FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs

Linfeng Du
linfeng.du@connect.ust.hk
Hong Kong University of Science and Technology
Kowloon, Hong Kong

Tingyuan Liang
tliang@connect.ust.hk
Hong Kong University of Science and Technology
Kowloon, Hong Kong

Sharad Sinha
sharad@iitgoa.ac.in
Indian Institute of Technology Goa
Goa, India

Zhiyao Xie
eezhiyao@ust.hk
Hong Kong University of Science and Technology
Kowloon, Hong Kong

Wei Zhang
eeweiz@ust.hk
Hong Kong University of Science and Technology
Kowloon, Hong Kong

ABSTRACT

Multi-die FPGAs are widely adopted to deploy large-scale hardware accelerators. Two factors impede the performance optimization of high-level synthesis (HLS) designs implemented on multi-die FPGAs. On the one hand, the long net delay due to nets crossing die-boundaries results in an NP-hard problem to properly floorplan and pipeline an application. On the other hand, traditional automated searching flow for HLS directive optimizations targets single-die FPGAs, and hence, it cannot consider the resource constraints on each die and the timing issue incurred by the die-crossings. Further, it leads to an excessively long runtime to legalize the floorplanning of HLS designs generated under each group of configurations during directive optimization due to the large design scale.

To co-optimize the directives and floorplan of HLS designs on multi-die FPGAs, we propose the FADO framework, which formulates the directive-floorplan co-search problem based on the multi-choice multi-dimensional bin-packing and solves it using an iterative optimization flow. For each step of directive optimization, a latency-bottleneck-guided greedy algorithm searches for more efficient directive configurations. For floorplanning, instead of repetitively incurring global floorplanning algorithms, we implement a more efficient incremental floorplan legalization algorithm. It mainly applies the worst-fit strategy from the online bin-packing algorithm to balance the floorplan, together with an offline best-fit-decreasing re-packing step to compact the floorplan, followed by pipelining of the long wires crossing die-boundaries.

Through experiments on a set of HLS designs mixing dataflow and non-dataflow kernels, FADO not only well-automates the co-optimization and finishes within 693X~4925X shorter runtime, compared with DSE assisted by global floorplanning, but also yields an improvement of 1.16X~8.78X in overall workflow execution time after implementation on the Xilinx Alveo U250 FPGA.

FPGA '23, February 12–14, 2023, Monterey, CA, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9417-8/23/02...$15.00
https://doi.org/10.1145/3543622.3573188

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Partitioning and floorplanning**.

KEYWORDS

High-Level Synthesis, Design Space Exploration, Multi-Die FPGA, Directive Optimization, Floorplanning

ACM Reference Format:
Linfeng Du, Tingyuan Liang, Sharad Sinha, Zhiyao Xie, and Wei Zhang. 2023. FADO: Floorplan-Aware Directive Optimization for High-Level Synthesis Designs on Multi-Die FPGAs. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543622.3573188

1 INTRODUCTION

Guided by optimization directives, high-level synthesis (HLS) compiles high-level behavioral specifications to register-transfer level (RTL) structures, supporting the ever-growing functional and structural complexity of hardware accelerators. The various directives contribute to large design space to search upon. For example, there are 26 directives in Xilinx Vitis HLS 2020.2 [11], each of which has a set of parameters and can be applied at different levels or structures of the HLS source code. Previous works [18, 27, 28, 31, 32, 34, 37, 38, 41, 45] mainly use automated design space exploration (DSE) algorithms to search for the Pareto-optimal directive configurations, targeting the lowest latency (execution time in clock cycle) under a specific resource constraint.

To deploy large-scale HLS designs on FPGAs, with consideration of chip yield in fabrication, larger FPGAs with multiple dies emerge based on 2.5D/3D integration techniques. However, the concomitant long net delay due to nets crossing die-boundaries harms the timing quality of the implemented designs. One of the multi-die packaging technologies is the Stacked Silicon Interconnect (SSI) from Xilinx [30], where a silicon interposer integrates multiple dies, also called super logic regions (SLRs). [4] states that the super long lines (SLLs) between dies cause ~1 ns delay, while [25] states that a

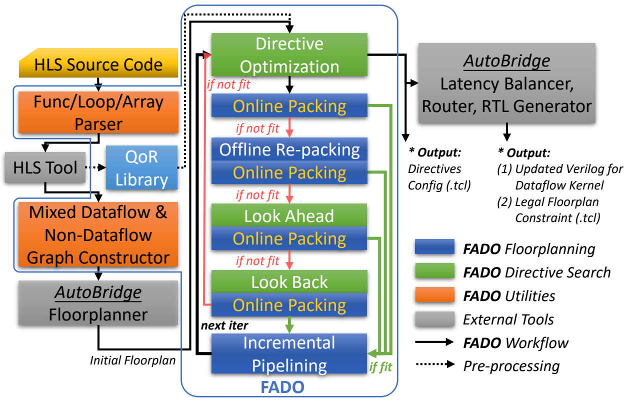


Figure 1: Overview of Our FADO Framework.

typical medium-length routing wire within a single die has 4X~8X shorter delay under the same manufacturing process. Further, [7] shows that for HLS dataflow designs, a handshake-based model for task-level parallelism, its floorplanning and pipelining can optimize the maximum achievable frequency (Fmax) on multi-die Alveo FPGAs to at most ~300 MHz, because of the delay of SLLs and long routes detoured by specialized IP blocks close to the I/O banks.

To mitigate the delay penalty on multi-die FPGAs, Xilinx proposes using the floorplanning method [14] to keep critical timing paths on a single SLR. However, the fine-grained gate/cell-level floorplanning is very time-consuming. In comparison, [7] requires that no function in the HLS dataflow region should spread over multiple SLRs and proposes a coarse-grained method to floorplan HLS functions at the SLR level to accommodate the large-scale dataflow designs and pipeline the wires crossing die-boundaries.

Min-cut floorplanning focuses on meeting the separate resource constraints on slots divided by SLR boundaries or I/O banks and the SLL number constraints between SLRs. However, an initial min-cut floorplan would not always support the latency-centric optimization of HLS directives. When a function's directive configuration changes, its resource also changes, and the original floorplan could be illegal because of resource over-utilization. For coordinating floorplanning with directive DSE, a simple combination is to solve the global floorplanning repeatedly, e.g., using mixed-integer linear programming (MILP) solver [7], whenever there's a new directive applied. This incurs an extended runtime of the DSE flow. Thus, we try to replace the global MILP solution with an incremental legalization algorithm to facilitate a highly-efficient integration of iterative directive search and floorplanning.

In this paper, we solve this challenge with a new framework named FADO [1], as shown in Fig. 1. It is the first work to co-optimize HLS directives and floorplanning on multi-die FPGAs, thus benefiting both latency and timing of HLS designs. We first formulate this complex co-optimization problem based on multi-choice multi-dimensional bin-packing (MMBP) [26], then develop an extremely efficient iterative solution. In each iteration, we apply a latency-bottleneck-guided greedy algorithm to search for more efficient directive configurations, followed by incremental floorplan legalization. Such legalization applies both worst-fit (WF) online bin-packing algorithm and the best-fit decreasing (BFD) offline algorithm [20]. The WF algorithm balances the resource utilization among slots on FPGA to avoid congestion, while the BFD algorithm breaks the balance with minimum cost to enable the floorplanning of overlarge HLS functions. At the end of each iteration, FADO incrementally adds/updates/removes the pipeline logic along the long wires crossing die-boundaries. This incremental floorplan update is much faster than global algorithms in previous works [7].

[1]The FADO framework for directive-floorplan co-optimization is fully open-sourced in https://github.com/RipperJ/FADO.

Table 1: Comparisons between FADO and Previous Work

	Directive Search	Multi-die Floorplanning	Floorplan-aware Directive DSE (**FADO**)
QoR	latency, resource	timing (frequency)	latency, resource, timing
Design Space	1. directives & parameters	2. SLR-level func location	1 & 2
DSE Efficiency	syn: slow; model: fast	slow (SA, MILP, bi-partition, ...)	fast (incremental floorplanning)
Type of Benchmark	dataflow or non-dataflow	dataflow	mixed dataflow & non-dataflow

In summary, FADO improves the overall performance of designs deployed on multi-die FPGAs with co-optimization and achieves high speed with its customized iterative solution. In our experiment, FADO can fully utilize resources on FPGA under all constraints. Also, FADO proves to scale well to large accelerators with both dataflow and non-dataflow kernels.

