# Automated Design Space Exploration in High-Level Physical Synthesis

Linfeng Du*, Jiawei Liang†, Jason Lau*, Yuze Chi*, Yutong Xie*, Chunyou Su†, Afzal Ahmad†
Zifan He‡, Jake Ke‡, Jinming Ge†, Jason Cong‡, Wei Zhang†§, Licheng Guo*§
*RapidStream Design Automation, Inc. †The Hong Kong University of Science and Technology ‡University of California, Los Angeles
linfeng.du@connect.ust.hk, wei.zhang@ust.hk, lcguo@rapidstream-da.com

***Abstract*—Implementing HLS accelerators on large-scale multi-die FPGAs presents significant challenges. To address this, researchers have proposed High-Level Physical Synthesis (HLPS), which co-optimizes high-level synthesis and physical design to improve achievable frequency. However, existing HLPS techniques suffer from unstable and inconsistent quality of results (QoRs), largely due to the vast number of parameters that need to be selected by the user in an ad-hoc way. As a result, achieving satisfactory solutions still requires substantial manual effort and expertise in low-level circuit design.**

**We propose a robust and practical design space exploration (DSE) framework that enhances the reliability and QoRs of HLPS by automating the iterative parameter tuning process. Informed by metrics extracted from physical implementation outcomes, the framework applies tailored heuristics to refine HLPS parameters, enabling consistent and automated timing closure. In evaluations with large-scale, real-world designs implemented on representative multi-die devices, our framework achieves an average frequency of 311.06 MHz, reaching $2.42\times$ the frequency of the AMD Vitis/Vivado toolchain (128.48 MHz) and $1.67\times$ that of the leading academic solutions (186.21 MHz).**

## I. Introduction

Modern FPGAs have evolved toward larger, multi-die architectures, yet compiling HLS accelerator designs onto these devices remains an open challenge, stemming from both device and compiler limitations. At the device level, physical wires across die boundaries suffer from limited density (only ∼1/4 of intra-die connections [1]) and increased signal delay (4-8×) in AMD/Xilinx *Stacked Silicon Interconnect (SSI)* technology [2]. At the compiler level, HLS tools, working with untimed high-level languages, often deliver unsatisfactory *quality of results (QoRs)* due to their inability to predict post-*placement and routing (PnR)* layout, which frequently leads to underpipelined, long-spanning connections that degrade operating frequency.

Recently, researchers introduced the *High-Level Physical Synthesis (HLPS)* [1], [3], [4] methodology to improve the achievable frequency, demonstrating significant benefits in many real-world designs [5]–[15]. The core idea of HLPS is to identify potential long wires at the C/C++ level and preemptively insert pipeline registers during the C-to-RTL compilation process, leveraging the inherent latency tolerance prevalent in HLS designs. HLPS partitions the design at latency-tolerant interfaces, floorplans it across the FPGA fabric, and inserts pipeline registers along long-distance connections to improve timing closure. This is achieved without compromising throughput and with only minimal impact on initial latency [4].

Although prior works have demonstrated the potential of HLPS, the timing quality of its solutions remains highly unstable due to the vast parameter space in partitioning, floorplanning, and pipelining. Fig. 1 illustrates how the same HLS design can yield drastically different layouts based on the chosen algorithmic parameters. Some solutions pack modules densely to reduce global wire lengths at the cost of increased local congestion, while others favor more distributed layouts, prioritizing local routability over global wire minimization.

However, due to the complex, black-box nature of PnR tools, predicting final QoRs without complete implementation remains highly challenging, if not impossible. For example, in experiments with a GPT-2 Medium accelerator [16], minor parameter adjustments between two similar floorplans lead to a 5× frequency difference (229.2 vs. 43.1 MHz). Moreover, different designs often benefit from distinct optimization strategies—one HLPS design may achieve 300+ MHz while another using identical parameters fails to route. Consequently, prior HLPS applications have invariably relied on non-trivial human intervention and extensive manual parameter tuning.

§Corresponding authors: Wei Zhang and Licheng Guo

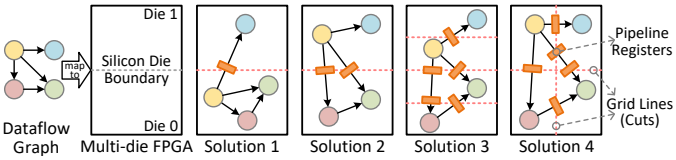


Fig. 1. A toy example demonstrating different device abstraction, floorplan and pipeline strategies in HLPS: Solution 1 divides the device along the die boundary; Solution 2 balances resource utilization between dies with added pipeline registers; Solution 3 addresses long vertical wires caused by the device's vertical-dominant aspect ratio through partitioning into four stacked slots; and Solution 4 adds vertical cuts for better horizontal logic distribution.

This paper introduces a practical and effective exploration framework that autonomously searches for optimal parameter configurations through iterative, physically guided optimization. The framework extracts metrics from the implementation of earlier iterations to guide parameter selection in subsequent ones, reducing the need for manual intervention and domain-specific expertise from accelerator designers. The main contributions of this work are as follows:

- To the best of our knowledge, this work presents the first iterative design space exploration (DSE) framework for HLPS, significantly improving solution robustness and achieving higher operating frequencies than existing commercial and academic approaches.
- We introduce a set of representative physical metrics to evaluate the quality of HLPS solutions and develop tools to extract these metrics. Based on the extracted metrics, we propose concise yet effective heuristics to guide the exploration of HLPS parameters.
- We demonstrate the strength of the framework through significant improvements in achievable frequency for real-world HLPS applications. Evaluated with six large-scale accelerator designs across four distinct multi-die architectures, our framework achieves an average frequency of 311.06 MHz, reaching 2.42× the frequency of commercial AMD Vitis (128.48 MHz) and 1.67× that of the previous state-of-the-art HLPS tool AutoBridge [4] (186.21 MHz).

## II. Background

### A. Multi-Die FPGAs

Modern FPGAs integrate multiple dies with diverse architectures. We examine four representative multi-die FPGAs (Fig. 2) to illustrate the challenges. Alveo U55C has three dies (or *super logic regions, SLRs* in AMD's terminology) with 32 memory ports in SLR0. The right column hosts Vitis Platform IP for host-device communication. Alveo U280 features two extra DDR *memory controllers (MCs)* on SLR0/SLR1. Alveo U250 has four dies, with the latest Vitis Platform IP preoccupying the entire right half of SLR1. Versal FPGAs access memories via hard *Network-on-Chip (NoC)* across the device, featuring larger logic dies and sparser *super long line (SLL)* distribution.

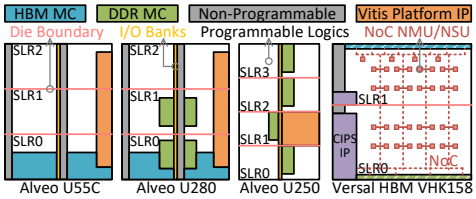


Fig. 2. Architectural overview of representative multi-die FPGAs evaluated.

TABLE I
TUNABLE PARAMETERS IN HLPS DESIGN SPACE

| Symbol | Definition | Domain |
|---|---|---|
| **Device Abstraction Related (Sec. VI-A)** | | |
| $\pi_X, \pi_Y$ | Device grid with $\pi_X$ (column) $\times$ $\pi_Y$ (row) slots | $\pi_X, \pi_Y \in \{1, 2, 3, 4\}$ |
| $M_{i,q}$ | The $i$-th interface of type $q$ bound to IP $M_{i,q}$ | $q \in$ {DDR, HBM, NoC, PCIe, ...} |
| $\Phi$ | Abstraction schedule | $\Phi \in$ {flat, hierarchical} |
| **Floorplanning Related (Sec. VI-B)** | | |
| $R_t^+, R_t^-$ | Global max/min resource limit of type $t$ | $R_t^{+/-} \in [0, 1]$; $t \in$ {BRAM, DSP, |
| $R_{s,t}^+, R_{s,t}^-$ | Slot-wise max/min resource limit of type $t$ in slot $s$ | $R_{s,t}^{+/-} \in [0, 1]$; FF, LUT, URAM} |
| **Pipelining and Routing Related (Sec. VI-C)** | | |
| $C_b$ | Inter-slot wire (crossing) capacity of boundary $b$ | $C_b \geq 0$ |
| $d$ | Pipeline density | $d \in$ {intra-slot, double, mixed, single} |
| $l_i$ | Ratio of usable SLLs in the $i$-th die crossing bundle | $l_i \in [0, 1]$ |

With their heterogeneous resources and complex interconnects, these architectures pose significant challenges to physical implementation.

