A Partitioning-Based CAD Flow For Interposer-Based Multi-Die FPGAs

Mahesh A. Iyer*, Andrew B. Kahng‡, Jason Luu†, Bodhisatta Pramanik‡, Kristofer Vorwerk*, Grace Zgheib*
‡University of California San Diego, La Jolla, CA, USA, *Altera, †Intel Corporation
Email: *{mahesh.iyer, kris.vorwerk, grace.zgheib}@altera.com, †{luu_jason}@hotmail.com, ‡{abk, bopramanik}@ucsd.edu

***Abstract*—Multi-die interposer-based FPGA architectures present several design challenges: (i) limited *inter*-die connectivity (interposer) resources and (ii) increased interposer delays compared to *intra*-die routing. Addressing these challenges is critical, as compared to intra-die routing, they directly impact the routability, routed wirelength (rWL) and maximum clock frequency (Fmax) of the design. In this paper, we present a partitioning-based CAD flow tailored for interposer-based multi-die FPGA architectures. Central to our approach is *FPGAPart*, the first open-source timing-driven netlist partitioner that can handle FPGA designs while addressing modern architectural constraints. We integrate *FPGAPart* with the open-source tool VTR 7.0. In particular, we use: (i) VTR 7.0's pre-packing solutions as clustering hints during partitioning and (ii) the FP-Growth algorithm [14] to detect frequently occurring patterns (instances) across multiple timing paths for clustering. Additionally, we introduce *neighborhood influences-based* cutting planes into the core ILP solver in *FPGAPart*, resulting in a $\sim$38$\times$ ILP runtime speedup with $<$1% degradation in solution quality, compared to using no neighborhood influences. Compared to the default VTR 7.0, our flow achieves a geometric mean improvement of $\sim$3% in rWL and $\sim$3% in Fmax for a two-die configuration, with similar improvements across other configurations. Compared to *hMETIS* [17], *METIS* [18] and *TritonPart* [5], *FPGAPart* achieves improvements up to $\sim$4% in rWL and $\sim$7% in Fmax for a two-die configuration, with similar improvements across other configurations.**

I. Introduction

The increasing complexity and size of modern designs have led to significant innovations in FPGA architectures. To meet the demand for higher device capacities, die-stacking technologies have emerged as a solution which enables the creation of larger FPGAs by integrating multiple smaller FPGA dies [1]. Among these technologies, interposer-based multi-die FPGAs have drawn considerable interest. This configuration connects multiple FPGA dies through a silicon interposer [19]. Commonly referred to as a multi-die, or 2.5D, architecture, this approach achieves higher integration levels by leveraging advanced stacking techniques [19].

Multi-die architectures based on interposers present significant challenges due to limited *inter*-die connectivity resources and increased interposer delay compared to *intra*-die routing. Routing through the interposer introduces higher delays, potentially making nets (paths) routed through it timing-critical. Placement and routing tools must address these challenges, specifically the limited inter-die interconnects and the higher interposer delay. These factors directly impact the routability, routed wirelength (rWL), and maximum clock frequency (Fmax) of the design. Furthermore, the limited availability of inter-die interconnects can lead to congestion at die boundaries, necessitating more sophisticated CAD tools.

To mitigate these challenges, classical *min-cut*[1] circuit (hypergraph) partitioning is often considered as an effective strategy. The partitioner divides the netlist into several *sub-netlists*—each sub-netlist is assigned and implemented on a specific die. While the min-cut objective aligns well with minimizing inter-die communication, multi-die FPGA partitioning involves more than optimizing cutsize; for example, the partitioner must be timing aware to prevent timing-critical nets (hyperedges) from getting cut and routed through the interposer. Heterogeneity (multi-dimensional weights) of FPGA designs and the presence of "groups" (*grouping constraints*) of logic (e.g., carry chains, DPS, RAMs) further increase the problem complexity—most hypergraph partitioners do not handle these constraints [6], [17].

In this work, we present a partitioning-based CAD flow tailored for interposer-based multi-die FPGA architectures. The main contributions of this paper are as follows.

- *A partitioning-based CAD flow*. We propose a partitioning-based CAD flow for interposer-based multi-die FPGAs. Our flow is implemented with the open-source tool VTR 7.0 (*interposer* branch) [39].
- *A partitioner for multi-die FPGAs*. We propose *FPGAPart*, the first open-source netlist partitioner that can handle FPGA designs. *FPGAPart* is timing aware and extends the capabilities of the open-source ASIC partitioner, *TritonPart* [5]. We propose three enhancements.
  - (i) *Pre-packing guidance*. We use VTR's pre-packer to pre-pack the netlist, helping the partitioner to avoid poor clustering solutions.
  - (ii) *Parallelized FP-Growth*. We propose a fully parallelized FP-Growth algorithm [14] to improve timing-driven clustering.
  - (iii) *Neighborhood influences*-based cutting planes. We introduce the concept of *neighborhood influences* and incorporate cutting planes into the core Inte-

[1]We use the terms *cut*, *cutsize* and *cutline* in this paper. *Cutsize* is used in the context of partitioning a hypergraph, *cutline* is used in the context of the FPGA architecture and *cut* is used in both contexts.

ger Linear Programming (ILP) solver in *FPGAPart* to achieve faster solver convergence. To the best of our knowledge, this is the first work to utilize cutting planes in the context of ILP-based hypergraph partitioning [32].

- *Experimental confirmations*. We evaluate the performance of our *FPGAPart*-based flow using the VTR 7.0 [30] and Koios benchmark [2] sets, and compare against the default VTR 7.0 flow and modern partitioners, including *hMETIS* [17], *METIS* [18] and *TritonPart* [5]. Experimental results demonstrate that *FPGAPart* can significantly improve Quality of Results (QoR) metrics, achieving improvements up to 14% in rWL and 23% in Fmax, compared to the default VTR flow. Additionally, our *FPGAPart*-based flow shows lower sensitivity to seed noise and produces more routable designs, even under constrained interposer conditions.

II. Related works

Traditionally, partitioning has been utilized in the FPGA CAD flow within two contexts: (i) *packing* and (ii) adding partitioner in the flow to improve quality of results.

Packing. Partitioning has previously been extensively used for packing [23], [11], [12], [8], [34], [35]. In partitioning-based packing, clusters are formed by recursively bi-partitioning the circuit until the partitions contain fewer *primitives* than a predefined limit. Following this, a series of inter-partition moves are performed to legalize the packing solution [11], [12]. Several seed-based methods [3], [22], [21] and clustering-based methods [31] have also been proposed across literature.