Our contributions in FADO are as summarized below:

- To the best of our knowledge, FADO is the first solution for co-optimization of HLS directives and floorplanning on multi-die FPGAs. It improves both latency and timing of complex implemented designs within a very short runtime.
- We propose the first precise mathematical formulation of this directive-floorplan co-optimization problem on multi-die FPGAs in FADO. Its solution is based on a well-customized iterative algorithm, which finishes in seconds, achieving orders-of-magnitude speedup over global algorithms in prior works, while producing even better final design quality.
- Compared with the directive DSE with the global MILP floorplanning and pipelining [7], FADO achieves 693X~4925X speedup with an even higher and near-optimal final design performance on FPGA for all six tested designs. The performance measured with overall workload execution time on each design improves 1.16X~8.78X.
- Compared with [7], our FADO framework can automatically handle not only dataflow benchmarks but also large-scale applications mixing dataflow and non-dataflow kernels.

2 RELATED WORK

As shown in Table. 1, FADO performs co-optimization considering latency, resource, and timing during floorplanning, while previous commonly used flows, such as directive search and multi-die floorplanning, only target one or two optimization objectives. To achieve multi-objective optimization, the design space is enormous, defined by the Cartesian product of directives, their respective parameters, and SLR-level function locations. Previous works may take a long time to traverse such a large design space. However, with our effective incremental floorplanning, FADO achieves orders-of-magnitude speedup in the search time. Moreover, existing floorplanning works for HLS designs [2, 7] are dedicated to dataflow applications. FADO is able to automatically solve the floorplanning for non-dataflow functions by adding additional constraints.

Table 2: Objectives of Multi-die FPGA Timing Optimizations

Objectives	Previous Works
Total Wirelength	[6, 21, 44]
Signal Delay	[8, 21, 25]
Number of Cut Net (SLL)	[7, 8, 25, 29]
Routing Congestion	[2, 8, 25, 29]
Aspect Ratio	[6, 29]

HLS Directive Optimization has been researched thoroughly. The featured challenges are listed below.

- Directives and their parameters contribute to an enormous design space [35]. Accordingly, we apply a latency-bottleneck-guided algorithm to minimize the overall latency and speed up the DSE effectively.
- Directive configurations have non-monotonic effects [35] on QoRs, which is also explained by [42] as inter-dependency among directives and structures in the source code. To avoid getting trapped in local optima on the non-monotonic design space, we design the look-ahead/back sampling methods.

The general techniques for HLS directive DSE includes (1) meta heuristics [31, 32, 40], (2) dedicated heuristics [27, 28, 42], (3) machine learning [24, 37, 43], and (4) graph analysis [34, 38, 41, 45]. Prior work on directive search mainly optimizes latency under an overall resource constraint of single-die FPGAs. To compare, multi-die FPGAs introduce separate constraints on each slot and between SLRs, and also the vital timing issue because of long wires crossing die-boundaries. Hence, we cannot directly apply the previous DSE algorithms to our co-optimization problem.

Multi-die FPGA Timing Optimization can be classified by their objectives as Table. 2 shows. [6] proposes optimization on total wirelength and aspect ratio of face-to-face-stacked floorplans. [21] minimizes total wirelength while reducing total and die-crossing delay. [44] proposes constructive floorplan optimizations dedicated to PEs of systolic arrays. [8] and [25] extend the P&R tool VPR [3] to multi-die scenario by adding parameters to the cost function including wire-cut ratio, delay increment, and cut number, while [25] also considers the congestion cost. [7] applies MILP to minimize the number of die-crossing long wires. It runs iterative bi-partitioning rather than N-way partitioning and prefers the most balanced floorplan across all slots divided by die-boundaries and I/O banks. [29] implements a partition-driven placer and an aspect-ratio-aware cut scheduling algorithm. [2] also applies ILP and resource balancing heuristics to partition dataflow accelerators and average congestion among different SLRs. It also discusses partitioning dataflow accelerators among multiple FPGAs through network interfaces.

In FADO, we mainly compare with [7] on the efficiency of coarse-grained floorplanning because fine-grained floorplanning in other works is too time-consuming for large-scale designs, not to mention floorplanning repetitively during iterative DSE. We identify that the major frequency improvement in [7] comes from the insertion of pipelining logic, while min-cut floorplanning mainly performs legalization for logic resources and die-crossings. Thus, we replace the timing-consuming min-cut MILP floorplanning with an incremental legalization algorithm with a partial pipelining update to speed up the DSE without sacrificing floorplanning quality.

Knapsack [23] and **Bin-Packing Problems** [22] are a series of classic combinatorial optimization problems having a common ground with the directive-floorplan co-search. The basic version is the 0-1 Knapsack problem, where multiple items with different weights and values are to be packed in a knapsack, and a binary choice is made for packing the item or not. [16, 33] introduces the multiple-choice Knapsack problem, where the items are classified, and exactly one from each class is chosen to form a solution. The classes here map to the directive configurations for an HLS function in our problem. [17] introduces the multiple-dimensional Knapsack problem, where the weight of each item and the capacity of knapsacks are in vectors. This corresponds with the types and amounts of resources on each die of a multi-die FPGA. [1, 26] separately formulates the multi-choice multi-dimensional Knapsack problem (MMKP) and bin-packing problem (MMBP). The optimal solution to this problem can be found using branch-and-bound with linear programming, but the high time complexity does not support a large number of variables and equations. Another approximation is using greedy approaches, generally sorting items based on the values and the weights in a certain order. We thus formulate our problem based on the MMBP and combine online Worst Fit (WF) and offline Best-Fit Decreasing (BFD) [20] bin-packing heuristics to solve it efficiently. Here, in the online algorithm, the decision of packing an item is irreversible, and the next item is only visible after the previous packing is settled. For offline algorithms, the value and weight of all items are visible from the very beginning, and we can sort the items to improve the packing quality. [15]

3 MOTIVATION

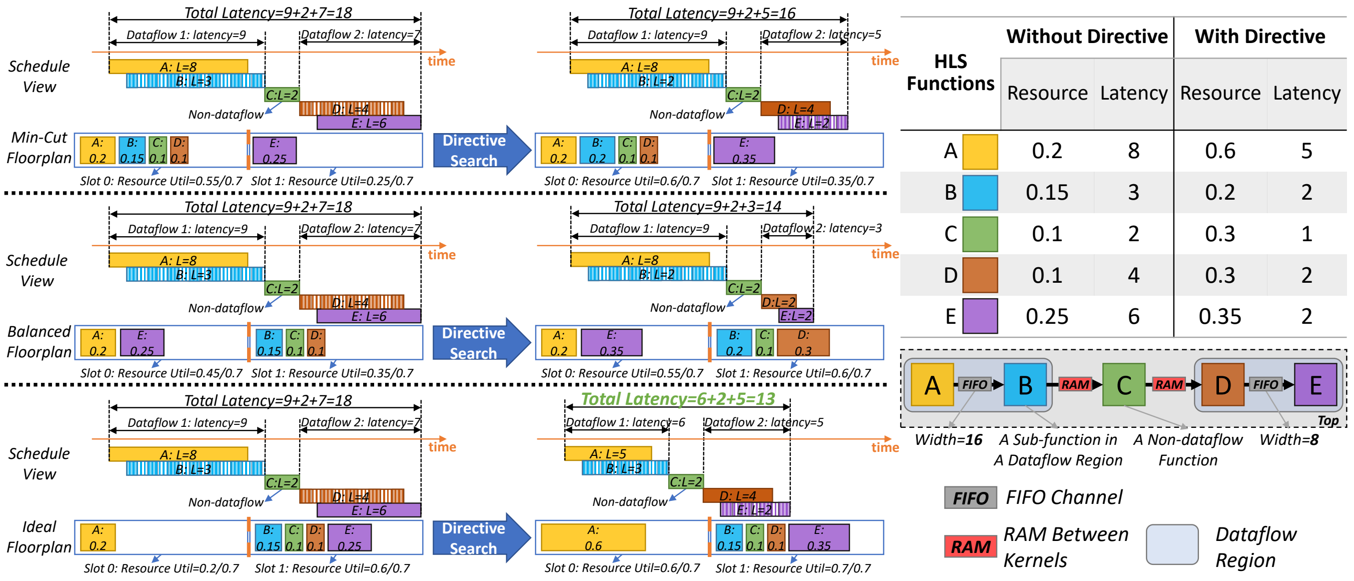
To show the different challenges of the directive-floorplan co-search problem on multi-die FPGAs, we use a toy example with one step of floorplanning followed by directive optimization to explain why general floorplanning algorithms and heuristics won't collaborate with directive search.

Suppose that we have a toy multi-die FPGA with two slots, each having a constraint of 70% of the total available resource, as suggested by [2, 7]. Although there are several different resources on a modern FPGA, such as Look-up Tables (LUT), Flip-Flops (FF), Digital Signal Processors (DSP), Block-RAMs (BRAM), UltraRAMs [10] (URAM), etc., in our experiments, we normalize and take the maximum among all resources as the final utilization ratio in evaluation.