### B. High Level Physical Synthesis

HLPS abstracts the input HLS design as a graph $G(V, E)$, where vertices $V$ represent design modules and edges $E$ represent latency-tolerant connections. The HLPS flow comprises three main phases:

(1) **Device Abstraction:** Represent the target FPGA device as a grid with $\pi_X \times \pi_Y$ slots, each representing a distinct physical region.
(2) **Floorplanning:** Assign modules to slots while satisfying resource constraints, balancing utilization, and avoiding congestion.
(3) **Pipelining and Routing:** Insert pipeline registers for inter-slot connections and solve routing paths under wire capacity limits.

After these three steps, HLPS generates physically aware RTL and floorplanning constraints, which are then passed to the synthesis and implementation tools to enforce the intended physical layout.

The conventional HLPS process involves a number of empirical parameters, as summarized in Table I. Subtle variations in these parameters can lead to dramatically different solutions, as shown in Fig. 1, potentially resulting in significant fluctuations in the final operating frequency. Here are a few representative examples to give the reader a flavor of these parameters. For instance, in Step (1), users must select the grid size $\pi_X \times \pi_Y$. Large slots may result in under-pipelined intra-slot connections, whereas small slots can compromise the quality of detailed placement. In Step (2), an appropriate resource upper bound for all slots must be defined by users to prevent local congestion and resource under-utilization. In Step (3), users need to specify both the number of pipeline registers to insert and the number of wires allowed between slot pairs. Proper configuration can help reduce wires in hot spots, thus alleviating routing congestion.

## III. MOTIVATING EXAMPLE

To illustrate the challenges and opportunities in HLPS parameter search, we demonstrate how tuning two key parameters—resource limits and wire capacity—can significantly impact implementation quality. We examine Sextans, a real-world *sparse-matrix dense-matrix multiplication (SpMM)* design [10] on Alveo U55C. The default AMD Vitis flow achieves 101.52 MHz for this design, while AutoBridge [4], a conventional HLPS approach, improves this to 182.05 MHz after PnR. Fig. 3 illustrates the device view.

The design's physical layout shows excessive congestion at the bottom die, particularly in slot X0Y0, due to concentrated HBM MC instantiation, while slots X0Y1/X0Y2 remain largely vacant. We address this utilization imbalance by reducing X0Y0's resource limit

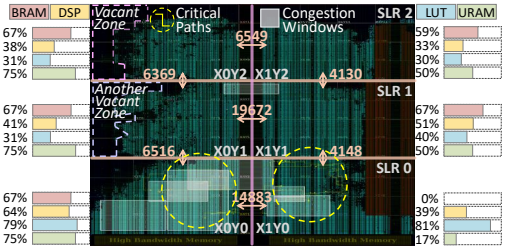


Fig. 3. Physical metrics of Sextans SpMM implemented on Alveo U55C, where the three-die device is abstracted as a 2 (column) $\times$ 3 (row) grid.

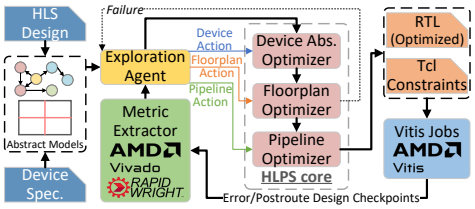


Fig. 4. The framework of metric-guided DSE in HLPS.

$R_{X0Y0,LUT}^+$ during floorplanning, resulting in modules relocated from X0Y0 to upper slots and a higher *maximum frequency (Fmax)* of 248.92 MHz, a 36.73% improvement over AutoBridge.

Further analysis of slot crossings reveals another optimization opportunity. The high volume of wires crossing vertical boundaries X0Y0-X1Y0 and X0Y1-X1Y1 create dense congestion. By reducing wire capacity at these boundaries, i.e., $C_{X0Y0-X1Y0}, C_{X0Y1-X1Y1}$, wires are rerouted through alternative paths during inter-slot routing, achieving more balanced utilization across boundaries. This optimization reaches 285.36 MHz, a 56.75% Fmax improvement.

These two intuitive fixes reveal significant optimization opportunities yet require systematic analysis to fully unleash the potential of parameter search. This observation motivates our HLPS DSE framework that establishes direct links between physical *metrics* and effective HLPS parameter tuning—HLPS *actions*.

## IV. FRAMEWORK OVERVIEW

Fig. 4 illustrates our automated HLPS DSE framework. The HLPS core performs physical layout optimization based on explored parameters. After each iteration, the *Metric Extractor* analyzes implementation results to guide the *Exploration Agent* in adjusting parameters, which we refer to as HLPS *actions*. Notably, we have enhanced all three phases of the HLPS core:

- **Device Abstraction Optimizer**: A **new** component that: (1) decides the early binding of interface $i$ of type $q$ to IPs $M_{i,q}$ (e.g., DDR/HBM/NoC), (2) configures device grid dimensions $(\pi_X, \pi_Y)$, and (3) adaptively schedules floorplanning *integer-linear programming (ILP)* between flat and hierarchical approaches, with tradeoffs in solution quality and execution time.
- **Floorplan Optimizer**: It enhances the conventional ILP formulation of HLPS floorplanning with finer control and better scalability (Table II). It finds optimal slot assignment under both global and slot-wise upper/lower resource constraints $(R_t^{+/-}, R_{s,t}^{+/-})$ while minimizing the total number of inter-slot connections.
- **Pipeline Optimizer**: It replaces the traditional fixed HLPS pipelining and routing with ILP (Table III). It selects optimal path $p_{n,i}$ for each inter-slot connection $n$ to minimize max crossing utilization $\mu$ under wire capacity limits $C_b$, while determining appropriate pipeline density $d$ for each connection. For die-crossing bundles, it additionally optimizes SLL availability ratio $l_i$ for finer control.

These three optimizers form the core of our framework. To guide their parameter selection effectively, we first present physical metrics that capture key characteristics of HLPS solutions.

TABLE II
MIN-TOTAL-CROSSING ILP FOR FLOORPLAN OPTIMIZATION

| | |
|---|---|
| **Obj.**: Minimize the total number of slot crossings | $\text{Min} \sum_{e\in E}\sum_{s\in S}\sum_{d\in S} c_{e,s,d}\cdot x_{e,s,d}, c_{e,s,d} = \sum_{c} 1, c \in path(s,d)$ |
| **Subject to**: | |
| Each edge is routed exactly once | $\sum_{s\in S}\sum_{d\in S} x_{e,s,d} = 1, x_{e,s,d} \in \{0,1\}, \forall e \in E, s, d \in S$ |
| Each vertex $v$ is assigned to exactly one slot $s$ | $\sum_{s\in S} y_{v,s} = 1, y_{v,s} \in \{0,1\}, \forall v \in V, s \in S$ |
| Match src. slot of edge with vertex assignment | $\sum_{d\in S} x_{e,s,d} = y_{v,s}, \forall e \in E, v = \text{src}(e), s \in S$ |
| Match dst. slot of edge with vertex assignment | $\sum_{s\in S} x_{e,s,d} = y_{v,d}, \forall e \in E, v = \text{dst}(e), d \in S$ |
| Global resource limits for each resource type $t$ | $R_t^- \cdot A_{s,t} \le \sum_{v\in V} r_{v,t}\cdot y_{v,s} \le R_t^+ \cdot A_{s,t}, \forall s \in S, t \in T$ |
| Slot-wise resource limits for each resource type $t$ | $R_{s,t}^- \cdot A_{s,t} \le \sum_{v\in V} r_{v,t}\cdot y_{v,s} \le R_{s,t}^+ \cdot A_{s,t}, \forall s \in S, t \in T$ |
| Definition of an edge $e$ crossing a cut $c$ | $z_{e,c} = \sum_{s\in S_1}\sum_{d\in S_2} x_{e,s,d} + \sum_{s\in S_2}\sum_{d\in S_1} x_{e,s,d}, \forall e \in E, c \in C$ |
| Width of edges $e$ crossing cut $c$ cannot exceed capacity $C_c$ | $\sum_{e\in E} w_e \cdot z_{e,c} \le C_c, \forall c \in C$ |

TABLE III
MIN-CROSSING-RATIO ILP FOR INTER-SLOT ROUTING OPTIMIZATION

| | |
|---|---|
| **Obj.**: Minimize the maximum crossing utilization ratio $\mu$ | Min $\mu$ |
| **Subject to**: | |
| Select exactly the $i$-th path for connection $n$ in $P_n$ candidates | $\sum_{i=1}^{P_n} p_{n,i} = 1, \forall n \in N$ |
| The maximum crossing util. ratio for each boundary $b$ | $\mu \ge \frac{\sum_{(n,i)\in crossings_b} w_n \cdot p_{n,i}}{\max(C_b, 1)}, \forall b \in B$ |
| Binary constraint for path selection | $p_{n,i} \in \{0,1\}, \forall n \in N, \forall i \in \{1,\ldots,P_n\}$ |
| Non-negative upper bound | $\mu \ge 0$ |

## V. PHYSICAL METRICS

We identify five key metrics extracted from post-implementation results to evaluate HLPS solutions and guide DSE. These metrics provide comprehensive feedback for parameter optimization:

(A) **Actual resource utilization:** Resource utilization per slot serves as a fundamental guidance for physical optimizations. A highly utilized slot indicates the need to assign less logic to it in order to reduce congestion. For each slot, we extract the post-implementation utilization of key resources: BRAM, DSP, FF, URAM, and LUT.

