graphs used in ISSHP, Algorithm 1.

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 $\tilde{G}$ that is spectrally-similar with G and has size at most $3|I|$, where $|I| = \sum_{e \in E} |e|$. More specifically, we build $\tilde{G}$ by replacing every hyperedge e in H with the sum of 3 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 $L_{\hat{G}}$ 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 $X \in \mathbb{R}^{|V| \times m}$ of m computed eigenvectors $\{x_1, x_2, ..., x_m\}$ that we use to construct a number of trees on V.

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 H 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 ζ cycles (as discussed also in Section 4.1.5). Suppose that $Y \in \mathbb{R}^{|V| \times d}$ 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 $e_{uv} \in \hat{G}$ to $||Y_u - Y_v||_2$, i.e., equal to the Euclidean distance between the two vertices in the embedding. We will be computing 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 $l(e_{uv})$ 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 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 $l(e_{uv})$ for any given e_{uv}. 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 $\hat{G}_{X_i}$ for $i = 1, \ldots, m$., and for the graph $\hat{G}_X$. Along with the path graphs, these comprise a family 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 $cutsize_H(S_{e_T})$.

Computing these edge weights on T can be done in $O(\sum_e |e| \log |e|)$ 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.

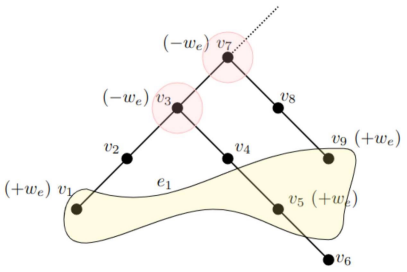


Figure 4: Hyperedge, junctions and their numerical labels

We consider T to be rooted at an arbitrary vertex. In the example of Figure 4, consider hyperedge $e = \{v_1, v_5, v_9\}$. The LCA of its nodes is v_7. Then, the weight of e should be accounted for the set $C_e \subset E_T$ of all tree edges that are ancestors of $\{v_1, v_5, v_9\}$ and descendants of v_7. We do this as follows. (i) We compute a set of *junction* vertices that are LCAs of $\{v_1, v_5\}$ and $\{v_1, v_5, v_9\}$. (ii) We then "label" these junctions with $-w_e$, where w_e is the weight of e. More generally, for a hyperedge e=$\{v_{i_1}, \ldots, v_{i_k}\}$ ordered according to T, we calculate the LCAs for the $k-1$ sets $\{v_{i_1}, \ldots, v_{i_j}\}$ for $j = 2, \ldots, k$, and the junctions are labeled with appropriate negative multiples of w_e. We also label the vertices in e with w_e. (iii) All other vertices are implicitly labeled with 0. Consider 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

[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.

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.

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-below-e_T will equal $cutsize_H(S_{e_T})$. 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 $S \subset V$ that cuts more than one edge on T we have $cutSize_H(S) \leq cutSize_T(S)$, and owing to the spectral origin of T we hope that $cutSize_T(S)$ can provide a good proxy for $cutSize_H(S)$ the cuts of H. Therefore, we use *METIS* [5] to solve a balanced partitioning problem on the reweighted tree, with the original vertex weights from H. This can potentially return a partition $S \subset V$ that cuts more than one edge on T. In some cases we do get $cutSize_H(S) \leq cutSize_H(S_{e_T})$, thus further improving the solution.

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 E5-2650L, 1.70GHz processors and 256 GB memory. We have compared our framework with two state-of-the-art hypergraph partitioners[3] (*hMETIS* [6] and *KaHyPar* [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.