CAD flow. Partitioners are often used in the CAD flow for implementing designs on a multi-die FPGA architecture. In [25], the authors investigated the impact of partitioning at two different stages of the VTR flow: *before packing* (on the primitive netlist) and *after packing* (on the clustered netlist). The study demonstrated that partitioning the primitive netlist yields better results compared to partitioning the clustered netlist. The authors use METIS [18] as their partitioning tool. However, METIS requires the conversion of a hypergraph to a graph, which can lead to information loss [33], and it lacks timing awareness. The authors of [27] explore the effect of imbalance on partitioning quality for multi-die FPGA systems. The authors use *hMETIS* [17] to partition the netlist prior to the packing step. They establish that an imbalance factor of 10 and 12 is best for a two-die and three-die architecture, respectively; however, the study does not examine the impact of hMETIS on rWL and Fmax. Additionally, *hMETIS* does not support multi-dimensional balance constraints (corresponding to different logic types on FPGAs) and lacks timing awareness—leaving room for further exploration and improvement.

In this work, we focus on integrating a timing-driven FPGA-based partitioner in a multi-die FPGA CAD flow. The rest of the paper is organized as follows. Section III discusses some preliminary studies and our problem statement; Section IV describes how we integrate a partitioner in the multi-die CAD flow; Section V describes our partitioner in detail; Section VI presents experimental setup and results; and Section VII offers concluding remarks.

III. Preliminaries

A. Modeling 2.5D Multi-Die FPGA Architectures

Open-source interposer-based multi-die FPGA architectures are still unavailable for academic research. In [25], the authors model multi-die architectures by adapting a flagship monolithic FPGA architecture in VTR 7.0. In this work, we use [25] to model interposer-based multi-die FPGA architectures for our experiments. Below, we briefly review the key strategies for modeling multi-die architectures (details can be found in [25]).

Introducing horizontal *cutlines*. The monolithic FPGA die is partitioned into multiple dies by defining horizontal cutlines.[2] These cutlines are directly incorporated into the FPGA fabric. The number of cutlines is user-defined; for example, a four-die FPGA requires three horizontal cutlines. The cutlines are aligned with the FPGA *grid*, ensuring equal die heights across all dies. Special care is taken to avoid intersecting any logical blocks with the cutlines, preserving logical integrity.

Modifying the routing resource graph. The routing resource graph (*rrgraph*[3]) is then modified to model interposer characteristics: limited inter-die interconnects and high interposer delay. A fraction of the routing wires is removed at the boundaries between dies to model these characteristics. An interposer architecture with a higher percentage of reduced wires can simplify manufacturability but increase overall routing complexity.

Adapting the placer. The default placer in VTR 7.0 assumes a monolithic FPGA die and does not account for the presence of an interposer. To address this, two modifications are made to the VTR placer:

- *Timing cost*. VTR's placement timing cost function is modified to consider the number of times a path crosses the interposer. The increased delay introduced by the interposer is factored into this cost function.
- *Wiring cost*. VTR's placement wiring cost function is adapted to reflect the reduced inter-die connectivity.

Adapting the router. Additional terms are introduced to the router's cost function to capture the interposer delay and costs associated with the modifed *rrgraph*.

B. Problem Statement

Since the number of wires that can cross the interposer between multiple FPGA dies is limited, using a partitioner to divide the circuit into one partition per die offers a promising approach for an efficient CAD flow. The objective is to optimize the performance metrics: rWL and Fmax. A partitioning-based CAD flow can also

[2]A cutline represents a split on the monolithic die. Cutlines can have various configurations—horizontal, vertical, or zigzag pattern—although [25] implements only horizontal cutlines.

[3]*rrgraph* is the data structure in VTR that defines all available routing wires (*rrnodes*) and switches in the FPGA.

TABLE I: Terminology and notation used in this work.

Symbol	Description
$w_v \in \mathbb{R}_+^m$, $w_e \in \mathbb{R}_+$	Input weight vector for vertex v and scalar weight for hyperedge e (input).
H, V, E	Hypergraph, vertex set, and hyperedge set (input).
K	Number of output blocks (input).
ϵ	Imbalance parameter for blocks (input).
$P = \{p_1, p_2, ..., p_l\}$	Set of timing-critical paths.
$slack_p, slack_e$	Slack for path p and hyperedge e.
$S = \{V_1, V_2, ..., V_K\}$	A partitioning solution with K blocks: $\{V_i \mid i = 1, 2, ..., K\}$.
$Cut(e)$	$Cut(e) = 1$ if e is cut by S, 0 otherwise.
t_p, t_e	Timing weight (cost) for path p and edge e.
$D(p)$	Number of times a path p is cut.
$SF(p)$	Snake factor of path p.
α	Hyperedge cut cost scaling factor.
β	Non-negative scalar for hyperedge timing cost scaling.
γ	Non-negative scalar for path timing cost scaling.
τ	Non-negative scalar for snaking cost scaling.
Λ	Non-negative scalar for setting cutting planes.

be beneficial by enabling faster compile times through parallelized compiles for each of the partitions. In the following, we discuss the partitioning problem.

Partitioning problem statement. Our partitioning problem formulation is adapted from [5]. The input is a hypergraph $H(V, E)$, where each vertex $v \in V$ is associated with a non-negative m-dimensional weight vector w_v, and each edge $e \in E$ has a non-negative scalar weight w_e. Additionally, we are given a positive integer K and aim to partition H into K partitions.

Given these inputs, $\mathcal{I}$, the goal is to compute a partition of V into K disjoint partitions $S = \{V_1, \ldots, V_K\}$ that minimizes the cost function: $\Phi(\mathcal{I}, S) = \Phi_{\text{cut}}(\mathcal{I}, S) + \Phi_{\text{time}}(\mathcal{I}, S)$. Here, $\Phi_{\text{cut}}(\mathcal{I}, S)$ measures the cutsize, consistent with the standard hypergraph partitioning formulation, while $\Phi_{\text{time}}(\mathcal{I}, S)$ accounts for the timing cost. The constraints we consider are as follows with details in [5].

- *Grouping constraints*. Vertices belonging to the same group must be assigned to the same partition in S [5]. An example of this constraint is to ensure that *groups* of logic, such as carry chains, digital signal processing (DSPs) or RAMs, remain grouped during partitioning.
- *Multi-dimensional balance constraints*. Let $w_v(j)$ denote the j^{th} coordinate of the weight vector w_v. We define $W_j = \sum_{v \in V} w_v(j)$ as the total weight in the j^{th} dimension. The solution S must satisfy the following balance constraint for all $1 \leq i \leq K$ and $1 \leq j \leq m$:

$$\left(\frac{1}{K} - \epsilon\right) W_j \leq \sum_{v \in V_i} w_v(j) \leq \left(\frac{1}{K} + \epsilon\right) W_j \quad (1)$$

  The standard balance constraint must hold along each weight dimension. This is a hard constraint and must always be satisfied.
- *Timing constraints*. We want the partitioner to minimize the number of timing-critical paths (P) that are cut. For a K-way partition, we also want to limit the maximum number of times any timing-critical path can be cut. The set P and its associated slacks can be provided as inputs by a static timing analyzer (STA) tool.