(B) **Routing congestion distribution:** We identify congestion windows (rectangular regions of densely used interconnects) as supplementary metrics to fine-tune floorplan-related parameters. Their distribution is analyzed through (1) per-slot coverage area in LUT count and (2) shared windows between neighboring slots, guiding both module redistribution and pipeline optimization.

(C) **Critical path distribution:** Timing paths with the lowest *worst negative slack (WNS)* may not always distribute over highly utilized or congested regions. Their traversal patterns across slots guide different optimization strategies: inter-slot critical paths inform pipeline density decisions, while intra-slot paths guide resource limit adjustments.

(D) **Slot-crossing distribution:** For a given slot boundary, the distribution of wire crossing points can be highly imbalanced, particularly along die boundaries. For instance, it is possible that most wires are routed through the left half of the boundary while the right half remains underutilized. Such spatial imbalance provides valuable insights that can guide further optimization, including reducing slot sizes, changing routing paths, and balancing the wire crossing points.

(E) **Vacant zone distribution:** We specifically identify vacant zones, which are defined as large contiguous regions with no placed logic. The presence of a vacant zone within a slot suggests that the available physical resources exceed the requirements. This serves as a strong signal that additional logic can be safely moved into the slot without violating resource constraints.

## VI. METRIC-GUIDED OPTIMIZATION ACTIONS

Based on extensive experiments with large-scale designs across UltraScale+ and Versal FPGAs, we have identified six effective optimization actions corresponding to HLPS's three phases: device abstraction, floorplanning, and pipeline optimization. Each action tunes specific parameters defined in Table I guided by physical metrics. These actions follow natural topological dependencies shown in Fig. 5 to form a systematic optimization flow.

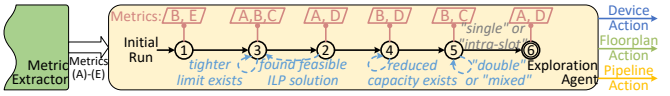


Fig. 5. Metric-guided optimization actions organized as a finite state machine.

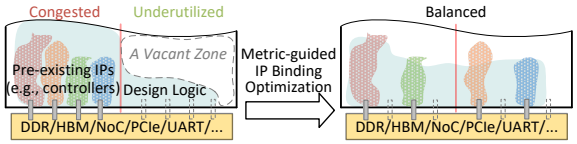


Fig. 6. Physical metric-guided interface binding to alternative ports/IPs in underutilized slots for congestion mitigation.

### A. Actions for Device Abstraction

***Action 1 - optimize IP binding***: Previous HLPS works [1], [3], [4] have demonstrated effectiveness in rapid floorplan and pipeline optimization but overlook interface binding to pre-existing IPs. Their approach relies on rough area estimates and intuitive slot assignments, where resource upper bounds are simply deducted by estimated IP area. Such inaccurate budgeting and fixed pre-assignment often lead to timing issues. Taking HBM MCs as an example, they usually occupy non-negligible logic resources[1] and cause congestion or routing detours [17]. Fig. 6 illustrates a case where fully used MCs cause congestion in the left slot. Although congestion (Metric (B)) can guide general re-binding decisions, we prioritize vacant zone analysis for pre-existing IPs—our strategy identifies alternative memory ports with sufficient adjacent vacant space (Metric (E)), as these densely packed IPs require specific resource patterns and routing space. This action is executed first due to its independence—it can be optimized until Fmax or resource utilization stabilizes and then reused across iterations regardless of subsequent actions.

***Action 2 - refine grid granularity***: Device abstraction requires gridding the device into $\pi_X$ columns by $\pi_Y$ rows. Finer grids enable granular partitioning and pipelining, breaking down timing paths but increasing logic overhead. Conversely, coarser grids minimize the overhead but may insufficiently segment critical paths.

We propose systematic exploration of grid dimensions $(\pi_X, \pi_Y)$, as previous HLPS works rely on empirical cut insertion without investigating its impact on placement quality. The optimal granularity varies across designs—overly large slots lead to dense placement regions requiring physical-level adaptation, while too fine grids cause logic fragmentation and utilization inefficiency, potentially resulting in timing degradation or infeasible solutions. Neither scenario aligns with HLPS's goal of automated physical optimization without requiring design-specific expertise or user's physical awareness.

Beyond prior HLPS works that only consider die boundaries or I/O banks as device cuts, we unlock the potential of finer gridding by introducing parallel or perpendicular cuts to these architectural boundaries. Our refinement strategy considers slot resource utilization (Metric (A)) and crossing distribution (Metric (D)). As Fig. 7 illustrates, a 2×2 grid leads to centrally concentrated logic, with underutilized edge regions and unevenly utilized die-crossing bundles. While both types of cuts help partition moderately utilized slots and encourage more balanced logic distribution across sub-slots, perpendicular cuts offer a critical advantage: when segmenting die boundaries into shorter ones, they help mitigate congestion around die-crossing bundles where limited wire capacity and unbalanced

[1]For example, 32 HBM MCs on Alveo U280 takes ∼15% LUTs on SLR0.

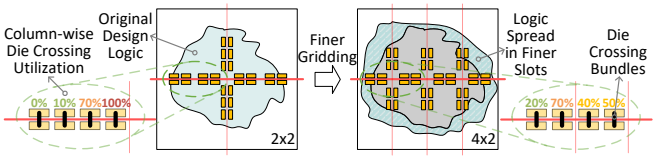


Fig. 7. Finer device gridding underlies better initial physical layout, wider logic spread and more balanced die crossings.

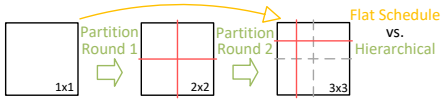


Fig. 8. Converting a flat abstraction schedule with too many cuts into a hierarchical schedule for the computational tractability of ILP.

routing create bottlenecks. This refinement leads to a 4×2 grid and enables the *Pipeline Optimizer* to utilize previously idle cuts between corner slots. The additional registers for slot-crossing paths introduce acceptable overhead given modern FPGAs' abundant FFs [17], further complemented by the wire capacity control detailed in Sec. VI-C.

The gridding proceeds by adding or adjusting one cut per iteration. We prioritize cutting longer boundaries, particularly applying vertical cuts across horizontal die boundaries to optimize timing. All divided regions must remain rectangular to avoid U-shaped regions. The process refines granularity until reaching termination conditions: frequency plateau, module-slot incompatibility, or reaching $n$ clock regions per slot, where a smaller $n$ is preferred for devices with larger clock regions to maintain sufficient granularity.

The computational overhead of ILP solving grows with finer slots, presenting an additional challenge. To improve ILP efficiency, cuts are grouped based on module assignment count, which correlates with problem complexity. Instead of using pure incremental floorplanning [17] or runtime-oriented iterative bi-partitioning [4], we implement an adaptive abstraction schedule that switches from flat to hierarchical scheduling when time limits are exceeded (Fig. 8).

Given that gridding granularity fundamentally impacts subsequent floorplan/pipeline quality and ILP feasibility, each new grid and schedule $(\pi_X, \pi_Y; \Phi)$ initiates a fresh series of floorplanning trials and precedes all pipeline actions in the DSE process.

### B. Actions for Floorplanning

***Action 3 - control resource limits:*** During floorplanning, resource limits per slot must be specified to guide module assignment while minimizing global wirelength. Higher limits allow denser module packing, reducing global wires at the cost of potential local congestion. Conversely, lower limits lead to more distributed placement, alleviating local congestion at the cost of longer global connections.

Our framework accepts user-specified ranges for global resource limit $R_t^{+/-}$. Beyond this global control, we propose Metric (A)∼(C)-guided slot-wise resource limit control to impose tighter constraints on congested slots, as detailed in Alg. 1. For example, with a 90% global limit (Fig. 9), the top-left slot may become overly congested when densely connected modules cluster together, causing wire detours and critical paths. Our framework then iteratively reduces limits for slots with high congestion or critical path density, driving their utilization towards the global average while ensuring sufficient budget in other slots for module relocation.

Introducing slot-wise max limit $R_{s,t}^{+}$ expands the solution space but makes floorplanning convergence more challenging. Fig. 10 shows five modules mapped to a 2×2 grid, where I/O modules (in gray) are pre-assigned to slots with memory ports. Using only global max limit $R_t^{+}$, a balanced state requires $R_t^{+}$ to fall within $[0.6, 0.7)$—a narrow range that is easy to miss with large DSE steps. Alternatively, setting

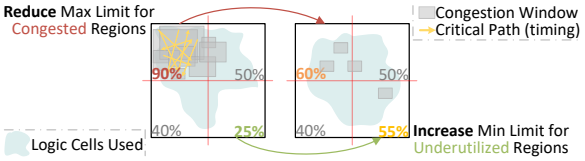


Fig. 9. Slot-wise resource limit adjustment guided by congestion distribution.