	Statistics		*Best*	*SpecPart*	$Best_w$	$SpecPart_w$
Benchmark	$\|V\|$	$\|E\|$	$\epsilon = 2/10$	$\epsilon = 2/10$	$\epsilon = 2/10$	$\epsilon = 2/10$
IBM01	12752	14111	203 [7] / 180 [43]	202 / 171	227 [44] / 215 [43]	215 / 197
IBM02	19601	19584	354 [43] / 262 [43]	336 / 262	266 [43] / 266 [45]	282 / 256
IBM03	23136	27401	957 [4] / 956 [7]	959 / 952	748 [43] / 681 [43]	813 / 541
IBM04	27507	31970	595 [43] / 542 [43]	593 / 388	506 [43] / 440 [43]	476 / 393
IBM05	29347	28446	1733 [43] / 1715 [7]	1720 / 1688	1727 [43] / 1716 [44]	1724 / 1692
IBM06	32498	34826	978 [43] / 885 [43]	963 / 733	531 [43] / 367 [43]	500 / 306
IBM07	45926	48117	951 [43] / 853 [43]	935 / 760	739 [43] / 737 [43]	776 / 634
IBM08	51309	50513	1159 [4] / 1159 [4]	1146 / 1140	1188 [43] / 1157 [43]	1196 / 1116
IBM09	53395	60902	629 [43] / 624 [25]	620 / 519	523 [43] / 523 [43]	519 / 519
IBM10	69429	75196	1333 [43] / 1254 [25]	1318 / 1261	1133 [43] / 756 [43]	1076 / 443
IBM11	70558	81454	1071 [43] / 960 [25]	1062 / 764	781 [43] / 695 [43]	765 / 649
IBM12	71076	77240	1918 [43] / 1872 [25]	1920 / 1842	1998 [43] / 1982 [43]	1965 / 1973
IBM13	84199	99666	859 [43] / 832 [25]	848 / 693	902 [43] / 833 [43]	843 / 822
IBM14	147605	152772	1865 [43] / 1805 [25]	1859 / 1768	1772 [43] / 1527 [43]	1819 / 1339
IBM15	161570	186608	2833 [43] / 2622 [25]	2741 / 2235	2099 [43] / 1801 [43]	1901 / 1605
IBM16	183484	190048	2059 [43] / 1720 [25]	1951 / 1619	1692 [43] / 1668 [43]	1623 / 1619
IBM17	185495	189581	2403 [43] / 2210 [25]	2354 / 1989	2353 [43] / 2257 [43]	2270 / 2008
IBM18	210613	201920	1587 [43] / 1541 [43]	1535 / 1537	1664 [43] / 1522 [43]	1612 / 1532

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.

5.1 Experimental Results

In this section, we present the experimental results of *SpecPart* with default parameter setting.[5] We run *SpecPart* as follows. Given a hypergraph H and an imbalance factor ϵ, we first run *hMETIS* and/or *KaHyPar* on H to generate an initial partitioning solution S_{init}, which is leveraged by *SpecPart* as a "hint" to generate an improved partition S_{out}. Here we run *hMETIS* and *KaHyPar* with their respective default parameter settings.[6] To avoid any possible confusion, we adopt these conventions: $SpecPart_h$ and $SpecPart_k$ represent the cutsizes of *SpecPart* with the initial solutions generated by *hMETIS* and *KaHyPar* respectively; *SpecPart* represents the best cutsize between $SpecPart_h$ and $SpecPart_k$; and $hMETIS_i$ and $KaHyPar_i$ represent the best cutsizes generated by running *hMETIS* and *KaHyPar* i times with different random seeds respectively.