Given a user-defined parameter α (Table I), [5] defines the cutsize (cut cost) as: $\Phi_{cut}(\mathcal{I}, S) = \alpha \sum_{e \in E} Cut(e)$.

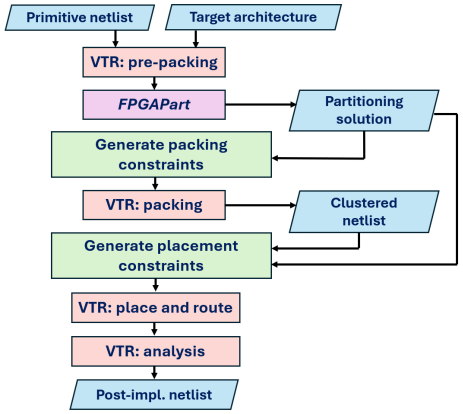

Fig. 1: Our partitioning-based VTR 7.0 flow.

The timing cost follows the formulation in [5], with the quantities $t_e, t_p, D(p)$, and $SF(p)$ defined in Table I. The timing cost $\Phi_{time}(\mathcal{I}, S)$ is based on three components, respectively associated with the timing costs of: (i) *hyperedges* that are cut (*Cut*); (ii) *timing-critical paths* that are cut ((D)); and (iii) the *snaking factor* (*SF*) [5]. The overall timing cost is expressed as:

$$\Phi_{time}(\mathcal{I}, S) = \sum_{e=1}^{|E|} \beta t_e Cut(e) + \sum_{p \in \{P\}} (\gamma D(p) t_p + \tau SF(p)) \quad (2)$$

Here, β, γ, τ are user-defined scalar parameters that control the relative importance of the respective costs (Table I). The final objective of our partitioner is twofold: (i) seamless integration into a standard CAD flow, and (ii) enhancing QoR (rWL and Fmax) for multi-die FPGAs.

IV. Integrating Partitioning into a CAD Flow

In [25], partitioning at the pre-packing stage is shown to yield better results; we followed suit by integrating *FPGAPart* in the VTR 7.0 flow prior to packing as well.[4] Our proposed multi-die FPGA CAD flow is illustrated in Figure 1 and works as follows.

- The input to this flow is a primitive netlist. To incorporate a hint of heterogeneity that can guide the partitioning process, we run VTR 7.0's pre-packing before partitioning.[5] Our experiments demonstrate that incorporating the pre-packing information during partitioning improves the quality of results compared to no pre-packing before partitioning (see Section V-A).
- From the generated partitioning solution, we derive packing constraints. Two *primitives* assigned to different partitions (dies) cannot be packed into the same *logical block*. This feature is implemented in VTR 7.0 [39].
- Next, the partitioning information is forwarded to the placement stage, where we generate placement (*region*)

[4]VTR 8.0 lacks infrastructure to support multi-die FPGA architectures, so we use VTR 7.0 in this work.

[5]The pre-packed solution consists of soft constraints for the VTR packer generated by grouping together netlist primitives that should stay together as a single unit during packing. For further details, refer to [21].

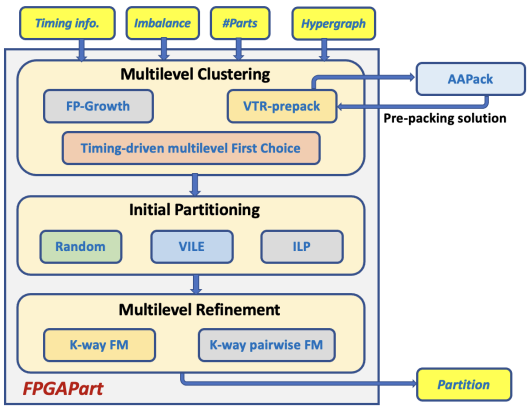

Fig. 2: *FPGAPart* framework. *AAPack* is VTR's packer.

constraints. The placement region of a logical block is determined as the intersection of the placement regions of the primitives packed into that block.

- Finally, we run the router to generate the post-implementation netlist and record the rWL and Fmax.

In this work, we leverage the open-source hypergraph partitioner *TritonPart* [5] and extend it to handle multi-die FPGA applications (*FPGAPart*). *FPGAPart* can handle multi-dimensional balance constraints. However, there are complex legality constraints governing which LUTs and FFs can be packed into logic blocks, as well as which multipliers can be packed into DSP blocks [20]. The partitioner is unaware of these constraints, since it is challenging to estimate accurate resource usage on the FPGA. While running the partitioner after the packing stage is a workaround, this degrades the solution quality. We describe our *FPGAPart* in more detail in Section V.

V. Our Partitioning Approach: FPGAPart

The overall partitioning framework for *FPGAPart* is shown in Figure 2. *FPGAPart* adopts the multilevel partitioning paradigm in [5]. There are three phases.

- *Multilevel clustering*. Here a sequence of progressively coarser hypergraphs is constructed. At each level of coarsening, clusters of vertices are identified and merged into a single vertex, representing the cluster in the coarser hypergraph.
- *Initial partitioning*. After completing the clustering process, we compute an initial partitioning solution for the coarsest hypergraph. The reduced size of this hypergraph allows us to leverage a range of partitioning methods: (i) random, (ii) *very illegal* (VILE) [7] and (iii) ILP.
- *Multilevel refinement*. After obtaining a feasible solution on the coarsest hypergraph from initial partitioning, we perform uncoarsening and move-based refinement to improve the partitioning solution. These steps are carried out level by level, gradually refining the solution as the hypergraph is uncoarsened. We use two variants of refinement: (i) direct K-way FM [5] and (ii) K-way pairwise FM (K-way P-FM) [9].

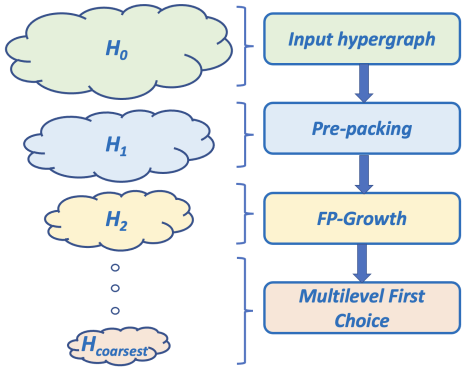

Fig. 3: *FPGAPart*'s multilevel clustering steps.

A. Timing-Driven Clustering

The timing-driven clustering approach in *FPGAPart* builds upon the multilevel First-Choice (FC) implementation in *TritonPart* [5] (illustration in Figure 3). *TritonPart* integrates timing awareness by incorporating a *rating function* that evaluates a *timing score* for clustering decisions. Specifically, it identifies hyperedges with high timing costs and prioritizes merging the vertices connected by these hyperedges. We extend TritonPart's timing-driven clustering by introducing two enhancements.