**Algorithm 1:** Slot-wise Resource Limit Adjustment

**Input:** Metrics $M$, previous limits $R_{old}$ ($R_{s,t}^{+/-}$ in Table I)
**Output:** Updated limits $R_{new}$

Initialize empty slot-wise limits $R^{+}$, $R^{-}$;
Load critical paths $P$, slot congestion area $C_s$, LUT util. ratio $u$ from $M$;
$\bar{u} \leftarrow$ average LUT util. ratio across slots; $n \leftarrow$ # slots to adjust in each iter.;
**foreach** *critical path* $p \in P$ *until* $|R^{+}| \geq n$ **do**
  **foreach** *slot* $s$ *traversed by path* $p$ *with* $u(s) > \bar{u}$ **do**
    $R^{+}(s) \leftarrow \max((1+\theta)\bar{u}, u(s) - \delta)$ ; // $\theta$: threshold, $\delta$: step
**foreach** *congested slot* $s$ *from top-*$n$ *slots by* $C_s$ **do**
  **if** $R^{+}(s)$ *exists* **then** $R^{+}(s) \leftarrow R^{+}(s) - \delta$;
  **else** $R^{+}(s) \leftarrow \max((1+\theta)\bar{u}, u(s) - \delta)$;
**foreach** *slot* $s \in$ *bottom-*$n$ *by utilization* **do**
  $R^{-}(s) \leftarrow \min((1-\theta)\bar{u}, \max(u(s), \gamma\bar{u}) + \delta)$ ; // $\gamma$: start ratio
**return** $R_{new}$ merged from $R^{+}$, $R^{-}$ and $R_{old}$;

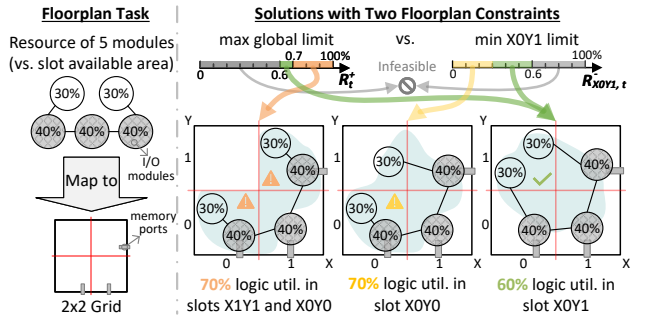


Fig. 10. Minimum utilization constraints complement maximum ones effectively, jointly accelerating convergence towards a balanced floorplan.

slot-wise max limits $[0.4, 0.7)$ for X0Y0 and X1Y1 could achieve the same balance. Without these constraints, any $R_t^{+} \in [0.7, 1.0]$ leaves X0Y1 unused due to the crossing ratio minimization objective. However, when we detect X0Y1's underutilization and introduce minimum limits, setting $R_{X0Y1,t}^{-} \in (0, 0.3]$ forces one module into X0Y1, while $R_{X0Y1,t}^{-} \in (0.3, 0.6]$ achieves ideal balance. This demonstrates the benefit of searching both global/slot-wise and max/min limits for faster and more robust floorplan convergence.

In practice, when no user input is provided, the global limit starts with suggested values from [18]. We initialize slot-wise limits from global ones: maximum limits $R_{s,t}^{+}$ start from $R_t^{+}$, while minimum limits $R_{s,t}^{-}$ considers faster convergence from a potentially higher value of $\gamma \cdot \bar{u}$, where $\bar{u}$ is the average utilization. The search proceeds with step size $\delta$ until the max/min limits converge to $\bar{u} + \theta, \bar{u} - \theta$, respectively, where $\theta$ is a threshold around the average. The process terminates early if either the ILP solver exceeds its time limit or encounters implementation failures.

Since pipelining depends on module locations determined by floorplanning, this resource limit exploration precedes pipeline actions, which reuse both the finest feasible device abstraction scheme and its optimized resource limits.

### C. Actions for Pipelining and Routing

***Action 4 - regulate boundary capacity:*** When addressing imbalanced slot utilization, another effective strategy is to manage wire capacity $C_b$ on each boundary $b$, where a higher boundary capacity allows more pipelined connections through the corresponding slot. If congestion is detected across a boundary, reducing its capacity can

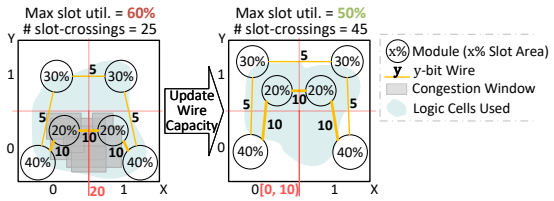


Fig. 11. Management of wire capacity at slot boundaries to improve resource utilization balance across slots.

**Algorithm 2:** Wire Capacity Adjustment

**Input:** Metrics $M$, previous capacity config $C_{old}$
**Output:** Updated device with adjusted wire capacities
Initialize empty capacity updates $C_{new}$;
Load congested neighboring slot pairs and slot-crossing util. $u$ from $M$;
**foreach** *congested slot pair* $(s_1, s_2)$ **do**
  $usage \leftarrow$ crossing count between $s_1$ and $s_2$;
  $C_{new}(s_1, s_2) \leftarrow usage \cdot \alpha / max\_crossings$ ; // $\alpha$: shrink ratio
**foreach** *slot pair* $(s_1, s_2)$ **do**
  **if** $(s_1, s_2) \notin C_{new}$ *and* $C_{old}(s_1, s_2) \cdot \beta < u(s_1, s_2)/u_{max}$ **then**
    $C_{new}(s_1, s_2) \leftarrow C_{old}(s_1, s_2) \cdot \beta$ ; // $\beta$: SLL shrink ratio
**return** device with updated capacities $C_{new}$;

encourage routing detours through other less congested boundaries, thereby alleviating local pressure and improving overall routability.

Fig. 11 shows an initial solution with 25 crossings in total and heavy congestion in the lower FPGA half. To balance slot utilization, we can enforce a minimum limit of $(0.3, 0.5]$ for slots X0Y1 and X1Y1. Alternatively, we can reduce wire capacity at boundaries with high congestion levels (Metric (B)). Lowering the capacity between slots X0Y0 and X1Y0 to $[0, 10)$ achieves better balance across slots by rerouting crossings to the upper boundary despite increasing the total crossings. Alg. 2 details both congestion-guided reduction and iterative die-crossing capacity adjustment when utilization approaches the maximum crossing ratio. This wire capacity optimization precedes other pipeline actions to establish better initial routing conditions, while delayed execution could disrupt their searched results.

***Action 5 - adjust pipeline density****:* Given the optimized floorplan and wire capacity from previous actions, pipeline register insertion strategy still profoundly impacts timing quality. Key questions arise: Should wires crossing slots within the same die use one level of pipeline, or are two levels more appropriate? How to handle die-crossing wires? Insufficient pipelining may fail to resolve critical paths, while excessive pipelining can backfire by introducing additional routing congestion.

Unlike previous HLPS works that use fixed pipeline density or specific designs requiring empirical deep pipelines [19]–[21], we automate the selection among four pipeline schemes $d$ with different density (Fig. 12), guided by congestion (Metric (B)) and critical paths (Metric (C)). The *intra-slot* scheme maximizes pipelining by adding registers even within slots. The *double* scheme applies two-level pipelining only for slot crossings, while the *mixed* scheme reduces to one level except at die boundaries. The *single* scheme employs minimal pipelining with one register per crossing. For larger designs with dense critical paths and congestion, we reduce pipeline density using *mixed* or *single* schemes. Conversely, for smaller designs where critical paths are less congestion-dependent, we explore *double* and *intra-slot* schemes to better utilize abundant routing resources.

This action follows wire capacity tuning as pipeline insertion affects logic (FF) density across slots. If executed earlier, each wire capacity adjustment would cause more dramatic resource fluctuations among slots, potentially destabilizing the search process. This sequencing ensures progressive and robust DSE improvements.

***Action 6 - balance SLL distribution****:* Fig. 13 illustrates an architectural sketch of die crossings in UltraScale+ FPGAs. SLLs are

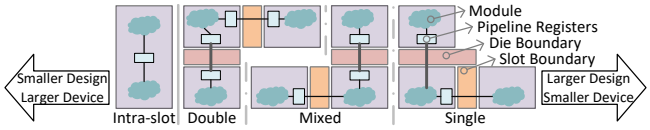


Fig. 12. Various pipelining schemes ranging from high (*intra-slot*) to low (inter-slot *single*) density.