Fig. 2 shows a design consisting of 2 dataflow and 1 non-dataflow kernel. A/B (or D/E) are functions connected by the FIFO channels in the same dataflow kernel (region), defined as an HLS function with directive "DATAFLOW", achieving task-level parallelism within the function through a handshake-based model. C is a non-dataflow function connected through RAM to other kernels. The channel width between A and B is 16, and that between D and E is 8.

For latency optimization, different QoRs are caused by various directive configurations in HLS. For example, when applying a smaller initiation interval (II) to the directive *PIPELINE*, or a larger factor to the *UNROLL*, the latency of an HLS design tends to decrease, while the resource utilization is likely to increase. In this example, assume that we only have one directive — two configurations for each function independently, either with or without that directive. When the directive is applied to a function, its resource consumption increases and latency decreases, as shown in Fig. 2.

For timing optimization, a design's frequency can be ensured at a high level by pipelining as long as the following floorplan conditions are satisfied. The first is having a legal floorplan which



HLS Functions		Without Directive		With Directive	
		Resource	Latency	Resource	Latency
A	🟨	0.2	8	0.6	5
B	🟦	0.15	3	0.2	2
C	🟩	0.1	2	0.3	1
D	🟧	0.1	4	0.3	2
E	🟪	0.25	6	0.35	2

Figure 2: A Toy Example with 2 Dataflow Kernels and 1 Non-dataflow Kernel, with Different Latency (L) and Resource Consumption (R). Three Different Floorplanning Methods and the Corresponding DSE Results Are Compared.

meets the resource constraint on every single slot. Second, only FIFO channel connections are allowed to cross the slot boundaries (within a limit on total width not reflected in this toy example) because the handshake interface of FIFO is easy to pipeline, while the complex RAM interface cannot be pipelined. Thus, when functions are connected through RAM, they should be grouped and floorplanned on the same slot, while functions connected through FIFO channels can be partitioned on different slots.

During directive DSE, we minimize the total latency of the design while ensuring the two floorplan conditions above. Suppose that every function has no directive applied at the beginning of DSE. We find an initial floorplan using a specific algorithm first and then try to improve some functions by applying their respective directive, subject to the resource constraint on each slot.

Fig. 2 shows the three floorplanning objectives compared in this toy example. The first minimizes the width of the FIFO channel crossing two slots (min-cut), as used in [7]. Since all functions cannot be packed in one single slot, the solution to the min-cut problem is 8, which is the width of the channel between D and E. Thus, function E should be assigned to the other slot. In this case, functions B and E can still fit into their slots when applying directives, and the final total latency is improved to 16.

The second floorplan refers to the heuristics of resource balancing [2]. Since functions B, C, and D are grouped during floorplanning, as the largest one, they are floorplanned onto the other slot initially. After DSE, directives are applied for functions B, D, and E, and the total latency is improved to 14.

To compare, an ideal floorplan for this design is to partition between A and B, and the best point improves total latency from 18 to 13 clock cycles, which is the minimum achievable latency for this case. If we have an ideal floorplan at the very beginning, it's natural for DSE to reach the optimal latency without any effort to change the floorplan. However, if we start with the other two floorplans, neither the min-cut nor balancing algorithm can further improve the achieved latency. In our FADO framework, the incremental floorplanning algorithm smartly re-packs the functions from either of the two prior floorplans and always reaches the ideal solution finally. To be specific, if FADO starts with the min-cut solution, A is identified as the latency bottleneck and has the top priority to apply the directive. When online packing finds no legal floorplan for A under the min-cut floorplan, the offline re-packing stage groups B, C, D, and E. Thus, the ideal floorplan is found, and the directive for A is successfully applied. It's a similar workflow if we start with the balanced floorplan. In Sec. 6, the control experiments on real benchmarks all start with a min-cut floorplan, to fairly compare the effectiveness of FADO with MILP floorplanning assisted DSE.

From this example, we want to show that previous floorplanning techniques could fail to assist directive optimization on multi-die FPGAs. On the contrary, an improper floorplan could prune the high-performance points in a design space. That's why we propose floorplan-aware directive optimization, the FADO framework.

4 PROBLEM FORMULATION

Based on the multi-choice multi-dimensional bin-packing problem, we formulate the directive-floorplan co-search problem on multi-die FPGAs. To accurately describe the problem and show the complexity compared with floorplanning in [7], we also present a MILP formulation below. Note that MILP is only used for a description of the problem. In the implementation (Sec. 5), we are using approximation heuristics for the objective and each constraint instead of repetitively calling the MILP solver.

4.1 Symbol Definition

Table 3 shows the domain and definition of all the variables used in our formulation.

4.2 MILP Formulation

The objective function in our problem minimizes the total latency of an HLS design. To show the generality of our problem, assume that we have a large accelerator containing multiple dataflow and non-dataflow kernels connected by RAMs. Minimizing the total

Table 3: Symbols Used in Problem Formulation

Symbols	Definition
L_{ij}	The latency of the j-th function in the i-th kernel along the longest path. $i \in \{1, 2, ..., m+n\}$, $j \in \{1, 2, ..., C_i\}$. $C_i = 1, \forall i \geq m+1$.
Z_{ijp}	Suppose that function j or kernel i has Q_{ij} directive choices in total. $Z_{ijp} = 1$ when directive choice p is applied to function j of kernel i. $p \in \{1, 2, ..., Q_{ij}\}$.
x_{ijk}	$x_{ijk} = 1$ when the sub-function j of kernel i is floorplanned to slot $k \in \{1, 2, ..., S\}$ among S slots.
r_{ijpt}	r_{ijpt} represents the consumption of resource type t of function j in kernel i, when directive choice p is applied. $t \in BRAM, DSP, FF, LUT, URAM$.
R_{kt}	The total amount of resource t on slot k.
$e_{i1,j1,i2,j2}$	$e_{i1,j1,i2,j2} \in \mathbb{N}$. It's positive when there exists a RAM connection from function $j1$ in kernel $i1$, to $j2$ in kernel $i2$, with a width of $e_{i1,j1,i2,j2}$. $i1, i2 \in \{1, 2, ..., m+n\}$, $j1 \in \{1, 2, ..., C_{i1}\}$, $j2 \in \{1, 2, ..., C_{i2}\}$.
W, H	Width/Height of an FPGA (by number of slots).
$b_{h(k,ks,kd)}$	$b_h \in \{0, 1\}$, it equals 1 when a source function on slot ks goes through the die boundary indexed with k and connects to a destination function on slot kd. $k = (k_x, k_y), k_x \in \{0, 1, ..., W-1\}, k_y \in \{0, 1, ..., H-2\}$.
B_h	Total amount of SLLs on a die boundary.

latency is equivalent to minimizing the sum of latency of kernels L_i along the longest path, as Eq. 1 shows.

$$\text{minimize} \sum_{i=1}^{N} L_i \tag{1}$$

Generally, a dataflow kernel's total latency is very close to the longest sub-function $\max_j L_{ij}$ in it, meanwhile relatively much smaller latency comes from the depth of dataflow — $\max_j L_{ij} \gg L_{depth}$. We here approximate the total latency of a dataflow kernel i with the latency of the longest sub-function.

For the longest-latency path with m dataflow kernels and n non-dataflow kernels, the objective function can be re-written as:

$$\text{minimize} \sum_{i=1}^{m} \max_j L_{ij} + \sum_{i=m+1}^{m+n} L_i \tag{2}$$

where i iterates on kernels, and j on sub-functions. There's no sub-function in non-dataflow kernels indexed from $m+1$ to $m+n$.

Table 4: Directives in the Design Space of FADO

Directives	Parameters
PIPELINE	Initiation Interval (II) (<int>: $\{MinII, ..., \lfloor 4 \times MinII, IterLat \rfloor\}$)
UNROLL	Factor (<int>: $\{1, 2, 4, ..., LoopBound\}$)
ARRAY_PARTITION	Type (Block/Cyclic/Complete) Dimension (<int>)
BIND_STORAGE	Implementation (BRAM/URAM)

4.2.1 *Constraint 1: multi-choice packing problem.*

During the directive search, we have multiple choices of directives and parameters for HLS functions, loops, and arrays. The directive design space of FADO is shown in Table 4. For *PIPELINE*, the lower bound of II, *MinII*, is determined by recurrence and resource analysis, and it's also revealed by the HLS report when applying the minimum value possible for target II, i.e., 1. The upper bound considers the iteration latency and four times the *MinII*. For *UNROLL*, the applicable value for its factor ranges from 1 to the loop bound. For *ARRAY_PARTITION*, we consider the three types of partitioning schemes and the dimension of an array. For *BIND_STORAGE*, an array is either implemented using BRAM or URAM.