Pre-packing guidance. We utilize VTR's pre-packing information as clustering hints during *FPGAPart*'s multilevel clustering. VTR employs a pre-packing step to identify logic which can be grouped together to form adaptive logic modules (ALMs), which can later be clustered into logic array blocks (LABs). This information is used to induce the first level of multilevel clustering, ensuring that primitives belonging to the same pre-packed group are assigned to the same cluster during the coarsening stage.

Algorithm 1: Parallelized FP-Growth algorithm.

Inputs: Dataset $\mathcal{D}$, Minimum support threshold $min\_support$, Number of threads $num\_threads$
Outputs: Frequent patterns F

/* Step 1: Filter and sort frequent items */
$item\_counts \leftarrow$ Count occurrences of each item in $\mathcal{D}$
$frequent\_items \leftarrow \{item \mid item\_counts[item] \geq min\_support\}$
foreach *transaction* $t \in \mathcal{D}$ **do**
  $t \leftarrow$ Filter t to retain only $frequent\_items$
  $t \leftarrow$ Sort t in descending order of frequency
/* Step 2: Construct the FP-Tree */
Initialize FP-Tree T with root node
foreach *transaction* $t \in \mathcal{D}$ **do**
  Insert t into T, updating counts for existing prefixes
/* Step 3: Mine patterns from the FP-Tree */
$F \leftarrow \emptyset$
Divide T into $num\_threads$ disjoint subtrees $T_1, T_2, \ldots, T_{num\_threads}$
Initialize $patterns \leftarrow$ vector of maps (one per thread)
foreach *thread* $i \in [1, num\_threads]$ *in parallel* **do**
  $patterns[i] \leftarrow$ Subtree Mining$(T_i, \emptyset, min_support)$
$F \leftarrow \bigcup_{i=1}^{num\_threads} patterns[i]$
/* Step 4: Return the result */
return F

Exploring patterns in timing paths. We extract the top $|P|$ timing-critical paths using VTR's STA tool [24]. From the P paths, we identify frequently occurring vertex sets (*patterns*) using pattern-mining techniques [4]. These vertex sets serve as additional clustering guidance. Since these sets include vertices that frequently appear in P, clustering them together can reduce the snaking factor. We use a parallelized FP-Growth algorithm adapted from [14] and presented in Algorithms 1 and 2.

Algorithm 1 begins by filtering *items* (primitives) in the dataset $\mathcal{D}$ (top P timing-critical paths) based on their *frequency* (Lines 2–6), retaining only those that meet or exceed the specified `min_support`. Each transaction—representing a collection of items—is then processed in two steps: (i) filtering to retain only frequent items and (ii) sorting these items by frequency to perform *prefix-based pattern mining* [14], which systematically explores shared prefixes[6] in transactions. In the next stage, a *Frequent Pattern Tree (FP-Tree)* [14] is constructed (Lines 8–10). This compact data structure maintains the frequency and order of items while eliminating redundancy. To exploit parallelism, the FP-Tree is divided into disjoint subtrees (Lines 12–16), which are independently *mined* using multiple threads. Each thread performs *subtree mining* (Algorithm 2), to explore frequent patterns within its assigned subtree. The results from all threads are combined (Line 17) to generate the output.

Algorithm 2: Subtree mining in FP-Growth.

```
   Inputs: FP-Tree T, Prefix pattern prefix, min_support
   Outputs: Frequent patterns F
1  /* Step 1: Initialize patterns                          */
2  F ← ∅
3  /* Step 2: Process each frequent item                   */
4  foreach item i ∈ T (in reverse frequency order) do
5      new_pattern ← prefix ∪ {i}
6      F ← F ∪ {new_pattern}
7      /* Step 3: Extract conditional pattern base         */
8      conditional_base ← Prefix paths leading to i
9      /* Step 4: Build conditional FP-Tree                */
10     conditional_tree ←
         Construct FP-Tree from conditional_base
11     /* Step 5: Recursively mine conditional
          FP-Tree                                          */
12     if conditional_tree ≠ ∅ then
13         F ← F ∪ Subtree Mining(conditional_tree,
             new_pattern, min_support)
14 return F
```

Our subtree mining methodology is detailed in Algorithm 2. Given a prefix pattern and `min_support`, the algorithm recursively extracts frequent patterns from a given FP-Tree. It begins by initializing an empty set of frequent patterns F (Line 2). Next, the algorithm processes each frequent item in reverse frequency order, generating new patterns by appending the current item to the prefix (Lines 4–6). For each new pattern, the *conditional pattern base*—the set of all prefix paths leading to the current item—is extracted (Line 8). Using this conditional pattern base, a *conditional FP-Tree* is constructed (Line 10), representing a reduced search space for further pattern mining. If the conditional FP-Tree is non-empty, the algorithm recursively mines it (Lines 12–13) to discover additional patterns, which are then added to F.

After completion of FP-Growth, the rest of the multilevel hierarchy is constructed by running [5]'s multilevel timing-driven clustering.

[6]A *prefix* refers to the common initial subsequence shared by multiple transactions. E.g., if two transactions are $[A, B, C]$ and $[A, B, D]$, the prefix is $[A, B]$.

B. Neighborhood Influences-Based Cutting Planes

Improving the scalability and runtime of the ILP solver is crucial for improving the performance of *FPGAPart*. To achieve this, we use *cutting planes*. The core idea is to add additional constraints, known as "cuts", that exclude non-*optimal* parts of the solution space. By narrowing the feasible region, cutting planes can potentially guide the solver toward an *optimal* solution quickly—accelerating the solver's convergence. In this work, we introduce *neighborhood-influences*-based cutting planes to accelerate the ILP solver. First, we formulate our ILP-based partitioning problem instance as follows.

ILP-based partitioning problem. Hypergraph partitioning can be solved optimally (or near-optimally) by casting the problem as an ILP [32]. To write balanced hypergraph partitioning as an ILP, for each block V_i we introduce binary $\{0,1\}$ variables, $x_{v,i}$ for each vertex v, and $y_{e,i}$ for each hyperedge e. Setting $x_{v,i} = 1$ signifies that vertex v is in block V_i, and setting $y_{e,i} = 1$ signifies that all vertices in hyperedge e are in block V_i. We then define the following constraints for each $0 \leq i < K$:

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

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

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

If accurate resource estimates for the FPGA dies are available, the constraints on the partition weight vectors can be replaced with precise resource counts for each type. In this work, we consider an imbalance factor of 5% [25].