	Statistics		$hMETIS_5$	$SpecPart_h$	$hMETIS_{20}$	$SpecPart_{20}$
Benchmark	$\|V\|$	$\|E\|$	$\epsilon = 2/20$	$\epsilon = 2/20$	$\epsilon = 2/20$	$\epsilon = 2/20$
sparcT1_core	91976	92827	1073 / 1242	1012 / 903	1066 / 1172	1012 / 903
neuron	92290	125305	260 / 228	252 / 206	260 / 228	252 / 206
stereovision	94050	127085	213 / 129	180 / 91	180 / 129	180 / 91
des90	111221	139557	403 / 377	402 / 358	402 / 377	402 / 358
SLAM_spheric	113115	142408	1061 / 1061	1061 / 1061	1061 / 1061	1061 / 1061
cholesky_mc	113250	144948	301 / 478	285 / 345	285 / 478	285 / 345
segmentation	138295	179051	141 / 112	126 / 78	136 / 112	126 / 78
bitonic_mesh	192064	235328	667 / 554	585 / 483	614 / 554	587 / 483
dart	202351	223301	849 / 546	807 / 543	841 / 540	807 / 540
openCV	217453	284108	535 / 552	510 / 518	511 / 541	510 / 518
stap_qrd	240240	290123	399 / 295	399 / 295	399 / 295	399 / 295
minres	261359	320540	215 / 189	215 / 189	215 / 189	215 / 189
cholesky_bdti	266422	342688	1161 / 1024	1156 / 998	1157 / 947	1156 / 947
denoise	275638	356848	814 / 478	416 / 224	722 / 478	416 / 224
sparcT2_core	300109	302663	1282 / 1630	1244 / 1245	1273 / 1447	1244 / 1245
gsm_switch	493260	507821	5883 / 5352	1852 / 1407	5077 / 5352	1827 / 1407
mes_noc	547544	577664	674 / 632	641 / 617	648 / 632	634 / 617
LU230	574372	669477	3328 / 2710	3273 / 2677	3328 / 2677	3273 / 2677
LU_Network	635456	726999	549 / 528	525 / 524	549 / 528	525 / 524
sparcT1_chip2	820886	821274	1198 / 1023	899 / 783	1198 / 951	899 / 783
directrf	931275	1374742	588 / 343	574 / 295	588 / 295	574 / 295
bitcoin_miner	1089284	1448151	1576 / 1225	1514 / 1225	1489 / 1225	1297 / 1225

Table 4: Statistics of Titan23 suite [8]. $hMETIS_5$ and $hMETIS_{20}$ represent the best cutsizes generated by running *hMETIS* 5 and 20 times with different random seeds. $SpecPart_h$ represents the cutsize generated by *SpecPart* where the hint is obtained from running *hMETIS* once with default random seed. $SpecPart_{20}$ represents the cutsize generated by *SpecPart* where the hint is the solution corresponding to $hMETIS_{20}$.

	Statistics		*KaHyPar*	$KaHyPar_{10}$	$SpecPart_k$
Benchmark	**# Vertices**	**# Hyperedges**	$\epsilon = 2/20$	$\epsilon = 2/20$	$\epsilon = 2/20$
industrial01	349927	428676	2910 / 2426	2806 / 2426	2814 / 2401
industrial02	499718	778588	1871 / 1436	1455 / 955	520 / 234
industrial03	522302	553375	10398 / 8628	8720 / 7646	8392 / 6711
industrial04	570076	648667	2232 / 2889	2058 / 2889	2057 / 2369
industrial05	656245	829321	2679 / 1838	2670 / 1838	2670 / 1829
industrial06	733740	796261	10929 / 8321	9852 / 7646	9884 / 7646
industrial07	733740	796261	680 / 560	680 / 560	680 / 560
industrial08	1245270	1262096	39785 / 34659	39518 / 34614	39546 / 34614

Table 5: Statistics of industrial benchmark suite from a leading FPGA company. $KaHyPar$ and $KaHyPar_{10}$ represent the best cutsize generated by running *KaHyPar* once and 10 times respectively. $SpecPart_k$ represents the cutsize generated by *SpecPart* where the hint is obtained from running *KaHyPar* once with default random seed.