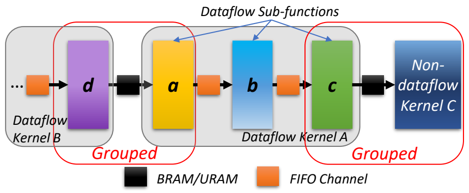


Figure 3: Grouped Floorplan for RAM-Interfaced Functions.

Every time we trigger the HLS, one group of directives and the corresponding QoR are applied for each function (including the loops and arrays within it), described as the constraint in Eq. 3.

$$\sum_{p=1}^{Q_{ij}} Z_{ijp} = 1, Z_{ijp} \in \{0, 1\} \tag{3}$$

4.2.2 *Constraint 2: multiple bins.*

During floorplanning, multi-die FPGA is partitioned into several slots by the die boundaries and I/O banks. Eq. 4 guarantees that there's no duplicated or missing floorplan for each HLS function.

$$\sum_{k=1}^{S} x_{ijk} = 1, x_{ijk} \in \{0, 1\} \tag{4}$$

4.2.3 *Constraint 3: multi-dimensional packing problem.*

In our problem, a single dimension of resource constraints corresponds with one type of resource on the multi-die FPGAs. The following constraint in Eq. 5 assures that functions on each slot with specific directive configurations respectively will not cause resource overflow in any type of resource.

$$\sum_{i=1}^{m+n} \sum_{j=1}^{C_i} x_{ijk} z_{ijp} r_{ijpt} \leq R_{kt} \tag{5}$$

4.2.4 *Constraint 4: grouping the RAM-connected functions.*

Another special type of constraint is introduced by RAM connection between kernels, as shown in Fig. 3. Since the interface between RAMs and functions is not the handshake model and is difficult to pipeline, the RAM-connected functions are grouped and assigned to the same slot during floorplanning. As Equation 6 states, $e_{i1,j1,i2,j2} x_{i1j1k1} x_{i2j2k2} = 0$ guarantees that either there's no RAM connection between two functions, or they are not separated on two different slots.

$$\begin{aligned} &e_{i1,j1,i2,j2} x_{i1j1k1} x_{i2j2k2} = 0, (i1 \neq i2 \vee j1 \neq j2) \wedge k1 \neq k2 \\ &e_{i1,j1,i2,j2} \in \mathbb{N}_+, i1 \neq i2 \vee j1 \neq j2 \end{aligned} \tag{6}$$

4.2.5 *Constraint 5: Limited number of SLLs.*

As Fig. 4 shows, Alveo U250 FPGA is vertically partitioned into two parts by the I/O banks and horizontally partitioned by die-boundaries into four parts. Suppose we have a source function "s" and a destination function "d" placed on SLR3:Slot0 and SLR0:Slot1, respectively. No matter which route between "s" and "d" is chosen, it crosses three horizontal die boundaries. For each boundary with vertical index k_y, the route passes through either the left half or the right half of it. The corresponding constraint is:

$$\sum_{k_x=0}^{W-1} x_{i1j1k1} x_{i2j2k2} b_{h(k,k1,k2)} \operatorname{sgn}(e_{i1,j1,i2,j2}) = 1 \tag{7}$$

Since the number of SLLs is limited between two dies, we have the formulation in Eq. 8.

$$\sum_{i1,j1,i2,j2,k1,k2} x_{i1j1k1} x_{i2j2k2} b_{h(k,k1,k2)} e_{i1,j1,i2,j2} \leq \beta B_h, \tag{8}$$

where β is the upper limit of SLL utilization. It is set to 90% in our implementation.

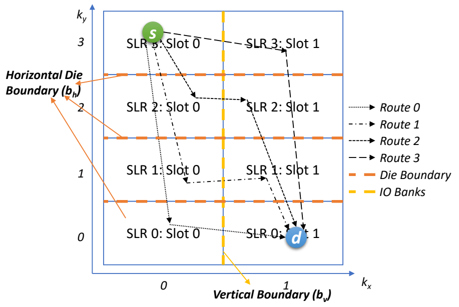


Figure 4: An Example about Different Routes between Two Functions Placed on Two Separate Slots of Alveo U250 FPGA.

5 IMPLEMENTATION

To solve the MILP formulation above by the FADO framework (Fig. 1), we describe the objective and multi-choice constraint (Eq. 3) in the DSE algorithm (Sec. 5.3), and other constraints in the floorplanning algorithms (Sec. 5.4). The interaction between FADO and external tools is explained in Sec. 5.6.

5.1 Pre-processing

In a large-scale HLS design, many function blocks could be based on the same template. When running an HLS over the entire design, excessively long synthesis time is wasted on analyzing those functions from the same template repeatedly. Besides, it is hard for existing graph analysis or machine learning methods to accurately predict the QoR of an HLS design with various coding styles, complicated control and data flow, and flexible compilation optimization.

Considering the small size of each function template, it only takes a short period to sample their directive choices and build a function-level QoR library to facilitate the whole workflow. To be specific, we classify the functions by either the template of C++ generics, or custom naming rules, e.g., all functions whose names match the regular expression *r'funcA_[0-9]_[0-9]'* will map to *'funcA'* in the QoR library. To apply directives to the C++ template, we follow [12]. As for custom regex, it's straightforward since they have distinct names. Accordingly, we only need to look up for QoRs in the library instead of running HLS flow for an entire design repeatedly.

5.2 Booting of FADO

As Fig. 1 shows, at the very beginning, the input to FADO is an HLS design without any directive of *PIPELINE*, *UNROLL*, *ARRAY_PARTITION*, and *BIND_STORAGE*. We parse the source code in "Func/Loop/Array Parser" to generate labels and hierarchy for nested loops, and to identify functions of the same template to build a QoR library *QoR_lib* in pre-processing. In the other workflow, the labeled code is synthesized by an external HLS tool. Then, the HLS report is analyzed by a graph constructor, where connections through FIFOs and RAMs are identified. Then, the graph is passed to the min-cut floorplanner of AutoBridge [7] to generate an initial legal floorplan, which is then passed on to FADO for directive-floorplan co-optimization.

5.3 Directive Optimization

The Top-Level Algorithm of floorplan-aware directive optimization is described in Alg. 1. In every iteration of floorplan-aware directive optimization, we identify the functions with the longest and second-longest latency among the whole HLS design, i.e., the sub-function j_1 of kernel i_1 is the bottleneck, represented by $(i_1, j_1) = \text{argmax}^1_{i,j}(L_{ij})$, and the second-longest function is (i_2, j_2). We apply the latency-bottleneck-guided search [18, 45] in Prune(), which extracts all design points of (i_1, j_1) with smaller latency compared with the second-longest function's $L_{(i_2,j_2)}$ to form a next-step design space *DS* (a set of directive configurations and their respective QoRs). Although applying any of the configurations in *DS* to function (i_1, j_1) would make (i_2, j_2) the new latency bottleneck of the whole design, we choose the design point *DP*, which has the largest latency among *DS*, for further floorplan legalization.

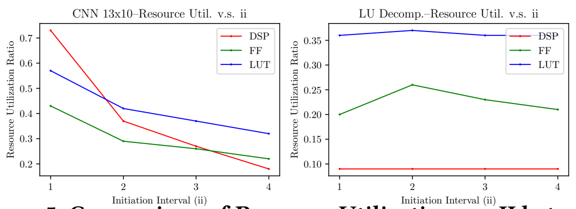


Figure 5: Comparison of Resource Utilization v.s. II between CNN and LU Benchmarks.

Algorithm 1: Top-level of FADO framework

```
Input: $QoR\_lib$, $N$: look-ahead step number
Output: Optimal Directives, Legal Floorplan
while True do
    $(i_1, j_1) = \text{argmax}^1_{i,j}(L_{ij})$                    // the longest functions
    $L_{(i_2,j_2)} = \max^2_{i,j}(L_{ij})$                         // the 2nd-longest latency
    $DS$ = Prune($QoR\_lib$, $(i_{max,1}, j_{max,1})$, $L_{i_{max,2},j_{max,2}}$);
    $DP = DS[-1]$, break if $DP$ is $None$                        // Constraint Eq. 3
    $fit, incrfp\_list$ = online_packing($(i_1, j_1)$, $DP$);
    if not fit then
        offline_repacking, online_packing
        if not fit then
            iterative look_ahead($N$), online_packing
            if not fit then
                iterative look_back(), online_packing
    incremental_Floorplan($incrfp\_list$);
    exit_condition_check();
```

To compare, FADO will not make one-off latency improvements for bottleneck functions or choose the design point with the lowest resource utilization. On the one hand, aggressive latency improvement usually results in a dramatic increase in the resource utilization of current function(s), and potential latency improvement for future bottlenecks could be precluded because of a lack of resources. On the other hand, since resource utilization is calculated by taking the maximum ratio among different resources, considering the non-monotonic design space, when utilization of one resource is minimized, others could still increase. Hence, we always assign the top priority to *DP* for floorplanning. Note that functions having the same latency with (i_1, j_1) are considered as a batch for efficiency.