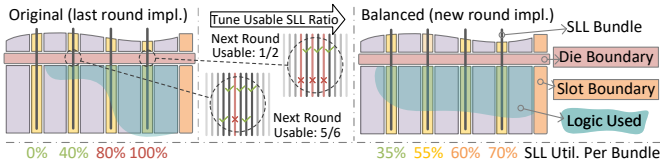


Fig. 13. Adjusting SLL availability ratio in die-crossing bundles to achieve horizontally balanced PnR result.

distributed horizontally as dedicated die-crossing bundles, supporting connections between logic on different dies. These die-crossing wires have much lower density than intra-die wires and show significant utilization variations across bundles. Fig. 13 (left) shows bundles near vertical slot boundaries experiencing higher utilization due to placement preferences, while edge bundles remain underutilized. This imbalanced distribution reflects placer limitations, forcing the router to consider detours through distant bundles, which often degrades timing quality. Such issues need to be addressed proactively before placement through SLL availability control.

We tune the SLL availability ratio $l_i$ of each bundle $i$ based on their utilization (Metric (D)) from previous implementations. As shown in Fig. 13 (middle), we reduce $l_i$ for previously fully used bundles while increasing it for underutilized ones. This routing resource adjustment guides the placer in distributing logic cells more evenly across bundles, improving overall timing. After per-bundle tuning, all $l_i$ values are further scaled according to the overall design size (Metric (A)). Lower ratios can enforce more aggressive re-routing in smaller designs but risk placement failures, while higher ratios offer conservative yet stable improvements for larger designs. After a new round of PnR, SLL utilization becomes more balanced across bundles, as shown in Fig. 13 (right). This SLL balancing is positioned as the final state in the DSE FSM, as its effectiveness depends on the upstream floorplan and pipeline decisions.

### *D. Complete Exploration Agent: Metric-to-Action Integration*

With the topological dependencies discussed, our DSE framework coordinates metrics and actions through a *finite state machine (FSM)* shown in Fig. 5. First, **Act. 1** optimizes pre-existing IP binding. Next, with a default SLR-level abstraction, **Act. 3** optimizes both global and slot-wise resource limits to balance utilization until further adjustment leads to frequency degradation or infeasible solutions. Then, **Act. 2** refines device gridding, with each feasible granularity initiating a new iteration of resource limit search. Based on the gridding achieving the highest Fmax, **Act. 4** regulates boundary capacity to balance wire distribution. Using the best-Fmax solution, **Act. 5** adjusts pipeline density based on design size and congestion patterns. Finally, **Act. 6** balances SLL distribution to optimize die-crossing utilization.

While alternative action orderings are possible, we face a significant runtime challenge from PnR tools, where a single PnR iteration for large designs often exceeds ten hours. This constrains our ability to explore the design space exhaustively. Although theoretical optimality is not guaranteed, the strength of our DSE framework lies in its substantial practical and engineering improvements, bringing tangible benefits to the HLPS community and freeing them from the tuning process. A detailed analysis of the improvements and execution time is presented in Sec. VII.

TABLE IV
RESOURCE STATISTICS OF BENCHMARKS EVALUATED

| Benchmark | Device | # Die | Resource Utilization | | | | |
|---|---|---|---|---|---|---|---|
| | | | BRAM | DSP | FF | LUT | URAM |
| Sextans [10] | Alveo U55C | 3 | 58% | 32% | 21% | 54% | 54% |
| Callipepla [11] | Alveo U55C | 3 | 29% | 18% | 15% | 37% | 40% |
| MM 10x13 [9] | Alveo U55C | 3 | 30% | 38% | 40% | 90% | 0% |
| MM 10x13 [9] | Alveo U250 | 4 | 22% | 29% | 32% | 69% | 0% |
| Serpens [6] | Alveo U280 | 3 | 58% | 17% | 17% | 38% | 54% |
| NTT [22] | Alveo U280 | 3 | 18% | 16% | 13% | 38% | 0% |
| GPT-2 Medium [23] | Versal VHK158 | 2 | 25% | 35% | 22% | 51% | 32% |

Note: the resource percentages are calculated relative to the total logic for users, excluding infrastructure IPs and memory controllers.

### E. Automated DSE

Our DSE and underlying HLPS concepts broadly apply to all FPGA vendors through device-agnostic models and core actions. The framework provides **automated** optimization requiring minimal user intervention beyond standard Vitis flow—we offer a device library covering tens of mainstream multi-die FPGAs, enabling automatic specification generation through a single line of configuration code. Parameters in Table I are automatically explored with initial values from post-synthesis resource estimates, while manual intervention remains possible for faster convergence. Framework robustness is maintained through DSE FSM's intermediate representation, enabling recovery from both user-initiated and system interruptions. While actions execute sequentially, each action's search space allows partially parallel exploration through concurrent implementation jobs, scaled according to available system memory.

## VII. EXPERIMENTS AND RESULTS

### A. Benchmarks and Experimental Setup

We implement the HLPS core upon AutoBridge [4] and extract metrics through both customized RapidWright programs ((A), (D), (E)) and Vivado Tcl scripts ((B), (C)). Our evaluation covers six large-scale HLS accelerators from diverse domains, each with unique architectural challenges and constraints.

- Sextans: An SpMM design on Alveo U55C utilizing 29 HBM MCs, featuring **balanced and above-average** resource utilization.
- Callipepla: A conjugate gradient solver leveraging 26 of 32 HBM channels, used for exploring the ultimate Fmax with our search.
- MM 10x13: A matrix multiplication systolic array utilizing **90% LUT** on Alveo U55C, serving as a high-density stress test.
- Serpens: *A sparse matrix-vector multiplication (SpMV)* implementation on Alveo U280 with **intensive memory access** through 32 HBM channels (256-bit) and 2 DDR interfaces (512-bit each).
- NTT: A number theoretic transform design representing inherently coarse-grained architectures, with **a dominant module** of multi-layer butterfly network occupying ∼1/3 of a die on Alveo U280.
- GPT-2 Medium: A transformer inference engine characterized by **complex control flow** and implemented on Versal VHK158, which has the two **largest dies** among these devices.

### B. Action-by-action: Case Studies

To demonstrate the effectiveness of actions presented in Sec. VI, we analyze the following representative iterations from our systematic HLPS DSE across seven benchmarks (Table IV). Each case study visually validates the optimization impact of a specific action.

*1) Action 1 - optimize IP binding:* The visualization of this action is straightforward and omitted for brevity.

*2) Action 2 - refine grid granularity:* Our motivation for finer gridding stems from unbalanced logic utilization across slots. GPT-2 Medium on VHK158 serves as an ideal case study, where the large die size provides sufficient space to demonstrate how sub-die level gridding can effectively drive logic towards device boundaries, achieving better resource utilization balance across the entire FPGA.

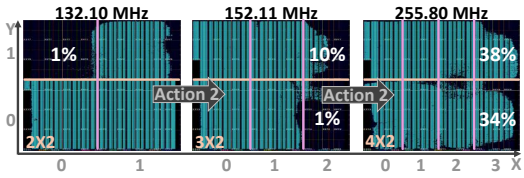


Fig. 14. Progressive resource balancing through grid refinement (2×2, 3×2, to 4×2) of GPT-2 Medium on Versal VHK158.

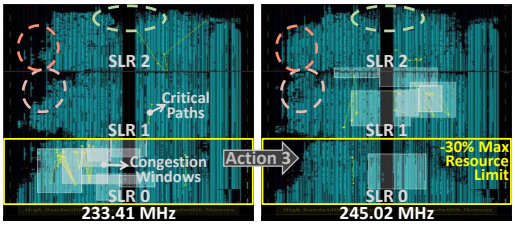


Fig. 15. Module redistribution in Sextans (Alveo U55C, 2×3) achieved by reducing SLR0's maximum resource limit by 30%. White boxes highlight the most congested regions; yellow arrows indicate critical paths.

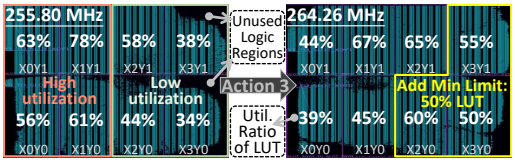


Fig. 16. Improved logic distribution in GPT-2 Medium (Versal VHK158, 4×2) through min LUT utilization limits (50%) on slots X2Y0, X3Y0, X3Y1.

Fig. 14 (left) shows an initial 2×2 grid on VHK158 with one vertical cut, where an 80% LUT utilization limit leads to unbalanced distribution with a nearly vacant slot X0Y1. Given the device's clock region arrangement (10×7), vertical cuts show a stronger influence on logic distribution. Omitting intermediate horizontal cut results, we examine 3×2 and 4×2 configurations with vertical cuts. The 4×2 grid achieves better slot-wise balance by matching module granularity, while the 3×2 variant shows insufficient distribution in slots X2Y0/X2Y1 under the same utilization constraint.