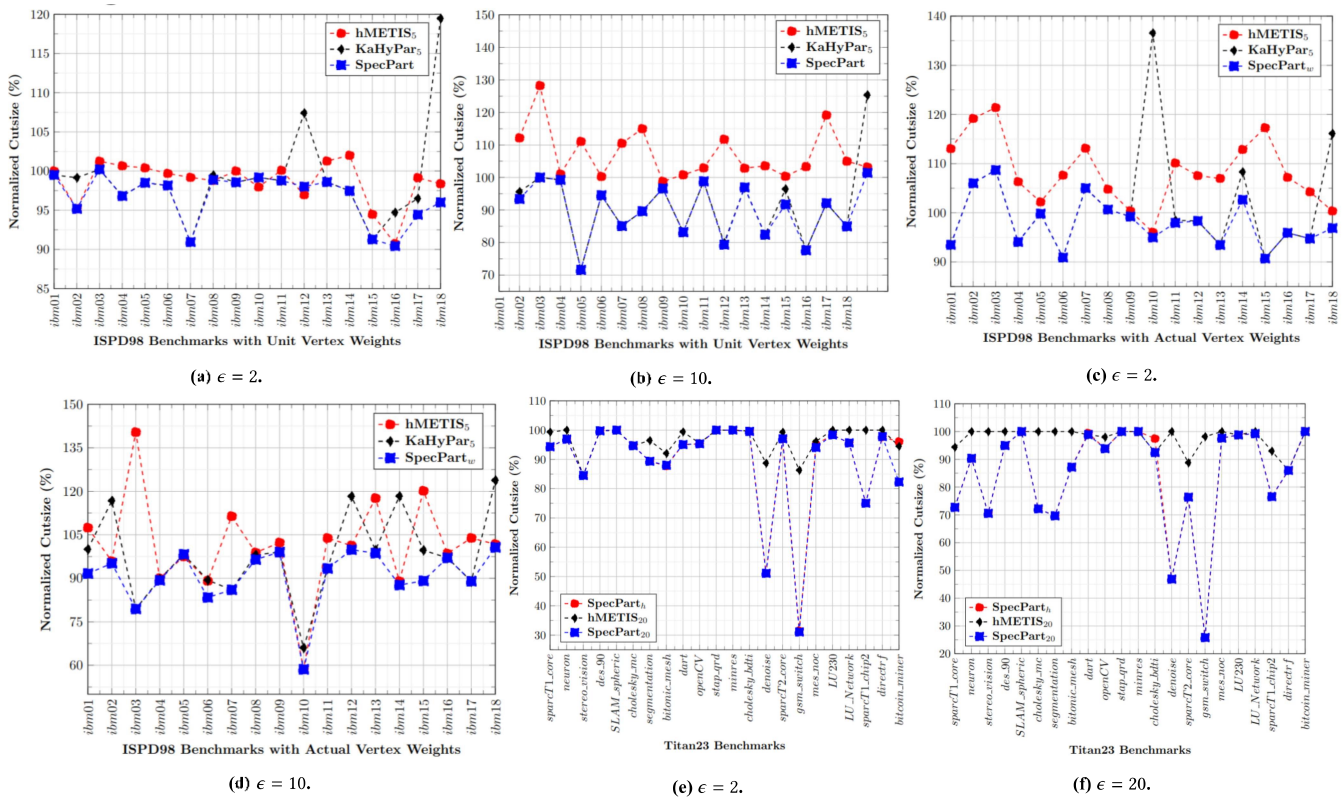
5.1.1 ***ISPD98 benchmarks with unit weights:*** Here we present results for the *ISPD98 VLSI Circuit Benchmark Suite* with unit vertex weights. In Table 3 we present the solutions generated by *SpecPart* and compare them with the corresponding best previously published solutions, with references to the corresponding publications. Figures 5(a)-(b) reports the solutions sizes obtained from *SpecPart*,

[3] We do not compare our results with *PaToH* since it generates weaker cuts compared to *hMETIS* and *KaHyPar* 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 (δ, β, γ, ζ, θ and m) are shown in Table 1.

[6] The default parameter setting for *hMETIS* [7] is: Nruns = 10, CType = 1, RType = 1, Vcycle = 1, Reconst = 0 and seed = 0. The default configuration file we use for KaHyPar is cut_rKaHyPar_sea20.ini [40]).



(a) $\epsilon = 2$. (b) $\epsilon = 10$. (c) $\epsilon = 2$.

(d) $\epsilon = 10$. (e) $\epsilon = 2$. (f) $\epsilon = 20$.