Look-Ahead and Look-Back. Guided by the latency bottleneck, the main algorithm prunes the ineffective design points with negligible improvement in the overall latency. However, in realistic HLS designs, the greedy algorithm could get stuck in local optima. Fig. 5 shows the different trend of resource utilization as the *PIPELINE* initiation interval (II) changes in two designs, CNN from [5] and LU from [36]. As II increases, latency also increases in both designs. It results in less utilization of the three types of resources in the CNN benchmark because computation instances are shared among multiple cycles. However, in contrast, there are a lot of loop-carried

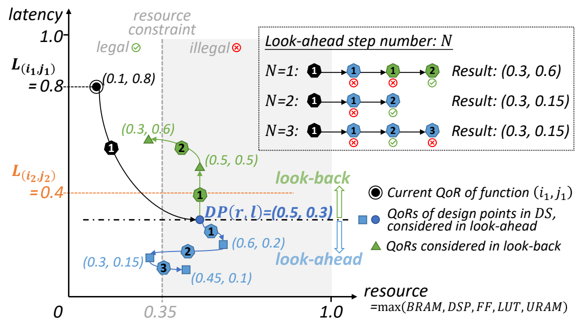


Figure 6: Effect of Look-Ahead with Step # N, and Look-Back.

dependencies in the LU benchmark, the results from the previous iteration cannot be directly passed to the next, and extra logic is used to buffer the results. Together with the effect of resource sharing, the utilization first increases as II increases from 1 to 2, and then decreases when II continues increasing.

To handle the non-monotonic design space, we propose the sampling methods of `look_ahead()` and `look_back()`. When the first two stages of floorplanning — online packing and offline re-packing fail to find a legal floorplan for *DP*, we further check the floorplan for a certain number of design points with lower latency yet potentially fewer resources. This is referred to as the look-ahead stage. If it still fails in floorplanning, we turn to check the points with larger latency than *DP* in the look-back stage. These points are more likely to have lower resource utilization and a legal floorplan.

Fig. 6 shows a snapshot of directive search for the current bottleneck function (i_1, j_1). QoR values are normalized for clarity. The design point with the top priority for floorplan checking is *DP*, with a resource utilization of 0.5 and a latency of 0.3 (the longest latency smaller than $L_{(i_2,j_2)} = 0.4$). However, only 0.35 resource is left for the current function, and there's no legal floorplan found for *DP* during online packing and offline re-packing. We now look ahead/back for other improvement opportunities with fewer resources. When we set the step number N to 1 during `look_ahead()`, the next design point consumes 0.6 utilization and also fails to be floorplanned. Thus, when `look_ahead()` also fails to find a point with a legal floorplan, `look_back()` traverses all the points with latency from 0.3 to 0.8. When N is set to 2 or 3, the directive configuration with a latency of 0.15 is found during `look_ahead()`.

For HLS designs, our implementation decides the step number N of `look_ahead()` by analyzing the range of parameters for *PIPELINE* and *UNROLL*, two of the most effective directives. We define the N as the largest number of different configurations on a single nested loop. To exclude the directives over-utilizing resources, we check at most three levels for each nested loop from the innermost level. For each nested loop of n levels, we index the innermost loop with 1, and the outermost loop with n. (1) For directive *PIPELINE*, since [13] suggests a maximum loop bound of 64 for auto pipelining, we set the range of II to the logarithm of the minimum between 64 and the iteration latency IL from HLS report. (2) For directive *UNROLL*, similar to *PIPELINE*, we take the minimum between 64 and the loop bound B. (3) For the combination of *PIPELINE* and *UNROLL*, since all the inner loops are completely unrolled when an outer loop is pipelined, we consider only the directive combination in 2 levels of loops. In all, the resulting step number N for a design with m nested loops is:

Algorithm 2: Online Packing

```
Input: $longest\_functions$ and QoRs $L_{ij}, R_{ij}$, design point $p$
Output: $fit$, $incrfsp\_list$
$fit$ = False, $incrfp\_list$ = [], $unfit\_funcs$ = [];
for $func$ in $longest\_functions$ do
    if $func$ still fits in the current slot $s_c$ then
        update directives for $func$, and resource util. for $s_c$;
        $L_{ij}, R_{ij} = L_p, R_p$; $fit$ = True;
    else
        // check constraint Eq. 6, Eq. 7 and Eq. 8
        calculate overflow ratio, and sort $other\_slots$ by $CR$;
        for each $s_o$ in $other\_slots$ do
            if no overflow when moving $func$ to $s_o$ then
                // check constraint Eq. 5
                update directives for $func$;
                update util. for $s_o, s_c$;          // constraint Eq. 4
                append $(func, s_o)$ to $incrfp\_list$;
                $L_{ij}, R_{ij} = L_p, R_p$; $fit$ = True; break;
        if not fit then
            append $func$ to $unfit\_funcs$;
if any func in unfit_funcs then
    clear $incrfp\_list$; $fit$ = False;
```

$$
\begin{aligned}
N_1 &= \max_{1 \le i \le m} \sum_{j=1}^{\max(3,n_i)} \log_2 \min(64, IL_{ij}) \\
N_2 &= \max_{1 \le i \le m} \sum_{j=1}^{\max(3,n_i)} \log_2 \min(64, B_{ij}) \\
N_3 &= \max_{1 \le i \le m} \sum_{j=1}^{\max(2,n_i)} \log_2 \min(64, B_{ij}) \\
N &= N_1 + N_2 + N_3
\end{aligned}
\tag{9}
$$

5.4 Incremental Floorplanning

The initial floorplan input to FADO is generated by an iterative min-cut MILP bi-partitioning in the "AutoBridge Floorplanner [7]" shown in Fig. 1. During FADO's iterations, we apply a resource-bottleneck-guided online WF algorithm. When the online packing fails in finding a legal floorplan, an offline BFD re-packing compacts the existing floorplan before calling the online packing again. The definition of "online" and "offline" algorithms refers to [15]. During online packing, HLS functions are optimized one after another, and the previous floorplan of a function will be kept unchanged. In contrast, without applying new directives, the offline stage reorder all functions by heuristics to improve the packing quality.

5.4.1 Online Packing. To avoid routing congestion from a high-abstraction view, the online packing tends to balance the utilization ratio among different resources and different slots, i.e., if a function fails to fit into its original slot after applying a new directive configuration, we try to floorplan it into other slots according to the non-decreasing order of critical resource (CR). CR refers to the type of resource having the most overflow percentage among BRAM, DSP, FF, LUT, and URAM. If there are multiple slots with the same CR, we sort them by the average utilization of the other four non-critical resources. The online packing algorithm is shown in Alg. 2. Note that overflow ratios are calculated by each resource.

5.4.2 Offline Re-packing. Since `online_packing()` tends to spread functions evenly on each slot, and when there's an aggressive move in directive search with a sharp increase in resource utilization, the

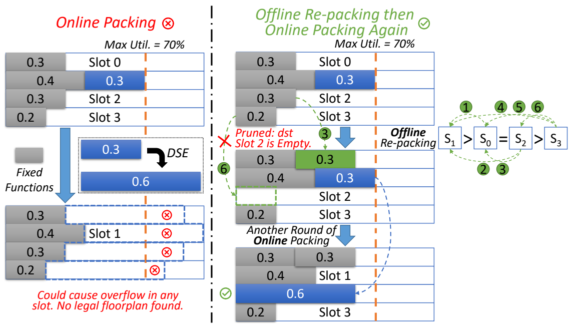

Online Packing
Max Util. = 70%
Slot 0
Slot 1
Slot 2
Slot 3
DSE
Fixed Functions
Could cause overflow in any slot. No legal floorplan found.
Offline Re-packing then Online Packing Again
Pruned: dst Slot 2 is Empty.
Offline Re-packing
Another Round of Online Packing

Figure 7: An Example of Offline Re-packing.