*3) Action 3 - control resource limits:* Finely modularized accelerator designs offer flexible floorplanning across slots, except for modules with hard constraints like HBM MCs. Fig. 15 demonstrates this challenge in Sextans on Alveo U55C, where 29 MCs along SLR0's bottom edge create significant resource pressure. Simply deducting MC resources from SLR0's max limit during floorplanning fails to resolve timing issues. The left portion of Fig. 15 reveals the consequence: concentrated congestion and critical paths in SLR0.

Throughout Action 3's stepped search, reducing SLR0's max resource limit constantly improves timing quality, as shown in the right portion of Fig. 15. Most congestion windows in SLR0 are eliminated, though minor congestion appears in SLR1, where limit control was not applied to previously uncongested regions. The elliptical frames highlight successfully relocated modules in SLR1 and SLR2's marginal regions, where pipelining secures timing closure. These optimized SLR0 limits serve as a reproducible reference for future iterations to prevent MC-induced congestion.

Complementary to max limit control, to demonstrate the effectiveness of minimum resource constraints, Fig. 16 compares the slot-wise resource distributions of GPT-2 Medium on VHK158 before and after adding minimum constraints in a 4×2 grid. The snapshot before optimization shows unbalanced resource utilization, with congested LUT usage in the left half and underutilized slots in the right half. By imposing minimum LUT utilization constraints on slots X2Y0

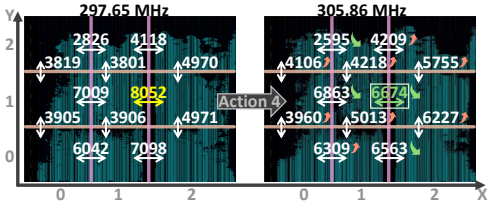


Fig. 17. Balanced wire distribution in Callipepla (Alveo U55C, 3×3) driven by reduced wire capacity at boundary X1Y1-X2Y1.

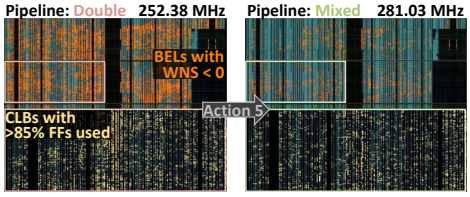


Fig. 18. Logic density mitigation in MM 10x13 (Alveo U55C, 3×3) supported by pipeline scheme optimization (*double* to *mixed*).

and X3Y0/X3Y1, the framework achieves a more balanced module distribution, improving Fmax from 255.80 MHz to 264.26 MHz.

*4) Action 4 - regulate boundary capacity:* Fig. 17 reveals uneven slot crossing distribution in Callipepla: slot X1Y1's east boundary carries 8,052 wires—2× more than its north and south crossings, despite low overall congestion. To address this imbalance, the second `for` loop in Alg. 2 reduces the capacity of heavily utilized boundaries. During the exploration, a 15% capacity reduction at this boundary effectively enforces wire detouring through underutilized boundaries. The capacity reduction not only balances crossing distribution by spreading wires more evenly across slot X1Y1's boundaries, but also creates better distributed routing paths across the device, leading to improved timing quality.

*5) Action 5 - adjust pipeline density:* Fig. 18 demonstrates pipeline density's impact on MM 10x13 (Alveo U55C), a fine-grained design with extreme LUT utilization. Yellow dots indicate *configurable logic blocks (CLBs)* with over 85% FF used, while orange dots mark *Basic Element of Logics (BELs)* with negative setup slacks. Switching from *double* to *mixed* scheme leads to sparser negative-slack BEL distribution with fewer pipeline registers. Although HLPS pipelining generally incurs minimal resource overhead [4], FF insertion can still significantly impact timing in such high-utilization scenarios.

However, the relation between pipeline density and timing proves non-monotonic in high-utilization designs. Reducing density from *double* to *mixed* improves frequency from 252.38 to 281.03 MHz, while further reduction to *single* degrades it to 246.71 MHz. This nonlinear behavior demonstrates the necessity of progressive exploration rather than applying fixed schemes based on utilization levels.

*6) Action 6 - SLL balancing:* We examine Sextans on Alveo U55C to demonstrate the implementation details and effectiveness of SLL balancing. U55C features 16 SLL bundles along each die boundary between its three vertically stacked dies[2]. Fig. 19 shows the left half boundary between SLR0 and SLR1, with a vertical cut in the middle using the 3×3 abstraction scheme. The upper portion reveals significant SLL utilization imbalance in default Vitis implementation, where some SLL bundles exceed 100% estimated utilization after placement while others remain underutilized.

To control SLL availability, we implement a placeholder-based mechanism leveraging Laguna TX/RX registers at bundle ends. As shown in Fig. 19, these registers provide dedicated, optimized connections to crossing bundles. Based on their utilization in the

[2]Typically for AMD/Xilinx UltraScale+, there are 16 SLL columns between two neighboring SLRs, and each column includes 1440 die-crossing wires.

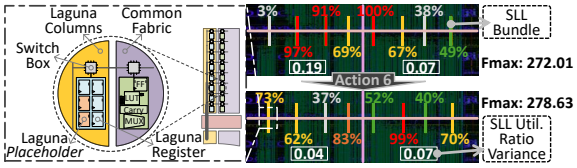


Fig. 19. Applying Laguna placeholder for SLL balancing in Sextans on Alveo U55C (3×3), focusing on left half of SLR0-SLR1 boundary.

previous implementation, we occupy different proportions of Laguna pairs with dummy registers: 2/3 for bundles with [90%, 100%] utilization, 1/2 for [75%, 90%), 1/3 for [60%, 75%), and 1/6 for [40%, 60%). These placeholders are connected through dummy nets prior to logic placement. After placement is completed, we remove these placeholder cells and nets before routing. The final implementation achieves more balanced bundle utilization across die boundaries, as evidenced by reduced variance values (in white boxes). By proactively signaling reduced capacity of each SLL bundle to the placer, this approach promotes horizontal module distribution and mitigates local congestion, leading to improved timing quality.

### C. End-to-end: Overall Timing Closure

Table V summarizes the end-to-end evaluation across all test cases, comparing our approach with state-of-the-art baselines. Experiments ran on an AMD EPYC 7742 server with 256GB DDR4-3200 memory.

*1) **Baselines**:* We evaluate our approach against two baselines: commercial AMD Vitis and academic AutoBridge [4], both preserving memory port binding from referenced sources. Vitis (HLPS-free) implementation use the highest `-O3` optimization with the `Explore` strategy. Target frequency begins at 300 MHz and decreases by 15 MHz steps until successful PnR, avoiding failures from overly aggressive constraints. AutoBridge applies 70%∼90% max global resource limit, scaled to design sizes to ensure sufficient resources.

*2) **Fmax**:* Timing results in Table V follow the action order in Fig. 5, where each column incorporates cumulative effects from all preceding actions. For example, Act. 2 results reflect the execution of Act. 1, Act. 3, and Act. 2 as a sub-FSM of the complete flow.

**Sextans** demonstrates progressive timing improvement through all actions. Initially, 29 HBM MCs cause SLR0 congestion, partially mitigated by Act. 1's congestion-guided memory port reassignment. Act. 3 further reduces SLR0's max limit by up to 30% and achieves 248.92 MHz, though limited by the initial 2×3 abstraction. Act. 2's transition to a 3×3 grid enables better logic distribution towards device edges, facilitating another round of resource limit tuning. The DSE identifies underutilized slots (X0Y1, X0Y2, X1Y2) in the top-left region and converges at 279.62 MHz by enforcing ∼38% minimum utilization constraints. Act. 4 addresses high-volume slot crossings at X1Y0-X2Y0 and X1Y1-X2Y1 boundaries through capacity reduction, effectively alleviating horizontal congestion. Given the design's high utilization, Act. 5 finds the *mixed* pipeline scheme more effective than both *double* and *single*, introducing neither too many FFs nor inter-slot timing issues. Finally, Act. 6 applies *high-level* SLL balancing and improves Fmax to 313.48 MHz.

Other designs exhibit distinct timing closure patterns due to their unique characteristics. **Serpens**, with its maximum MC utilization across all HBM and DDR banks, benefits primarily from Act. 2's max limit reduction on SLR0/SLR1 for congestion mitigation. Given its low utilization, fine-grained resource limit, and wire capacity searching prove less effective, while *intra-slot* pipelining (Act. 5) yields more significant improvements. This *intra-slot* scheme similarly benefits **Callipepla**. However, Serpens' DDR MCs on SLR0/SLR1 limit the impact of SLL balancing, whereas Callipepla, with fewer HBM MCs and no DDR, achieves the highest Fmax through Act. 6.