Neighborhood influences. We use CPLEX [37] as our ILP solver and leverage the `UserCallback` API to dynamically add cutting planes. We first find vertices with *undetermined assignments*, where a vertex x_v is considered undetermined if the difference between its two largest assignment variables is less than a specified tolerance—we use 0.5 in our implementation. For such vertices, the *neighborhood influence* is calculated by analyzing the sum of assignment variables of its neighboring vertices for each partition. Based on this influence, the partition i_{best} with the highest influence is selected. Cutting planes are then added to encourage the decision: $x_{v,i_{\text{best}}} \geq \Lambda$ for the most influenced partition, and for all other partitions: $x_{v,j} < \Lambda \quad \text{for all } j \neq i_{\text{best}}$.

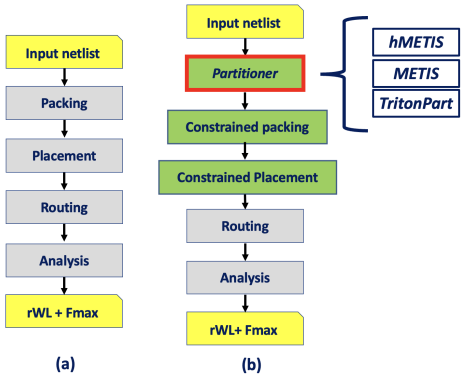

Fig. 4: (a) Default VTR 7.0 flow, and (b) the partitioner-augmented VTR 7.0 flow.

In our implementation, we use $\Lambda = 0.7$. This value is empirically chosen to ensure that the most influenced partition, i_{best}, receives a sufficiently high assignment variable $x_{v,i_{\text{best}}}$. In our extensive background experiments, using $\Lambda = 0.7$ achieves the best quality-runtime tradeoff.

VI. Experimental Setup and Results

FPGAPart is implemented in C++ using approximately 14K lines of code and is built on the OpenROAD infrastructure [16], [38], with all scripts and code available in our public repository [36]. We use CPLEX version 12.10 [37] as our ILP solver. We implemented our *FPGAPart*-based CAD flow using the VTR 7.0 *interposer* branch [39].[7] For all experiments in this paper, we use the flagship architecture of the VTR 7.0 project (*k6_frac_N10_mem32K_40nm.xml*) [20].[8] For the hyperparameters mentioned in Section III-B (α, β, γ and τ), we use the default values from [5]. We divide our validation efforts into four major categories: (i) validation of partitioning in the CAD flow (Section VI-A), (ii) an ablation study of the *neighborhood influences*-based cutting planes (Section VI-B), (iii) study of seed noise (Section VI-C), and (iv) assessment of routability (Section VI-D).

Baselines for comparisons. We use the VTR 7.0 benchmark set [30] and the Koios benchmark set [2] for our evaluations. The characteristics of these benchmarks are shown in Tables II and III.[9] We compare our *FPGAPart*-based flow with the default VTR 7.0 flow (i.e., with no partitioner). Additionally, we integrate *TritonPart* [5], *hMETIS* [17] and *METIS* [25] with the VTR 7.0 flow and compare their performance with *FPGAPart* (Figure 4). We use an imbalance factor [17] of 5% [25] for all partitioners. In our experiments, all circuits are evaluated under "low-stress" routing conditions, using a channel width that is 30% larger than the minimum required for routability [25].

[7]The VTR 7.0 interposer branch provides infrastructure to model interposer-based multi-die FPGAs. We hence use this version of VTR 7.0 in all of our experiments.

[8]The implementation in [39] does not support modern Stratix architecture, so we do not use them in our evaluations.

[9]A few benchmarks were excluded because they result in segmentation faults when run with the default VTR 7.0 interposer branch [39].

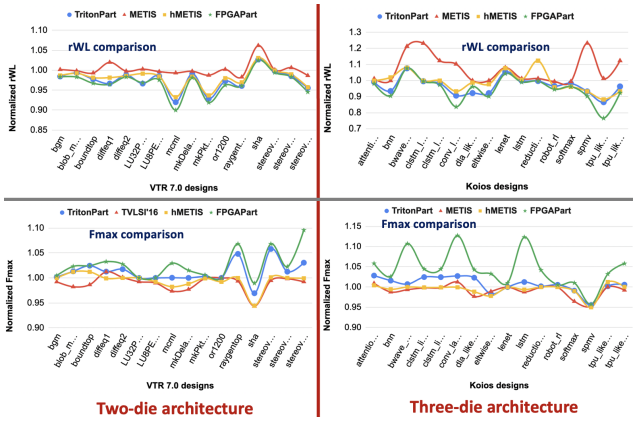

Fig. 5: Comparisons of rWL (top) and Fmax (bottom). Left: VTR 7.0 benchmarks, two-die architecture. Right: Koios benchmarks, three-die architecture.

A. Validation of Partitioning in the CAD Flow

Our evaluation includes three architectural configurations: two-die, three-die, and four-die setups. To model the interposer, we remove 70% of the interconnects crossing the dies and set an interposer delay of $1ns$. These configurations are generated using the scripts provided in [39], with the following settings: (i) `num_cuts` set to 1, 2 and 3, (ii) `percent_wires_cut` set to 70 and (iii) `delay_increase` set to 1. All numbers presented are averaged over 10 unique seed runs.

Comparison with default. Results (rWL and Fmax) are shown in Tables II and III for the VTR 7.0 and Koios benchmarks, respectively. Our observations are as follows.

- *VTR 7.0 benchmarks*. For all configurations, the *FPGAPart*-based VTR 7.0 flow almost always achieves better rWL and Fmax compared to the default VTR 7.0 flow (Table II). The geomean rWL improvements are 3%, 3% and 5% for the two-die, three-die and four-die configurations, respectively, while the geomean Fmax improvements are 3%, 2% and 2% for the same configurations. Notably, in the two-die setup, the design *mcml* shows an rWL improvement of 10%, while *stereovision2* achieves an Fmax improvement of 9%. Additional comparisons of best results across all seeds from the default VTR 7.0 flow and the *FPGAPart*-based VTR 7.0 flow are available in the repository [36].
- *Koios benchmarks*. The *FPGAPart*-based VTR 7.0 flow achieves considerable improvements compared to the default VTR 7.0 flow (Table III). The geomean rWL improvements are 3%, 6% and 10% for the two-die, three-die, and four-die configurations, respectively, while the geomean Fmax improvements are 3%, 4% and 5% for the same configurations. In the two-die setup, the design *lstm* shows an rWL improvement of 14%, while *clstm_like.medium* achieves an Fmax improvement of 23%.

FPGAPart consumes 1038 MB peak memory for Koios designs, whereas the default VTR 7.0 flow consumes 1536 MB. This shows that integrating *FPGAPart* into VTR does not incur any significant memory overhead.

TABLE II: Default VTR 7.0 vs. *FPGAPart*-based VTR 7.0 flow on VTR 7.0 designs (*k6_frac_N10_mem32K_40nm*).