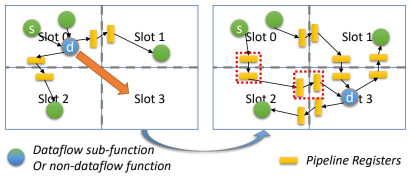


Figure 8: Incremental Update of Pipeline Registers.

balance could preclude the new design point from taking effect. Thus, offline re-packing sorts all the slots SL_i by resource utilization in non-increasing order. Then, it respectively sorts all the functions F_{ij} on each slot SL_i by resource in non-increasing order as well. The re-packing starts with moving the F_{21} from the second fullest slot SL_2 to the fullest SL_1, and then the second largest function F_{22} from SL_2 to SL_1, etc. When SL_1 is full, or SL_2 is empty, we turn to move functions F_{31}, F_{32}, etc., from the third fullest slot SL_3 to SL_1, then to SL_2. A general step of re-packing the m-th fullest slot is to move $F_{m1}, F_{m2}, ..., F_{mn_m}$ to $SL_1, SL_2, ..., SL_{m-1}$ in turn.

Fig. 7 shows the floorplanning of 5 functions onto 4 slots. We reduce multiple resources to one dimension by taking the maximum ratio among them. Through directive search, the resource of the blue function expands from 0.3 to 0.6. However, since other gray functions were fixed during the online stage, the expanded function fits nowhere. To compare, the re-packing stage sorts the slots by utilization in non-increasing order and executes six trials in order. Trials 1/2 fail because slot 1 is full. During trial 3, a gray function is moved from slot 2 to slot 0. Trials 4/5 also fail because the destinations, slots 0 and 1, are full. Trial 6 is canceled because the destination slot 2 has been found empty. With re-packing, the expanded function fits in slot 2 after a new round of online packing.

Re-packing applies different strategies for functions in dataflow kernels and non-dataflow kernels. The floorplan change is free for dataflow functions if the SLL utilization constraint is met. For non-dataflow parts, since their connections with other adjacent functions are not through the FIFO channel, the long wires could not be broken by inserting pipeline logic. Hence, instead of moving them, we try to force other dataflow functions that have no RAM connection with them to different slots. Thus, FADO assigns more resources to the slots containing non-dataflow for further DSE.

5.5 Incremental Pipelining

When moving a function across slots, as Fig. 8 shows, each time a path crosses a boundary between two slots (SLR boundary or I/O banks), additional pipeline registers should be added beside the boundaries to break down the long wires. We here set a constraint of 90% (Eq. 8) for SLL utilization and incrementally update the pipelining logic of long wires crossing slot boundaries. As Fig. 8

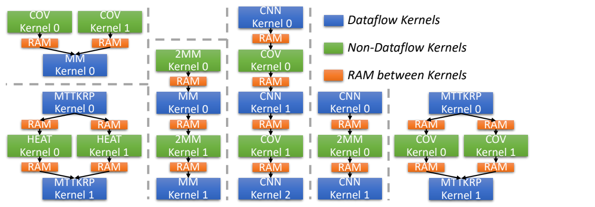


Figure 9: The 6 Benchmarks Used to Evaluate FADO.

shows, when function "d" is moved from Slot 0 to Slot 3, two groups of additional pipeline registers are added between "s" and "d".

5.6 Exit Condition and External Tools

During iterative optimization, when there's no legal floorplan found for the next design point of the current longest function or when no other directive configuration could improve the bottleneck's latency further, it is excluded in future iterations. FADO stops when there's no function left for the bottleneck analysis. Then, it dumps the optimal directives to a TCL file to guide the re-synthesis of the HLS code to generate a high-performance RTL design. FADO also delivers the final floorplan to the global router, latency balancer, and dataflow RTL generator within [7] to update the pipelining of dataflow kernels in the Verilog code, and generate another TCL script to guide the floorplanning during implementation in Vitis.

6 RESULTS

6.1 Benchmarks and Experiment Settings

We mainly adopt large-scale open-source HLS designs with compatible interfaces for evaluation and filter out many commonly used but unsuitable benchmarks. To be specific, most of the designs in Vitis Libraries [39], CHstone [9], Rosetta [46], etc., occupy less than 10% of resources on the Alveo U250 FPGA. They only have several functions to consider during coarse-grained floorplanning, which is not challenging even if we increase the design size, e.g., by applying a larger bitwidth. Besides, interface incompatibility makes it difficult to scale up by connecting multiple designs from these benchmarks. Hence, we generate large dataflow *CNN*, *MM* and *MTTKRP* using PolySA [5] and AutoSA [36]. As for non-dataflow designs, we use *2MM*, *COV*, and *HEAT* from Polybench [19], which are general programs also used in CPU, GPU, etc. To best show the generality of our solution, we assemble six large benchmarks mixing the dataflow and non-dataflow kernels above to evaluate the performance of our framework, as Fig. 9 shows. The kernels connect through RAMs, which enlarges the design space compared with a single dataflow kernel.

To show the scale of our problem, we visualize the HLS-function-level data flow graph of the *CNN*2+2MM*1* benchmark in Fig. 10. The two yellow bounding boxes mark the two *CNN13x2* dataflow kernels, each containing hundreds of sub-functions. The red circles on the top of this figure are the non-dataflow *2MM* kernel and the two RAMs connected to it. The RAM module "temp_xin1_V_U" is connected to two input sub-functions of *CNN13x2* Kernel 1, and RAM "temp_xout0_V_U" is connected to one output sub-function of *CNN13x2* Kernel 0. Since the connection among them are not through FIFO channels, they are grouped during floorplanning and always placed in the same slot. As for a dataflow kernel, the green boxes are FIFO channels, and the blue circles are dataflow sub-functions. Dataflows can be partitioned, floorplanned, and pipelined on any slot as long as the resource constraints are met. The overall

Table 5: QoR Comparison between FADO and Other DSE Strategies

Benchmarks	CNN*2+2MM*1					MM*1+COV*2					MTTKRP*2+HEAT*2				
	Resource[a]	Runtime[b]	Latency[c]	Fmax[d]	**Exe_time**[g]	Resource	Runtime	Latency	Fmax	**Exe_time**	Resource	Runtime	Latency	Fmax	**Exe_time**
Original (no directive)	28%	-	8933	-	-	20%	-	131839	-	-	57%	-	8147919	-	-
Initial FP[e] -> Iterative DO[f]	28%	2.24	735	**300.45**	2,445	19%	0.16	131839	**282.86**	466,094	46%	1.36	8138605	*Failure*	-
Iterative (DO + AutoBridge FP)	48%	1658	92.7	235.89	393	41%	10307	2549	278.84	9,141	62%	3554	598532	159.97	3,741,525
Iterative (DO + Incr FP) (**Ours**)	55%	2.39	**91.2**	269.95	**338**	41%	2.17	**1647**	274.47	**6,001**	63%	1.95	**128104**	**300.45**	**426,374**

Benchmarks	MM*2+2MM*2					CNN*3+COV*2					MTTKRP*2+COV*2				
	Resource	Runtime	Latency	Fmax	**Exe_time**	Resource	Runtime	Latency	Fmax	**Exe_time**	Resource	Runtime	Latency	Fmax	**Exe_time**
Original (no directive)	40%	-	259516	-	-	31%	-	18130	-	-	38%	-	8113234	-	-
Initial FP -> Iterative DO	59%	1.13	258842	274.10	944,335	39%	2.03	6716	**300.45**	22,354	42%	2.18	8113234	**300.45**	27,003,607
Iterative (DO + AutoBridge FP)	60%	32656	67652	*Failure*	-	62%	8301	1278	222.01	5,754	61%	12627	562017	**300.45**	1,870,585
Iterative (DO + Incr FP) (**Ours**)	58%	6.63	**66158**	**300.00**	**220,527**	63%	5.04	**1233**	**300.45**	**4,105**	64%	4.89	**126921**	**300.45**	**422,437**

[a] The maximum utilization ratio among BRAM, DSP, FF, LUT, and URAM. [b] DSE runtime in seconds. [c] Execution time of HLS designs in number of *thousand* clock cycles.
[d] Maximum achievable frequency in MHz. [e] FP: Floorplanning. [f] DO: Directive Optimization. [g] Overall execution time (cycle number/frequency) of HLS designs in microseconds (μs).