TABLE V
TOP FMAX (MHZ) ACHIEVED AND CUMULATIVE TIME ELAPSED (HOURS) AFTER SPECIFIC ACTIONS IN TOPOLOGICAL ORDER OF FIG. 5

| Design | Device | Vitis | | AutoBridge | | **Act. 1** | | **Act. 3** | | **Act. 2** | | **Act. 4** | | **Act. 5** | | **Act. 6** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Fmax | Time | Fmax | Time | Fmax | T(n,p) | Fmax | T(n,p) | Fmax | T(n,p) | Fmax | T(n,p) | Fmax | T(n,p) | Fmax | T(n,p) |
| Sextans | Alveo U55C | 101.72 | 16 | 182.05 | 16 | 233.42 | 33(7,2) | 248.92 | 67(6,2) | 279.62 | 119(8,3) | 291.72 | 152(6,2) | 310.46 | 170(2,1) | **313.48** | 187(2,1) |
| Callipepla | Alveo U55C | 149.74 | 10 | 263.13 | 10 | ~~240.15~~ | 30(3,1) | 271.64 | 52(8,2) | 305.15 | 92(15,4) | 310.49 | 113(5,2) | 327.69 | 123(2,1) | **343.52** | 132(3,1) |
| MM 10x13 | Alveo U55C | 89.42 | 14 | 159.10 | 14 | 184.29 | 30(5,2) | *N/A* | 46(2,1) | 252.38 | 61(4,1) | *N/A* | 76(2,1) | **281.03** | 92(1,1) | ~~266.45~~ | 105(3,1) |
| MM 10x13 | Alveo U250 | 118.28 | 18 | *N/A* | 19 | 142.68 | 20(4,1) | 295.48 | 59(7,2) | ~~181.29~~ | 82(2,1) | 317.35 | 120(5,2) | **332.89** | 138(2,1) | *N/A* | 156(3,1) |
| Serpens | Alveo U280 | 174.88 | 10 | 244.80 | 11 | *N/A* | 10(1,1) | 293.17 | 30(8,2) | 308.06 | 71(11,4) | 310.57 | 91(8,2) | **325.83** | 101(2,1) | ~~324.36~~ | 108(3,1) |
| NTT | Alveo U280 | 150.36 | 7 | 152.30 | 7 | *N/A* | 8(4,1) | 195.62 | 15(4,1) | *N/A* | 23(2,1) | 251.87 | 39(5,2) | **310.95** | 46(2,1) | *N/A* | 51(3,1) |
| GPT-2 M. | Versal VHK158 | 114.93 | 19 | 115.90 | 19 | 132.10 | 61(12,3) | 229.20 | 101(7,2) | 255.80 | 233(19,6) | 264.26 | 291(10,3) | **269.75** | 310(2,1) | *N/A* | 325(3,1) |

**Fmax (MHz)** — "~~Numbers~~": the best result achieved in this step while previous actions have generated better results; "N/A": no successful Vitis impl. found here.
**T(n,p) (hours(#,#))** — '**T**': time elapsed till this action finishes; '**n**': # solutions searched in this action; '**p**': # batches of parallel Vitis impl. for the '**n**' solutions.

**MM 10x13**'s extreme LUT utilization (90%) on Alveo U55C makes the *mixed* pipeline scheme particularly effective. High utilization requires 3×3 or finer abstraction, with the best solution achieving near-perfect balance (89%∼90% LUT per slot)—any wire capacity adjustment makes some slots overly congested. On the larger Alveo U250, however, Act. 4 effectively reduces dense slot crossings surrounding the right half of SLR2—a congestion-prone region due to DDR MCs in SLR2/SLR3 and the unavailable right half of SLR1.

**NTT** demonstrates HLPS DSE's effectiveness on under-optimized designs when software abstraction leads to insufficient physical awareness among HLS engineers. It features a large module that could have been better partitioned at the HLS level. Constrained to slots X0Y1/X0Y2 in Alveo U280's 2×3 grid, this module causes ILP failures under any memory, floorplan, or SLL distribution adjustment. Despite these limitations, our framework identifies non-overlapping timing bottlenecks: X0Y0 congestion windows from HBM MCs and cross-die critical paths at the X0Y1-X1Y1 boundary. This spatial separation enables the effective application of *double* and *intra-slot* schemes in Act. 5, improving Fmax to 310.95 MHz.

**GPT-2 Medium** features distinct NoC port binding and requires a finer device grid due to its large dies. Key improvements emerge when horizontal cuts divide vertically stretching wires in its tall and narrow dies. The design reaches peak performance with 4×4 gridding after progressive min-limit exploration of the rightmost slots.

On average, our HLPS DSE achieves 2.42× and 1.67× the Fmax of Vitis and AutoBridge baselines, respectively.

### D. Execution Time

To address the aforementioned runtime challenge, our framework mitigates PnR overhead through parallel HLPS DSE. Constrained by available system memory, for experiments shown in Table V, each action supports up to 4 concurrent Vitis jobs per batch, with actual parallelism determined by search dependencies. For instance, Sextans executes one 2×4 and five 3×3 solutions (both extending 2×3) in two parallel batches, while 4×3 and 3×4 configurations require an additional batch as they build upon 3×3 results.

The convergence patterns vary significantly across designs. Counter-intuitively, the small-scale Callipepla requires more Act. 2 iterations due to its gradual expansion from SLR0/SLR1 to SLR2, driven by lower min-constraints. In contrast, MM 10x13 on U55C achieves rapid convergence with high initial min-constraints, needing only an 85% constraint for slot X0Y2. Serpens bypasses Act. 1 due to full memory utilization, while NTT's large module naturally constrains the physical design space. GPT-2 Medium, with its finest grid granularity, demands the most extensive slot-wise limit tuning.

Despite the DSE process extending to days with limited design space coverage, our approach demonstrates significant practical value over manual tuning. Feedback from hardware engineers consistently indicates acceptance of longer runtime in exchange for improved frequency. One potential future direction is reducing the runtime by extracting physical feedback from partial PnR of a subset of the design, which is beyond the scope of this work.

## VIII. RELATED WORKS

### A. Timing Optimization in Accelerator Designs

Research efforts in accelerator timing optimization have made significant strides through various techniques. Early works focused on pipelining strategies [19]–[21], particularly addressing die-crossing challenges. The effectiveness of these approaches was further enhanced by leveraging FPGA primitives' built-in registers and their naturally optimized interconnections [20], [24]. As designs grew more complex, researchers began exploring floorplanning techniques [12], [25], [26] to reduce SLL utilization and alleviate congestion. More recently, studies have advanced to iterative optimization between front-end design and back-end implementation [27], [28], achieving notable results through the integration of multiple approaches [27]. However, these methods typically target smaller-scale designs and require substantial manual effort for customization, limiting their applicability to large-scale accelerator designs.

### B. High-level Physical Synthesis

Recent advances in physical-aware HLS optimization have shown promising progress. AutoBridge [4] pioneered HLPS techniques by introducing ILP-based floorplanning with fixed routing schemes to minimize die/slot crossings. Building upon this foundation, subsequent works [1], [17], [29]–[31] expanded HLPS methodology into HLS compiler optimizations, incorporating buffer management and task-parallel acceleration APIs. The scope of HLPS has further evolved with research efforts [18], [30], [32]–[35] addressing diverse multi-die and multi-FPGA partitioning challenges. While these approaches have demonstrated promising results, they primarily focus on individual optimization aspects without considering comprehensive physical feedback. Earlier attempts at physical-aware optimization either target smaller designs [36] or focus solely on wirelength optimization [37]. Even with modern architectures like Versal FPGAs featuring hard NoC, recent studies [38] emphasize the continuing importance of efficient SLL utilization. Despite these advances, HLPS techniques have yet to be systematically orchestrated and explored with the guidance of physical metrics.

## IX. CONCLUSION

Our work produces the first automated design space exploration framework for high-level physical synthesis (HLPS). We identify representative physical metrics to evaluate HLPS solution quality and develop a set of concise yet effective heuristics guided by the metrics to search for HLPS parameters leading to timing closure. Through evaluation with six large-scale designs and four different multi-die FPGAs, our tool demonstrates significant performance, achieving an average frequency of 311.06 MHz—2.42× and 1.67× higher frequency compared to the AMD Vitis/Vivado toolchain (128.48 MHz) and leading academic solutions (186.21 MHz), respectively.

## REFERENCES


[1] L. Du, T. Liang, S. Sinha, Z. Xie, and W. Zhang, "Fado: Floorplan-aware directive optimization for high-level synthesis designs on multi-die fpgas," in *Proc. of the 2023 ACM/SIGDA Int. Symp. on Field Program. Gate Arrays*, 2023, pp. 15–25.

[2] Advanced Micro Devices, Inc., "Versal devices using SSI technology (ug1304)," https://docs.amd.com/r/en-US/ug1304-versal-acap-ssdg/Versal-Devices-Using-SSI-Technology, 2023.

[3] J. Lau, Y. Xiao, Y. Xie, Y. Chi, L. Song, S. Xiang, M. Lo, Z. Zhang, J. Cong, and L. Guo, "Rapidstream ir: Infrastructure for fpga high-level physical synthesis," in *Proc. of the 43rd IEEE/ACM Int. Conf. on Comput. Aided Des. (ICCAD)*, 2024, pp. 1–11.