Design	#Primitives	Two Dies				Three Dies				Four Dies			
		Routed WL		Fmax (MHz)		Routed WL		Fmax (MHz)		Routed WL		Fmax (MHz)	
		Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart
bgm	51267	596332	**586373**	39.02	**39.21**	586332	**568438**	39.22	**39.55**	586871	**536433**	39.53	**39.96**
blob_merge	12577	87668	**86173**	101.61	**103.98**	85561	**85272**	99.42	**101.98**	**89007**	89066	101.66	**103.14**
boundtop	3574	29678	**28705**	152.60	**156.36**	30490	**28518**	154.29	**156.36**	30506	**29993**	152.40	**155.47**
diffeq1	569	9406	**9072**	47.20	**48.74**	9732	**9095**	46.50	**47.33**	9689	**8937**	46.91	**47.77**
diffeq2	481	6926	**6808**	60.30	**61.96**	7404	**7172**	61.33	**62.58**	7673	**7172**	60.64	**60.68**
LU32PEEng	31342	1738456	**1683622**	**9.07**	**9.07**	1744322	**1723214**	9.04	**9.07**	1746956	**1735422**	9.17	**9.19**
LU8PEEng	104372	416791	**406528**	8.89	**8.90**	412923	**412716**	9.04	**9.11**	406075	**405922**	8.88	**8.91**
mcml	137425	1132942	**1019365**	12.57	**12.94**	**1143349**	1144356	12.09	**13.45**	1140695	**1139241**	12.07	**14.09**
mkDelayWorker32B	6638	110718	**108623**	140.59	**142.72**	108944	**108542**	135.06	**136.12**	111674	**110392**	144.61	**145.92**
mkPktMerge	335	13818	**12698**	254.62	**256.13**	13587	**12692**	244.22	**244.54**	13748	**12968**	258.87	**259.91**
or1200	3972	49419	**47562**	73.03	**73.04**	50379	**46246**	70.56	**70.98**	50491	**50026**	73.48	**74.56**
raygentop	2733	28778	**27643**	194.69	**207.98**	27969	**25641**	210.15	**211.34**	27977	**26255**	204.79	**204.91**
sha	2330	**21861**	22404	**78.34**	77.51	**22034**	22647	**77.49**	74.47	22294	**20581**	**72.50**	72.12
stereovision0	14672	90851	**90297**	236.36	**252.47**	93553	**92393**	244.25	**245.67**	97389	**90725**	244.12	**245.61**
stereovision1	14312	161077	**158368**	177.51	**181.52**	161302	**160671**	174.49	**178.59**	168532	**153661**	174.39	**179.38**
stereovision2	26672	855932	**809111**	56.08	**61.46**	798888	**734728**	56.98	**61.51**	824386	**720036**	60.83	**62.23**
Improv. (geo. mean) %			**3%**		**3%**		**3%**		**2%**		**5%**		**2%**

TABLE III: Default VTR 7.0 vs. *FPGAPart*-based VTR 7.0 flow on Koios designs (*k6_frac_N10_mem32K_40nm*).

Design	#Primitives	Two Dies				Three Dies				Four Dies			
		Routed WL		Fmax (MHz)		Routed WL		Fmax (MHz)		Routed WL		Fmax (MHz)	
		Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart	Def.	FPGAPart
bnn	204601	2317469	**2223826**	85.02	**85.37**	2458889	**2223826**	84.09	**86.27**	2275962	**1976057**	86.45	**89.27**
bwave_like.fixed.small	16632	490975	**468131**	89.27	**93.41**	**554864**	597075	78.74	**87.17**	**457783**	462614	77.31	**83.77**
clstm_like.medium	743071	4028628	**4013949**	65.77	**80.81**	4200372	**4177747**	81.82	**85.48**	4387657	**4060581**	81.58	**83.17**
clstm_like.small	402331	**1645582**	1757997	107.93	**108.85**	1684506	**1639474**	109.10	**113.95**	1661891	**1511011**	105.46	**105.87**
conv_layer_hls	12097	104564	**96725**	107.99	**110.55**	105613	**88319**	107.92	**121.65**	108383	**80253**	112.71	**119.53**
dla_like.small	260199	**1299244**	1365961	90.08	**93.64**	1386845	**1333858**	95.02	**99.12**	1352230	**1246486**	93.08	**94.06**
eltwise_layer	16187	186987	**175506**	210.39	**214.49**	189793	**171108**	207.35	**214.13**	195773	**179746**	193.72	**212.79**
lenet	190809	217427	**217281**	**109.88**	108.05	**216757**	226055	110.39	**111.51**	**232488**	232699	95.02	**109.24**
lstm	247060	2105091	**1815726**	96.88	**105.00**	2055257	**2035348**	82.68	**92.89**	2105748	**1897236**	82.44	**94.05**
reduction_layer	18323	169277	**154729**	134.74	**136.04**	159782	**159168**	132.92	**138.48**	162764	**145871**	**142.01**	141.88
robot_rl	30529	210775	**204080**	98.59	**99.50**	218674	**206651**	98.49	**99.03**	228486	**200707**	93.45	**98.14**
softmax	13177	171535	**170419**	150.78	**153.98**	174350	**167756**	156.39	**157.97**	176558	**153687**	149.17	**153.88**
spmv	17734	224674	**213602**	153.72	**156.19**	251221	**226637**	**140.93**	134.92	275868	**227806**	141.26	**144.87**
tpu_like.small.os	27097	466148	**446796**	137.64	**137.95**	444053	**339903**	141.22	**145.74**	451459	**394132**	138.70	**145.98**
tpu_like.small.ws	21962	449553	**432153**	104.66	**104.70**	440624	**406784**	100.67	**106.50**	455362	**423278**	96.87	**100.10**
Improv. (geo. mean) %			**3%**		**3%**		**6%**		**4%**		**10%**		**5%**

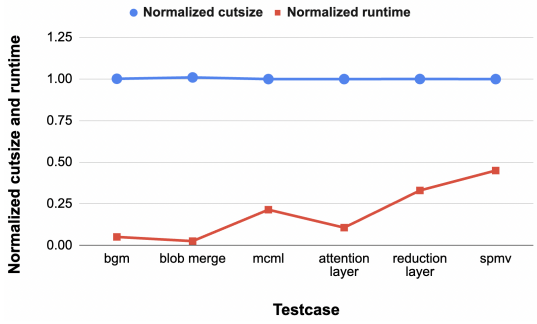

Fig. 6: Validation of *neighborhood influences*.

Comparisons with partitioners. Results of our comparison with *hMETIS*, *METIS* and *TritonPart* on the VTR 7.0 benchmark suite are presented in Figure 5. All numbers in the figure are normalized to default VTR 7.0. We present results for two-die on the VTR 7.0 benchmarks and three-die on the Koios benchmarks, with more results available in [36]. Our *FPGAPart*-based VTR 7.0 flow achieves improvements up to 10% in rWL and 9% in Fmax.