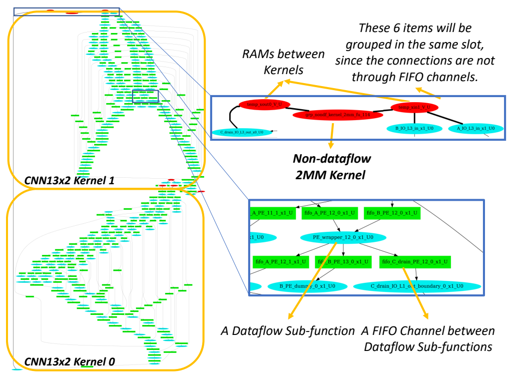

These 6 items will be grouped in the same slot, since the connections are not through FIFO channels.
RAMs between Kernels
Non-dataflow 2MM Kernel
CNN13x2 Kernel 1
CNN13x2 Kernel 0
A Dataflow Sub-function
A FIFO Channel between Dataflow Sub-functions

Figure 10: The Scale of the CNN*2+2MM*1 Benchmark.

design space of FADO is the Cartesian product of directive space and floorplan space. For directive search, the space ranges from millions to billions in our benchmarks, considering the parameters in Table 4. For floorplanning, it maps hundreds of functions to four slots, and the space size is four to the power of hundreds.

We use the Xilinx Vitis HLS 2020.2 for HLS synthesis and Vitis for implementation. We evaluate our framework on the Xilinx Alveo U250 FPGA, which contains eight slots defined by the 4 SLRs and an I/O bank in the middle. Note that the rightmost column of clock regions is occupied by Vitis platform IP. Hence the resource calculation excludes that column. We reduce the floorplanning of HLS designs to the lower half[2] (4 slots on SLR 0 and SLR 1) of the FPGA to enable exhaustive floorplan search in analyzing the optimality of our results. In our experiments, we find the resource constraint of 70% still leads to placement or routing failure sometimes. Hence, we tighten the limit to 65% for each slot during DSE.

6.2 Comparative Experiments

Table. 5 compares FADO with different top-level DSE algorithms and floorplanning algorithms. We report both the DSE runtime and the quality of each design implementation with their latency, maximum frequency, and overall execution time. Among these metrics, the overall execution time combines latency and timing quality, reflecting the ultimate design performance on FPGA.

In Table. 5, the "Initial FP -> Iterative DO" baseline performs the directive optimization using one-off initial floorplanning. It applies the min-cut MILP floorplanning from [7], and all HLS functions' positions are fixed during the iterative directive search. The limited optimization opportunities caused by the fixed initial floorplan lead to an under-utilization of resources. This seriously limits the latency optimization, resulting in the longest latency for all benchmarks. It fails in the implementation of *MTTKRP*2+HEAT*2* because two *HEAT* kernels are floorplanned on the same slot, but each has a large array using more than one column of BRAM or URAM, which triggers an exception during placement.

The "Iterative (DO + AutoBridge FP)" baseline runs the min-cut MILP floorplanning iteratively. Note that the heuristics of look-ahead and look-back are also applied in this algorithm for fairness when compared with FADO. This algorithm results in orders-of-magnitude longer runtime than FADO due to repetitively calling the MILP solver. Meanwhile, since AutoBridge [7] applies iterative bi-partitioning rather than one-off eight-way partitioning[3], it fails to reach some of the solutions. As reflected by the execution time, its design implementation quality is inferior to FADO in all six benchmarks. In summary, this method takes a significantly longer time while still resulting in a sub-optimal design.

As for our FADO, the online packing and offline re-packing strategies alternatively balance and compact the floorplan, contributing to full utilization of resources on multiple dies (the highest utilization ratio under resource constraint of 65% in five out of all six benchmarks). Accordingly, the high-quality floorplan provides strong support for exploring a larger design space during the directive search, thus our FADO achieves 33.12% smaller latency on average compared with the time-consuming "iterative (DO + AutoBridge FP)", and attains the lowest latency for all benchmarks over all baselines. The latency improvement varies because of the nature of benchmarks – it's more significant when FADO legalizes the floorplan for some bottleneck functions with a great latency-resource tradeoff, as the cases *MM*1+COV*2*, *MTTKRP*2+HEAT*2*, and *MTTKRP*2+COV*2* show. As for frequency, experiments show that when the utilization gets close to 65%, although the frequency could vary to some extent due to non-determinism in floorplanning and further implementation, our incremental solution still outperforms the baselines, with both a higher average Fmax of 290.96 MHz and lower variance. Moreover, since our incremental legalization leads to a minimum change of floorplan in each iteration of co-optimization, it's much more efficient than updating all functions' locations globally. This efficient legalization algorithm contributes to a speedup of 693X~4925X in the runtime of

[2]The lower half of Alveo U250 FPGA excluding the rightmost column of clock region contains 2016 BRAMs, 5184 DSPs, 1319040 FFs, 659520 LUTs, and 544 URAMs.

[3]Eight-way partitioning runs even more than 10x slower compared with bi-partitioning in directive-floorplan co-search experiments using benchmarks above.

Table 6: Stages of Floorplan-aware Directive Optimization

Benchmarks	CNN*2+2MM*1		MM*1+COV*2		MTTKRP*2+HEAT*2		MM*2+2MM*2		CNN*3+COV*2		MTTKRP*2+COV*2	
Stages	Resource[a]	Latency[b]	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency	Resource	Latency
Online	28.27%	735	40.66%	5167	63.15%	163241	40.28%	259516	31.79%	4718	62.95%	141260
Offline	40.12%	132	40.66%	5167	64.67%	153927	40.28%	259516	31.79%	4718	62.95%	141260
Look-Ahead	55.01%	91.4	40.66%	1651	63.26%	129184	55.25%	66184	31.79%	4718	64.49%	126921
Look-Back	54.56%	91.2	40.66%	1647	63.25%	128104	57.53%	66158	63.32%	1233	64.49%	126921

[a] The maximum utilization ratio among BRAM, DSP, FF, LUT, and URAM. [b] Execution time of HLS designs in number of *thousand* clock cycles.

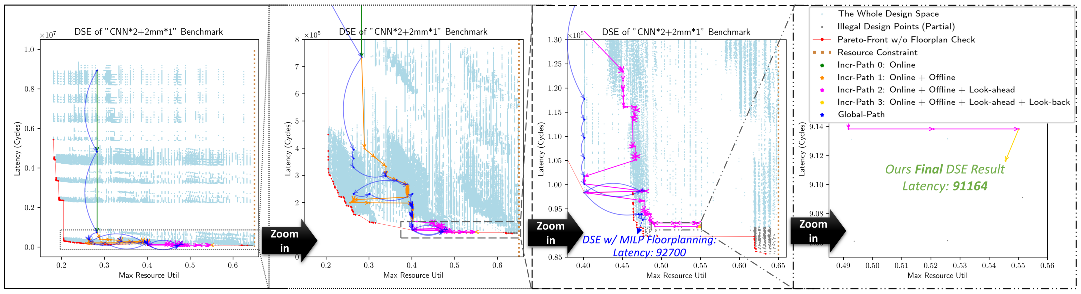


Figure 11: DSE Stages and Results on the CNN*2+2MM*1 Benchmark.

the entire co-optimization. Without any loss in timing quality, the design implementation quality reflected in overall execution time is 1.16X~8.78X better than the best baseline.

6.3 Analysis of DSE Stages

Fig. 11 shows the multiple stages of directive-floorplan co-search for *CNN*2+2MM*1* benchmark. The horizontal axis takes the maximum utilization among resources on the FPGA, and the vertical axis shows the latency in the number of clock cycles. The cyan points represent the whole directive design space without floorplan legality check, with red dots showing the Pareto-front. Our search starts from the highest latency point (28.27%, 8933000). In the first stage, the green arrows at the beginning are showing online floorplanning. It stops at (28.27%, 734592) because of a sharp resource increase of the large non-dataflow kernel. In the second stage, the offline re-packing clears out the dataflow sub-functions on the least-occupied slot, and continues until (40.12%, 131752), where the yellow arrows fall into a local maximum resource in the second sub-figure. It would fail to continue without looking ahead for points with less utilization of the current critical resource. As the third sub-figure shows, the pink arrows reach (55.01%, 91384) by looking ahead for three more design points after failure in previous stages. Finally, the DSE stops at (54.56%, 91164) after the final look-back. To compare, the DSE with global MILP floorplanning stops earlier at (47.59%, 92700). To show the optimality of our result, we check the floorplan legality for all design points with less latency than our result of 91164 — all the gray points have no legal floorplan when running global MILP floorplanning solely.

Table. 6 shows the DSE results of different optimization stages in FADO. Note that the four stages are running sequentially in each iteration, and the latency/resource in this table is not the result of each stage acting alone, except for "Online". For example, for stage "Look-Ahead", it includes the joint effort of (1) online packing, (2) offline re-packing followed by another round of online packing, and (3) look-ahead followed by online packing, as described in Alg. 1. It's possible that for some iterations, we only use (1), or (1)+(2), while using (1)+(2)+(3) in the worst cases. The QoR of each stage shown in Table 6 measures the legal design point with the smallest latency achieved before the first call to the next stage. For example, the results for "Look-Ahead" is the legal point with the smallest latency achieved before the first call to `look_back()`.