[4] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die fpgas," in *The 2021 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*, 2021, pp. 81–92.

[5] A. Wadood, A. Lu, K. Zhang, and Z. Fang, "Forc: A high-throughput streaming fpga accelerator for optimized row columnar file decoders in big data engines," in *2024 34th Int. Conf. on Field-Program. Log. and Appl. (FPL)*. IEEE, 2024, pp. 318–324.

[6] L. Song, Y. Chi, L. Guo, and J. Cong, "Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication," in *Proc. of the 59th ACM/IEEE Des. Automat. Conf.*, 2022, pp. 211–216.

[7] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM Int. Conf. On Comput. Aided Des. (ICCAD)*. IEEE, 2021, pp. 1–9.

[8] Y.-K. Choi, Y. Chi, J. Lau, and J. Cong, "Taro: Automatic optimization for free-running kernels in fpga high-level synthesis," *IEEE Trans. on Comput.-Aided Des. of Integr. Circuits and Syst.*, vol. 42, no. 7, pp. 2423–2427, 2022.

[9] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *The 2021 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*, 2021, pp. 93–104.

[10] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proc. of the 2022 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*, 2022, pp. 65–77.

[11] L. Song, L. Guo, S. Basalama, Y. Chi, R. F. Lucas, and J. Cong, "Callipepla: Stream centric instruction set and mixed precision for accelerating conjugate gradient solver," in *Proc. of the 2023 ACM/SIGDA Int. Symp. on Field Program. Gate Arrays*, 2023, pp. 247–258.

[12] Z. He, L. Song, R. F. Lucas, and J. Cong, "Levelst: Stream-based accelerator for sparse triangular solver," in *Proc. of the 2024 ACM/SIGDA Int. Symp. on Field Program. Gate Arrays*, 2024, pp. 67–77.

[13] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "Flexcnn: An end-to-end framework for composing cnn accelerators on fpga," *ACM Trans. on Reconfigurable Technol. and Syst.*, vol. 16, no. 2, pp. 1–32, 2023.

[14] X. Tian, Z. Ye, A. Lu, L. Guo, Y. Chi, and Z. Fang, "Sasa: A scalable and automatic stencil acceleration framework for optimized hybrid spatial and temporal parallelism on hbm-based fpgas," *ACM Trans. on Reconfigurable Technol. and Syst.*, vol. 16, no. 2, pp. 1–33, 2023.

[15] Y. Chi, L. Guo, and J. Cong, "Accelerating sssp for power-law graphs," in *Proc. of the 2022 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*, 2022, pp. 190–200.

[16] Z. He, H. Gupta, H. Ke, and J. Cong, "Intrra: Inter-task resource-repurposing accelerator for efficient transformer inference on fpgas," in *Proc. of the 2025 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*. Association for Computing Machinery, 2025, p. 44.

[17] L. Du, T. Liang, X. Zhou, J. Ge, S. Li, S. Sinha, J. Zhao, Z. Xie, and W. Zhang, "Fado: Floorplan-aware directive optimization based on synthesis and analytical models for high-level synthesis designs on multi-die fpgas," *ACM Trans. on Reconfigurable Technol. and Syst.*, 2024.

[18] T. Alonso, L. Petrica, M. Ruiz, J. Petri-Koenig, Y. Umuroglu, I. Stamelos, E. Koromilas, M. Blott, and K. Vissers, "Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning," *ACM Trans. on Reconfigurable Technol. and Syst. (TRETS)*, vol. 15, no. 2, pp. 1–34, 2021.

[19] E. Fournier, C. Teodorov, and L. Lagadec, "Carnac: Algorithm variability for fast swarm verification on fpga," in *2021 31st Int. Conf. on Field-Program. Log. and Appl. (FPL)*. IEEE, 2021, pp. 185–189.

[20] S. Kashani, M. Emami, K. Kamahori, S. Pourghannad, R. Raj, and J. R. Larus, "A 475 mhz manycore fpga accelerator for rtl simulation," in *Proc. of the 2024 ACM/SIGDA Int. Symp. on Field Program. Gate Arrays*, 2024, pp. 78–84.

[21] M. Lo, Z. Fang, J. Wang, P. Zhou, M.-C. F. Chang, and J. Cong, "Algorithm-hardware co-design for bqsr acceleration in genome analysis toolkit," in *2020 IEEE 28th Annu. Int. Symp. on Field-Program. Custom Comput. Machines (FCCM)*. IEEE, 2020, pp. 157–166.

[22] Y.-K. Choi, "Autontt: Code generator for fpga ntt accelerator," https://github.com/applesforme/AutoNTT, 2024.

[23] Z. He, A. Truong, Y. Cao, and J. Cong, "Intar: Inter-task auto-reconfigurable accelerator design for high data volume variation in dnns," *arXiv preprint arXiv:2502.08807*, 2025.

[24] J. Zhang, W. Zhang, G. Luo, X. Wei, Y. Liang, and J. Cong, "Frequency improvement of systolic array-based cnns on fpgas," in *2019 IEEE Int. Symp. on Circuits and Syst. (ISCAS)*. IEEE, 2019, pp. 1–4.

[25] A. K. Jain, C. Ravishankar, H. Omidian, S. Kumar, M. Kulkarni, A. Tripathi, and D. Gaitonde, "Modular and lean architecture with elasticity for sparse matrix vector multiplication on fpgas," in *2023 IEEE 31st Annu. Int. Symp. on Field-Program. Custom Comput. Machines (FCCM)*. IEEE, 2023, pp. 133–143.

[26] S. K. Prakash, H. Patel, and N. Kapre, "Managing hbm bandwidth on multi-die fpgas with fpga overlay nocs," in *2022 IEEE 30th Annu. Int. Symp. on Field-Program. Custom Comput. Machines (FCCM)*. IEEE, 2022, pp. 1–9.

[27] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear algebra on hbm-equipped fpgas using hls: A case study on spmv," in *Proc. of the 2022 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*, 2022, pp. 54–64.

[28] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking," in *The 2021 ACM/SIGDA Int. Symp. on Field-Program. Gate Arrays*, 2021, pp. 105–115.

[29] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang *et al.*, "Tapa: a scalable task-parallel dataflow programming framework for modern fpgas with co-optimization of hls and physical design," *ACM Trans. on Reconfigurable Technol. and Syst.*, vol. 16, no. 4, pp. 1–31, 2023.

[30] N. Prakriya, Y. Chi, S. Basalama, L. Song, and J. Cong, "Tapa-cs: Enabling scalable accelerator design on distributed hbm-fpgas," in *Proc. of the 29th ACM Int. Conf. on Architectural Support for Program. Languages and Operating Syst., Volume 3*, 2024, pp. 966–980.

[31] M. Khatti, X. Tian, Y. Chi, L. Guo, J. Cong, and Z. Fang, "Pasta: Programming and automation support for scalable task-parallel hls programs on modern multi-die fpgas," in *2023 IEEE 31st Annu. Int. Symp. on Field-Program. Custom Comput. Machines (FCCM)*. IEEE, 2023, pp. 12–22.

[32] J. Luo, X. Liu, F. Chen, and Y. Ha, "Hrff: Hierarchical and recursive floorplanning framework for noc-based scalable multi-die fpgas," *IEEE Trans. on Circuits and Syst. I: Regular Papers*, 2023.

[33] M. Mazraeli, Y. Gao, and P. Chow, "Partitioning large-scale, multi-fpga applications for the data center," in *2023 33rd Int. Conf. on Field-Program. Log. and Appl. (FPL)*. IEEE, 2023, pp. 253–258.

[34] T. Nguyen, Z. Blair, S. Neuendorffer, and J. Wawrzynek, "Spades: A productive design flow for versal programmable logic," in *2023 33rd Int. Conf. on Field-Program. Log. and Appl.* IEEE, 2023, pp. 65–71.

[35] Y. Xiao, D. Park, Z. J. Niu, A. Hota, and A. Dehon, "Exhipr: Extended high-level partial reconfiguration for fast incremental fpga compilation," *ACM Trans. on Reconfigurable Technol. and Syst.*, vol. 17, no. 2, pp. 1–28, 2024.

[36] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and effective placement and routing directed high-level synthesis for fpgas," in *Proc. of the 2014 ACM/SIGDA Int. Symp. on Field-Program. gate arrays*, 2014, pp. 1–10.

[37] F. Mao, W. Zhang, B. Feng, B. He, and Y. Ma, "Modular placement for interposer based multi-fpga systems," in *2016 Int. Great Lakes Symp. on VLSI (GLSVLSI)*. IEEE, 2016, pp. 93–98.

[38] S. Liu, J. Ke, T. Nowatzki, and J. Cong, "Demystifying fpga hard noc performance," *arXiv preprint arXiv:2503.10861*, 2025.