Our *FPGAPart* is slower than *hMETIS* (avg. $\sim 2.5\times$) and *METIS* (avg. $\sim 100\times$), but faster than *TritonPart* (avg. $\sim 5\times$). However, the overall runtime of the *FPGAPart*-based VTR 7.0 flow is faster than the default VTR 7.0 flow (avg. $\sim 1.4\times$). Notably, for larger designs, *FPGAPart* accounts for only $\sim 5\%$ of the total flow runtime.

B. Validation of Neighborhood Influences

To evaluate the impact of *neighborhood influences*-based cutting planes, we compare the *cutsize* and runtime generated by the ILP solver with cutting planes to those generated by the solver without cutting planes. We select six benchmarks from our suite[10] and collect the cutsize and runtime from the ILP on the coarsest hypergraph obtained from *FPGAPart*'s clustering (Section V-A). These hypergraphs typically contain a few hundred vertices and hyperedges. The results, presented in Figure 6 for a four-way partitioning with 5% imbalance, show that our cutting plane technique achieves $\sim 38\times$ runtime speedup and $< 1\%$ cutsize degradation.[11]

C. Study of Seed Noise

We evaluate the impact of *seed* selection on QoR for both our flow and the default VTR 7.0 flow. Similar to Section VI-A, we modify the *seed* values in *FPGAPart* and VTR 7.0, and run our experiments on a subset of the VTR 7.0 and Koios benchmark sets. For each benchmark, we perform 100 runs, each with a unique seed. We run this

[10]We observe similar results across all benchmarks; for brevity, we present results for six benchmarks.

[11]To assess *optimality*, we apply FM-based refinements to the ILP-generated solutions to evaluate how close they are to a near-*optimal* cutsize. On some benchmarks, these refinements lead to $< 1\%$ cutsize improvement, indicating that while the solutions obtained with cutting planes may not be strictly *optimal*, they are still of very high quality.

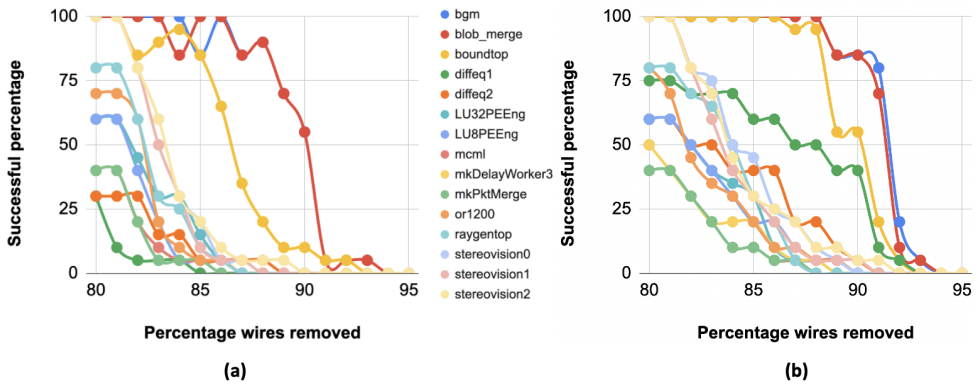

Fig. 7: Routability comparisons between (a) default VTR 7.0 flow and (b) *FPGAPart*-based flow.

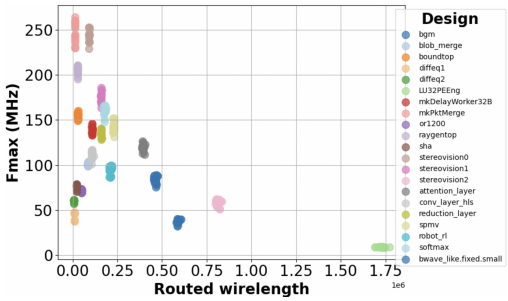

Fig. 8: Seed sensitivity of default VTR 7.0 flow.

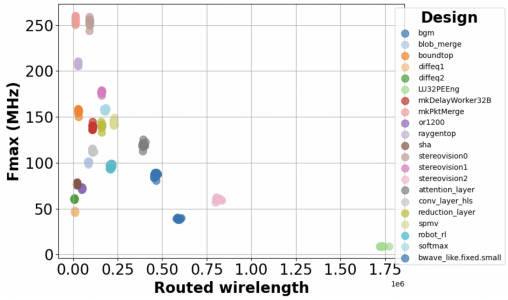

Fig. 9: Seed sensitivity of *FPGAPart*-based VTR 7.0 flow.

evaluation on a three-die configuration with 70% wires removed and interposer delay of $1ns$. The results (Figure 8 and Figure 9) reveal that the default VTR 7.0 flow exhibits high sensitivity in Fmax across different seeds. The arithmetic mean (among the selected benchmark set) of the *coefficient of variation (CV)* in the default VTR 7.0 flow is 3.44 for Fmax, and 1.49 for rWL. In comparison, the *FPGAPart*-based flow achieves a significantly lower CV of 1.35 for Fmax and 1.44 for rWL. This shows the improved stability of our approach to seed noise.

D. Routability Evaluation

We further evaluate the benefits of *FPGAPart* by analyzing its impact on routability (Figure 7). For this evaluation, we consider the same subset of benchmarks as in Section VI-C and run both the default VTR 7.0 flow and our *FPGAPart*-based flow using 20 unique seeds on a four-die architecture. To simulate increasing interposer constraints, we gradually reduce the available interconnects at the die boundaries by varying `percent_wires_cut` from 80% to 95%, while keeping the interposer delay fixed at 1 ns. The results indicate that the *FPGAPart*-based flow achieves more successfully routed runs compared to the default flow. As shown in Figure 7, we observe a "cliff-ing" effect, where routing failures become significantly more frequent as interposer constraints tighten. Notably, our *FPGAPart*-based flow delays the onset of this cliff, effectively pushing it to the right and improving routability under constrained interposer conditions.

VII. Conclusion

This work introduces a partitioning-based CAD flow for interposer-based multi-die FPGAs and presents the first open-source FPGA-specific partitioner, *FPGAPart*. By adapting the multilevel partitioning paradigm and incorporating pre-packing guidance alongside a parallelized FP-Growth algorithm, *FPGAPart* integrates seamlessly into an FPGA CAD flow. Additionally, the use of *neighborhood influences*-based cutting planes is shown to accelerate convergence of the ILP solver. Extensive experimental results validate the benefits of *FPGAPart* compared to the default VTR 7.0 flow, demonstrating superior performance relative to leading partitioners such as *hMETIS*, *METIS*, and *TritonPart*.