For benchmarks *CNN*2+2MM*1* and *MTTKRP*2+HEAT*2*, each stage is more effective than the previous ones to avoid local optima. However, the offline method fails to improve the results in *MM*1+COV*2*, *MM*2+2MM*2*, and *MTTKRP*2+COV*2*, compared with online stage. This happens when there are large design points with over-utilization. For example, the non-dataflow *COV* kernel consumes 30 DSPs when without any directive. However, when we unroll the loop containing the multiplication operation, the DSP increases to 1920, which is more than the total DSP available in any slot. Thus, offline stage fails to optimize the floorplan, and the bottleneck, DSP utilization always remains the same value during DSE in *MM*1+COV*2* and *MTTKRP*1+COV*2*. For *CNN*3+COV*2*, since *COV* kernel has a longer latency than *CNN*, major improvements are enabled by the look-back applied at the beginning of DSE.

6.4 Optimality Analysis

To analyze the optimality of our latency achieved, we check the floorplan legality for all design points with less latency than our final result. For those benchmarks with too many design points to check legality, we sample 2000 points from them and run MILP floorplanning respectively for each point. Our results show that there's no legal floorplan when a design point's latency is below our final result for all six benchmarks.

7 CONCLUSION

Our work produces FADO, an open-source framework that co-optimizes the directives and floorplan of HLS designs implemented on multi-die FPGAs. FADO combines a latency-bottleneck-guided directive optimization and an incremental floorplanning algorithm mixing various bin-packing heuristics. On the one hand, our well-customized incremental floorplanning achieves a speedup of 693X ~4925X over the global MILP floorplanning [7]. On the other hand, our co-optimization enables full utilization of resources on multiple dies and greatly benefits both the latency and timing. Among all six large-scale benchmarks mixing dataflow and non-dataflow kernels, FADO optimizes their execution time with a speedup of 1.16X~8.78X compared with the global floorplanning solution.

ACKNOWLEDGMENTS

This work is partially supported by the RGC GRF grant 16215319, and ACCESS — AI Chip Center for Emerging Smart Systems, Hong Kong SAR. We would like to thank all the anonymous reviewers for their valuable comments.

REFERENCES

[1] Md Mostofa Akbar, Eric G Manning, Gholamali C Shoja, and Shahadat Khan. 2001. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *International Conference on Computational Science*. Springer, 659–668.

[2] Tobias Alonso, Lucian Petrica, Mario Ruiz, Jakoba Petri-Koenig, Yaman Umuroglu, Ioannis Stamelos, Elias Koromilas, Michaela Blott, and Kees Vissers. 2021. Elastic-DF: Scaling performance of DNN inference in FPGA clouds through automatic partitioning. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 2 (2021), 1–34.

[3] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*. Springer, 213–222.

[4] Raghunandan Chaware, Kumar Nagarajan, and Suresh Ramalingam. 2012. Assembly and reliability challenges in 3D integration of 28nm FPGA die on a large high density 65nm passive interposer. In *2012 IEEE 62nd Electronic Components and Technology Conference*. IEEE, 279–283.

[5] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[6] Yangdong Steven Deng and Wojciech Maly. 2003. Physical design of the "2.5 D" stacked system. In *Proceedings 21st International Conference on Computer Design*. IEEE, 211–217.

[7] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. 2021. Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 81–92.

[8] Andre Hahn Pereira and Vaughn Betz. 2014. Cad and routing architecture for interposer-based multi-fpga systems. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 75–84.

[9] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1192–1195.

[10] Xilinx Inc. 2016. UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices. https://docs.xilinx.com/v/u/en-US/wp477-ultraram

[11] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Optimization-Directives

[12] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Applying-Optimization-Directives-to-Templates

[13] Xilinx Inc. 2020. Vitis High-Level Synthesis User Guide. https://docs.xilinx.com/r/2020.2-English/ug1399-vitis-hls/Automatic-Loop-Pipelining

[14] Xilinx Inc. 2022. Floorplanning With Stacked Silicon Interconnect (SSI) Devices. https://docs.xilinx.com/r/en-US/ug906-vivado-design-analysis/Floorplanning-With-Stacked-Silicon-Interconnect-SSI-Devices

[15] Richard M Karp. 1992. On-line algorithms versus off-line algorithms: How much is it worth to know the future?. In *Algorithms, Software, Architecture: Information Processing 92: Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain*, Vol. 1. 416.

[16] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. The multiple-choice knapsack problem. In *Knapsack Problems*. Springer, 317–347.

[17] Ariel Kulik and Hadas Shachnai. 2010. There is no EPTAS for two-dimensional knapsack. *Inform. Process. Lett.* 110, 16 (2010), 707–710.

[18] Tingyuan Liang, Jieru Zhao, Liang Feng, Sharad Sinha, and Wei Zhang. 2019. Hi-clockflow: Multi-clock dataflow automation and throughput optimization in high-level synthesis. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.

[19] Tomofumi Yuki Louis-Noel Pouchet. 2016. PolyBench/C. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[20] EG Co man Jr, MR Garey, and DS Johnson. 1996. Approximation algorithms for bin packing: A survey. *Approximation algorithms for NP-hard problems* (1996), 46–93.

[21] Fubing Mao, Wei Zhang, Bo Feng, Bingsheng He, and Yuchun Ma. 2016. Modular placement for interposer based multi-FPGA systems. In *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE, 93–98.

[22] Silvano Martello and Paolo Toth. 1990. Bin-packing problem. *Knapsack problems: Algorithms and computer implementations* (1990), 221–245.

[23] Silvano Martello and Paolo Toth. 1990. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.

[24] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. 2016. Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 918–923.

[25] Ehsan Nasiri, Javeed Shaikh, Andre Hahn Pereira, and Vaughn Betz. 2015. Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 5 (2015), 1821–1834.

[26] Boaz Patt-Shamir and Dror Rawitz. 2010. Vector bin packing with multiple-choice. In *Scandinavian Workshop on Algorithm Theory*. Springer, 248–259.

[27] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. 2015. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 157–162.

[28] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. 2017. COSMOS: Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–22.

[29] Chirag Ravishankar, Dinesh Gaitonde, and Trevor Bauer. 2018. Placement strategies for 2.5 D FPGA fabric architectures. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 16–164.

[30] Kirk Saban. 2011. Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency. *Xilinx, White Paper* 1, 1 (2011), 1–10.

[31] Benjamin Carrion Schafer. 2017. Parallel High-Level Synthesis Design Space Exploration for Behavioral IPs of Exact Latencies. 22, 4, Article 65 (may 2017), 20 pages. https://doi.org/10.1145/3041219

[32] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. 2009. Adaptive Simulated Annealer for high level synthesis design space exploration. In *2009 International Symposium on VLSI Design, Automation and Test*. 106–109. https://doi.org/10.1109/VDAT.2009.5158106

[33] Prabhakant Sinha and Andris A Zoltners. 1979. The multiple-choice knapsack problem. *Operations Research* 27, 3 (1979), 503–515.

[34] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. 2021. Enabling Automated FPGA Accelerator Optimization Using Graph Neural Networks. *CoRR* abs/2111.08848 (2021). arXiv:2111.08848 https://arxiv.org/abs/2111.08848

[35] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.

[36] Jie Wang, Licheng Guo, and Jason Cong. 2021. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 93–104.

[37] Zi Wang, Jianqi Chen, and Benjamin Carrion Schafer. 2020. Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 145–150.

[38] Nan Wu, Yuan Xie, and Cong Hao. 2021. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*. 39–44.

[39] Xilinx. 2022. Vitis Accelerated Libraries. https://github.com/Xilinx/Vitis_Libraries/

[40] Sotirios Xydis, Christos Skouroumounis, Kiamal Pekmestzi, Dimitrios Soudris, and George Economakos. 2010. Efficient high level synthesis exploration methodology combining exhaustive and gradient-based pruned searching. In *2010 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 104–109.

[41] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2021. ScaleHLS: Scalable High-Level Synthesis through MLIR. *arXiv preprint arXiv:2107.11673* (2021).

[42] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. 2018. S2FA: An accelerator automation framework for heterogeneous computing in datacenters. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[43] Georgios Zacharopoulos, Andrea Barbon, Giovanni Ansaloni, and Laura Pozzi. 2018. Machine learning approach for loop unrolling factor prediction in high level synthesis. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 91–97.

[44] Jiaxi Zhang, Wentai Zhang, Guojie Luo, Xuechao Wei, Yun Liang, and Jason Cong. 2019. Frequency improvement of systolic array-based CNNs on FPGAs. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

[45] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 430–437.

[46] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, and Zhiru Zhang. 2018. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)* (Feb 2018).