Figure 5: Results of *SpecPart* on *ISPD98 VLSI Circuit Benchmark Suite* [4] and *Titan23 Suite* [8] with different imbalance factors (ϵ).

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*$_5$, which is subject to significant improvement. This illustrates that *SpecPart* can improve very quickly upon solutions computed under stringent walltime requirements.[7]

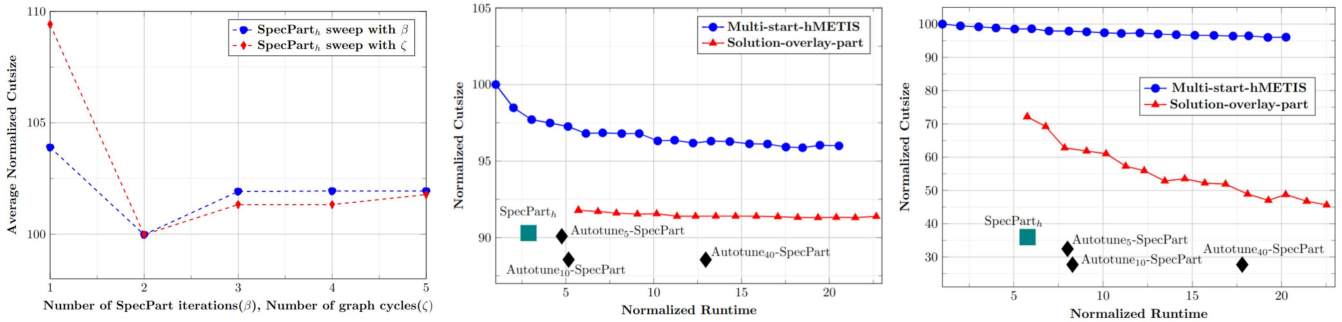
5.1.2 ***ISPD98 benchmarks with actual weights:*** We further verify our framework on the vertex-weighted ISPD98 benchmarks. Mirroring the considerations of section 5.1.1, the results are presented in Table 3 and Figures 5(c)-(d). The inclusion of weights makes the problem more general and potentially 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*, but also for *KaHyPar* which does not include *Titan23* in its experimental study [24].

5.1.3 ***Titan23 benchmarks:*** Table 4 and Figures 5(e)-(f) show the results. While the *SpecPart* runtime overhead over *hMETIS*$_5$ remains at around 50%, the runtime of *KaHyPar* on some of these benchmarks is very large (more than two hours), too high for any reasonable industrial setting (for more details on runtime see [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*$_5$, 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* and generate the cutsize *SpecPart*$_{20}$. We observe that *SpecPart*$_h$ 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, taking almost one hour on some of the industrial benchmarks; *SpecPart* adds less than 5% overhead to single

[7] 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*.



(a) Validation of *SpecPart* default parameter values. (b) Comparison on benchmark *sparcT2_core* ($c = 10$). (c) Comparison on benchmark *gsm_switch* ($\epsilon = 10$).

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$_h$*, and *Autotune$_i$-SpecPart*. *Multi-start-hMETIS* = best cutsize from running *hMETIS* multiple times with different random seeds. *Solution-overlay-part* = cutsize from running *Cut-Overlay Clustering* and *Optimal-Attempt Partitioning* directly on candidate solutions. *SpecPart$_h$* = cutsize from *SpecPart* when the initial solution is from one *hMETIS* run with default random seed. *Autotune$_i$-SpecPart* = cutsize 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 *KaHyPar$_{10}$* 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 *KaHyPar$_{10}$* 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$_h$* with respect to *hMETIS$_5$* on benchmarks *sparcT1_core*, *cholesky_mc*, *segmentation*, *denoise*, *gsm_switch* and *directf*. When we sweep (i.e., vary the value of) one parameter, the remaining parameters are fixed at their default values (Table 2) and ϵ is set to 20. The results appear in Figure 6(a). Sweeping for δ and 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 $\{S_1, S_2, ..., S_\psi\}$ by running *hMETIS* ψ times with different random seeds, and report the best cutsize *Multi-start-hMETIS*. Here ψ 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 $\{S_1, S_2, ..., S_\psi\}$ and report the cutsize *Solution-overlay-part*. For each value of ψ, 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 S_{init}, 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.