Our ongoing efforts are focused on three main directions. First, we aim to adapt *FPGAPart* to the modern VTR 8.0 framework and the enhanced VTR placement flow in [29], enabling compatibility with the latest architectures, algorithms and designs. Second, we seek to enhance the timing-driven refinement in *FPGAPart*, particularly by better usage of STA tools and exploring slack budgeting techniques. Finally, we feel that incorporating spatial awareness into *FPGAPart* could be beneficial for multi-die FPGAs where blocks may be configured in various physical arrangements (e.g., stacked vertically or laid out in a 2D grid). To further explore this, we also intend to evaluate *FPGAPart* under static die configurations with varying aspect ratios.

Acknowledgments. This work is supported by Intel Corporation.

REFERENCES

[1] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing and R. Mahnkopf, "More than Moore white paper", *International Technology Roadmap for Semiconductors*, 2010.

[2] A. Arora, A. Boutros, D. Rauch, A. Rajen, A. Borda, et al., "Koios: A deep learning benchmark suite for FPGA architecture and CAD research", *Proc. FPL*, 2021, pp. 355–362.

[3] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: area efficiency vs. input sharing and size", *Proc. CICC*, 1997, pp. 551–554.

[4] C. Borgelt, "An implementation of the FP-growth algorithm", *Proc. OSDM*, 2005, pp. 1–5.

[5] I. Bustany, G. Gasparyan, A. B. Kahng, Y. Koutis, B. Pramanik and Z. Wang, "An open-source constraints-driven general partitioning multi-tool for VLSI physical design", *Proc. ICCAD*, 2023, pp. 1–9.

[6] I. Bustany, A. B. Kahng, Y. Koutis, B. Pramanik and Z. Wang, "SpecPart: a supervised spectral framework for hypergraph partitioning solution improvement", *Proc. ICCAD*, 2022, pp. 1–9.

[7] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved algorithms for hypergraph bipartitioning", *Proc. ASP-DAC*, 2000, pp. 661–666.

[8] D. T. Chen, K. Vorwerk and A. Kennings, "Improving timing-sriven FPGA packing with physical information", *Proc. FPL*, 2007, pp. 117–123.

[9] J. Cong and S. K. Lim, "Multiway partitioning with pairwise movement", *Proc. ICCAD*, 1998, pp. 512–516.

[10] J. Fairbrother, A. N. Letchford and K. Briggs, "A two-level graph partitioning problem arising in mobile wireless communications", *Computational Optimization and Applications* 69(3) (2018), pp. 653–676.

[11] W. Feng, "K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint", *Proc. FPT*, 2012, pp. 8–15.

[12] W. Feng, J. Greene, K. Vorwerk, V. Pevzner and A. Kundu, "Rent's rule based FPGA packing for routability optimization", *Proc. ISFPGA*, 2014, pp. 31–34.

[13] P. Garrou, M. Koyanagi and P. Ramm, *Handbook of 3d integration: 3d process technology*, New York, Wiley, 2014.

[14] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation", *Proc. SIGMOD*, 2000, pp. 1–12.

[15] T. Heuer, "Engineering initial partitioning algorithms for direct k-way hypergraph partitioning", Karlsruhe Institute of Technology, 2015.

[16] A. B. Kahng and T. Spyrou, "The OpenROAD project: unleashing hardware innovation", *Proc. GOMACTech*, 2021.

[17] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain", *IEEE Trans. on VLSI* 7(1) (1999), pp. 69–79.

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

[19] N. Kim, D. Wu, D. Kim, A. Rahman and P. Wu, "Interposer design optimization for high frequency signal transmission in passive and active interposer using through silicon via (TSV)", *Proc. ECTC*, 2011, pp. 1160–1167.

[20] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu et al., "VTR 7.0: next generation architecture and CAD system for FPGAs", *ACM Trans. on RETS* 7(2) (2014), pp. 1–30.

[21] J. Luu, J. Rose and J. Anderson, "Towards interconnect-adaptive packing for FPGAs", *Proc. FPGA*, 2014, pp. 21–30.

[22] A. S. Marquardt, V. Betz and J. Rose, "Using cluster-based logic blocks and timing driven packing to improve FPGA speed and density", *Proc. FPGA*, 1999, pp. 37–46.

[23] Z. Marrakchi, H. Mrabet and H. Mehrez, "Hierarchical FPGA clustering based on multilevel partitioning approach to improve routability and reduce power dissipation", *Proc. ReConfig*, 2005, pp. 25–28.

[24] K. E. Murray and V. Betz, "Tatum: parallel timing analysis for faster design cycles and improved optimization", *Proc. FPT*, 2018, pp. 110–117.

[25] E. Nasiri, J. Shaikh, A. H. Pereira and V. Betz, "Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs", *IEEE Trans. on VLSI* 24(5) (2016), pp. 1821–1834.

[26] A. H. Pereira and V. Betz, "CAD and routing architecture for interposer-based multi-FPGA systems", *Proc. FPGA*, 2014, pp. 75–84.

[27] R. Raikar and D. Stroobandt, "Multi-die heterogeneous FPGAs: how balanced should netlist partitioning be?", *Proc. SLIP*, 2022, pp. 1–7.

[28] R. Raikar and D. Stroobandt, "LiquidMD: optimizing inter-die and intra-die placement for 2.5D FPGA architectures", *Proc. HEART*, 2024, pp. 90–98.

[29] R. S. Rajarathnam, K. Thurmer, V. Betz, M. A. Iyer and D. Z. Pan, "Better together: combining analytical and annealing methods for FPGA placement", *Proc. FPL*, 2024, pp. 43–52.

[30] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, et al. "The VTR project: architecture and CAD for FPGAs from verilog to routing", *Proc. FPGA*, 2012, pp. 77–86.

[31] L. Singhal, M. A. Iyer and S. Adya, "LSC: a large-scale consensus-based clustering algorithm for high-performance FPGAs", *Proc. DAC*, 2017, pp. 1–6.

[32] T. Heuer, "Engineering initial partitioning algorithms for direct k-way hypergraph partitioning", Karlsruher Institute of Technology, 2015.

[33] Y. Wang and J. Kleinberg, "From graphs to hypergraphs: hypergraph projection and its remediation", *arXiv:2401.08519*, 2024. https://arxiv.org/abs/2401.08519

[34] D. Vercruyce, E. Vansteenkiste and D. Stroobandt, "Runtime-quality tradeoff in partitioning based multithreaded packing", *Proc. FPL*, 2016, pp. 1–9.

[35] D. Vercruyce, E. Vansteenkiste and D. Stroobandt, "How preserving circuit design hierarchy during FPGA packing leads to better performance", *IEEE Trans. on CAD* 37(3) (2018), pp. 629–642.

[36] *FPGAPart* public repository. https://github.com/ABKGroup/FPGAPart

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

[38] The OpenROAD project. https://github.com/The-OpenROAD-Project/OpenROAD

[39] VTR 7.0 interposer branch. https://github.com/verilog-to-routing/vtr-verilog-to-routing/tree/interposer