REFERENCES

[1] M. Cucuringu, I. Koutis, S. Chawla, G. Miller and R. Peng, "Simple and scalable constrained clustering: a generalized spectral method", *Proc. International Conference on Artificial Intelligence and Statistics*, 2016, pp. 445-454.

[2] N. Alon, R. M. Karp, D. Peleg and D. West, "A graph-theoretic game and its application to the k-server problem", *SIAM Journal on Computing* (24)(1) (1995), pp. 78-100.

[3] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proc. American Mathematical Society* (7)(1) (1956), pp. 48-50.

[4] C. J. Alpert, "The ISPD98 circuit benchmark suite", *Proc. ACM/IEEE International Symposium on Physical Design (ISPD)*, 1998, pp. 80-85.

[5] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", *SIAM Journal on Scientific Computing* (20)(1) (1998), pp. 359-392.

[6] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (7)(1) (1999), pp. 69-79.

[7] G. Karypis and V. Kumar, "hMETIS, a hypergraph partitioning package, version 1.5.3", 1998. http://glaros.dtc.umn.edu/gkhome/fetch/sw/hMETIS/manual.pdf

[8] K. E. Murray, S. Whitty, S. Liu, J. Luu and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD", *Proc. International Conference on Field programmable Logic and Applications*, 2013, pp. 1-8.

[9] Ü. Çatalyürek and C. Aykanat, "PaToH (partitioning tool for hypergraphs)", Boston, MA, Springer US, 2011.

[10] J. Bezanson, A. Edelman, S. Karpinski and V. B. Shah, "Julia: a fresh approach to numerical computing", *SIAM Review* (59)(1) (2017), pp. 65-98.

[11] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions", *Proc. IEEE/ACM Design Automation Conference (DAC)*, 1982, pp. 175-181.

[12] R. Shaydulin, J. Chen and I. Safro, "Relaxation-based coarsening for multilevel hypergraph partitioning", *Multiscale Modeling & Simulation* (17)(1) (2019), pp. 482-506.

[13] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method", *SIAM Journal on Scientific Computing* (23)(2) (2001), pp. 517-541.

[14] T. Heuer, P. Sanders and S. Schlag, "Network flow-based refinement for multilevel hypergraph partitioning", *ACM Journal of Experimental Algorithmics* (24)(2) (2019), pp. 1-36.

[15] D. Kucar, S. Areibi and A. Vannelli, "Hypergraph partitioning techniques", *Dynamics of Continuous, Discrete & Impulsive Systems. Series A: Mathematical Analysis* (11)(2) (2004), pp. 339-367.

[16] R. Merris, "Laplacian matrices of graphs: a survey", *Linear Algebra and its Applications* (197) (1994), pp. 143-176.

[17] A. V. Knyazev, I. Lashuk, M. E. Argentati, and E. Ovchinnikov, "Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in hypre and PETSc", *SIAM Journal on Scientific Computing* (25)(5) (2007), pp. 2224-2239.

[18] I. Koutis, G. L. Miller and R. Peng, "Approaching optimality for solving SDD linear system", *SIAM Journal on Computing* (43)(1) (2014), pp. 337-354.

[19] J. G. Sun and G. W. Stewart, *Matrix Perturbation Theory*, Cambridge, MA, ACADEMIC PressINC, 1990.

[20] I. Koutis, G. L. Miller and D. Tolliver, "Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing", *Computer Vision and Image Understanding* (115)(12) (2011), pp. 1638-1646.

[21] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved algorithms for hypergraph bipartitioning", *Proc. IEEE/ACM Design Automation Conference (DAC)*, 2000, pp. 661-666.

[22] J. R. Lee, S. O. Gharan and L. Trevisan, "Multiway spectral partitioning and higher-order cheeger inequalities", *Journal of the ACM (JACM)* (61) (2014), pp. 1-30.

[23] T. Heuer, "Engineering initial partitioning algorithms for direct k-way hypergraph partitioning", Karlsruher Institut für Technologie, 2015.

[24] S. Sebastian, H. Tobias, G. Lars, A. Yaroslav, S. Christian and S. Peter, "High-quality hypergraph partitioning", *ACM Journal of Experimental Algorithmics* (2022).

[25] S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders and C. Schulz, "k-way Hypergraph Partitioning via n-Level Recursive Bisection", *Proc. The Meeting On Algorithm Engineering And Experiments (ALENEX)*, 2016, pp. 53-67.

[26] J. Y. Zien, M. D. F. Schlag and P. K. Chan, "Multilevel spectral hypergraph partitioning with arbitrary vertex sizes", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (18)(9) (1999), pp. 1389-1399.

[27] L. Hagen and A. B. Kahng, "Fast spectral methods for ratio cut partitioning and clustering", *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1991, pp. 10-13.

[28] N. Rebagliati and A. Verri, "Spectral clustering with more than K eigenvectors", *Neurocomputing* (74)(9) (2011), pp. 1391-1401.

[29] C. J. Alpert and A. B. Kahng, "Multiway partitioning via geometric embeddings, orderings, and dynamic programming", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (14)(11) (1995), pp. 1342-1358.

[30] R. Horaud, "A short tutorial on graph Laplacians, Laplacian embedding, and spectral clustering", 2009. https://csustan.csustan.edu/~tom/Clustering/GraphLaplacian-tutorial.pdf.

[31] F. R. K. Chung, "Spectral graph theory", *CBMS Regional Conference Series in Mathematics*, 1997.

[32] M. Kapralov and R. Panigrahy, "Spectral sparsification via random spanners", *Proc. Innovations in Theoretical Computer Science Conference*, 2012, pp. 393-398.

[33] S. Hoory and N. Linial, "Expander graphs and their applications", *Bulletin of the American Mathematical Society* (43) (2006), pp. 439-561.

[34] C. Ravishankar, D. Gaitonde and T. Bauer, "Placement strategies for 2.5D FPGA fabric architectures", *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 16-164.

[35] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem", *Annals of the History of Computing* (7)(1) (1985), pp. 43-57.

[36] IBM ILOG CPLEX optimizer, https://www.ibm.com/analytics/cplex-optimizer.

[37] V. D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks", *Journal of Statistical Mechanics: Theory and Experiment* (2008)(10) (2008), pp. 10008.

[38] E. G. Boman and B. Hendrickson, "Support theory for preconditioning", *SIAM Journal on Matrix Analysis and Applications* (25)(3) (2003), pp. 694-717.

[39] C. J Alpert, A. B. Kahng and S.-Z. Yao, "Spectral partitioning with multiple eigenvectors", *Discrete Applied Mathematics* (90)(1) (1999), pp. 3-26.

[40] https://github.com/kahypar/kahypar/blob/master/config/cut_rKaHyPar_sea20.ini

[41] Partition solutions, scripts and SpecPart, https://github.com/TILOS-AI-Institute/HypergraphPartitioning.

[42] Ray, https://docs.ray.io/en/latest/index.html.

[43] Latest actual area results for hMETIS, https://vlsicad.ucsd.edu/UCLAWeb/cheese/errata.html.

[44] Comparison of UCLA MLPart (v4.17) and hMETIS (v1.5.3) on instances with actual cell areas (2% configuration), https://vlsicad.ucsd.edu/UCLAWeb/benchmarks/hMETISML02Tab.html.

[45] Comparison of UCLA MLPart (v4.17) and hMETIS (v1.5.3) on instances with actual cell areas (10% configuration), https://vlsicad.ucsd.edu/UCLAWeb/benchmarks/hMETISML10Tab.html.