AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs

Licheng Guo[1], Yuze Chi[1], Jie Wang[1], Jason Lau[1], Weikang Qiao[1], Ecenur Ustun[2], Zhiru Zhang[2], Jason Cong[1]
[1]University of California, Los Angeles [2]Cornell University
{lcguo,yuzechi,jaywang,lau,wkqiao,cong}@cs.ucla.edu,{eu49,zhiruz}@cornell.edu

ABSTRACT

Despite an increasing adoption of high-level synthesis (HLS) for its design productivity advantages, there remains a significant gap in the achievable clock frequency between an HLS-generated design and a handcrafted RTL one. A key factor that limits the timing quality of the HLS outputs is the difficulty in accurately estimating the interconnect delay at the HLS level. Unfortunately, this problem becomes even worse when large HLS designs are implemented on the latest multi-die FPGAs, where die-crossing interconnects incur a high delay penalty.

To tackle this challenge, we propose AutoBridge, an automated framework that couples a coarse-grained floorplanning step with pipelining during HLS compilation. First, our approach provides HLS with a view on the global physical layout of the design, allowing HLS to more easily identify and pipeline the long wires, especially those crossing the die boundaries. Second, by exploiting the flexibility of HLS pipelining, the floorplanner is able to distribute the design logic across multiple dies on the FPGA device without degrading clock frequency. This prevents the placer from aggressively packing the logic on a single die which often results in local routing congestion that eventually degrades timing. Since pipelining may introduce additional latency, we further present analysis and algorithms to ensure the added latency will not compromise the overall throughput.

AutoBridge can be integrated into the existing CAD toolflow for Xilinx FPGAs. In our experiments with a total of 43 design configurations, we improve the average frequency from 147 MHz to 297 MHz (a 102% improvement) with no loss of throughput and a negligible change in resource utilization. Notably, in 16 experiments we make the originally unroutable designs achieve 274 MHz on average. The tool is available at *https://github.com/Licheng-Guo/AutoBridge*.

KEYWORDS

High-Level Synthesis; Multi-Die FPGA; Frequency; Timing Closure; Floorplan; Dataflow; Pipeline; Latency Insensitive Design.

ACM Reference Format:

Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, Jason Cong, 2021. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28–March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages.

FPGA '21, February 28-March 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439289

1 INTRODUCTION

High-level synthesis (HLS) tools simplify the FPGA design processes by allowing users to express untimed designs in high-level languages such as C/C++ or OpenCL without concern for cycle-accurate details at the register-transfer level (RTL). However, while the productivity is significantly improved, there still exists a considerable gap between the quality of result (QoR) of an HLS-generated design and what is achievable by an RTL expert.

One major cause that leads to the unsatisfactory frequency is that HLS cannot easily predict the physical layout of the design after placement and routing. Current HLS tools typically rely on pre-characterized operation delays and a very crude interconnect delay model to insert clock boundaries (i.e., registers) into an untimed design to generate a timed RTL implementation [1, 2, 3]. Afterwards, optimizations in RTL and physical synthesis such as retiming are expected to fix the potential critical paths due to inadequate pipelining. However, while retiming can redistribute the registers along a path, the total number of registers along each path or cycle must remain a constant [4], significantly limiting the scope of improvement. Hence, as the HLS designs get larger, the timing quality of the synthesized RTLs usually further degrade.

This timing issue is worsened as modern FPGA architectures become increasingly heterogeneous [5]. The latest FPGAs integrate multiple dies using silicon interposers to pack more logic on a single device; however, the interconnects that go across the die boundaries will carry a non-trivial delay penalty. In addition, specialized IP blocks such as PCIe and DDR controllers are embedded amongst the programmable logic. These IP blocks usually have fixed locations near dedicated I/O banks and will consume a large amount of programmable resources nearby. As a result, these dedicated IPs often detour the signals close-by towards more expensive and/or longer routing paths. Further, modules interacting with such fixed-location IPs are also more constrained in their layout. This, in turn, results in long-distance communication to other modules. Together these factors tend to further lower the final clock frequency.

There are a number of prior attempts that couple the physical design process with HLS compilation [1, 6, 7, 8, 9]. Zheng *et. al.* [1] propose to iteratively run placement and routing to obtain accurate delay statistics of each wire and operator. Based on the post-route information, HLS re-runs the scheduling step for a better pipelining; Cong *et. al.* [6] is another representative work that presents placement-driven scheduling and binding for multi-cycle communications in an island-style architecture similar to FPGAs.

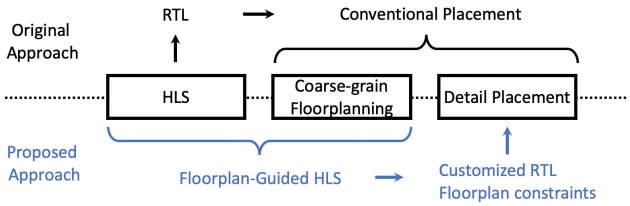

Figure 1: Core Idea of the Proposed Methodology.

The previous approaches share the common aspect of focusing on the fine-grained interaction between HLS and physical design, where individual operators and the associated wires and registers are all involved during the delay prediction and iterative HLS-layout co-optimization. While such a fine-grained method can be effective on relatively small HLS designs and FPGA devices, it is too expensive (if not infeasible) for today's large designs targeting multi-die FPGAs, where each implementation iteration from HLS to bitstream may take days to complete.

In this paper we propose *AutoBridge*, a *coarse-grained* floorplan-guided pipelining approach that addresses the timing issue of large HLS designs in a highly effective and scalable manner. Instead of coupling the entire physical design process with HLS, we guide HLS with a coarse-grained floorplanning step, as shown in Figure 1. Our coarse-grained floorplanning involves dividing the FPGA device into a grid of regions and assigning each HLS function to one region during HLS compilation. For all the inter-region connections we further pipeline them to facilitate timing closure while we leave the intra-region optimization to the default HLS tool.

Our methodology has two major benefits. First, the early floorplanning step provides HLS a view of the global physical layout which helps HLS more accurately identify and pipeline the long wires, especially those crossing the die boundaries. Compared to retiming [10], HLS-level pipelining creates more optimization opportunities for the downstream synthesis and physical design steps, thus potentially leading to higher performance. Second, the pipelining-aware floorplanning can reduce local routing congestion by guiding the subsequent placement steps to better distribute logic across multiple dies, instead of attempting to pack the logic into a single die as much as possible.

While AutoBridge can improve the *frequency* with additional interconnect pipelining, we also need to ensure the added latency does not negatively impact the overall throughput of the design. To this end, we present analysis and latency balancing algorithms to guarantee the throughput of the resulting design is not negatively impacted.

Our specific contributions are as follows:

- To the best of our knowledge, we are the first to tackle the challenge of high-frequency HLS design on multi-die FPGAs by coupling floorplanning and pipelining.
- We design a coarse-grained floorplan scheme tailored for HLS which can distribute the design logic across multiple dies on an FPGA to effectively reduce local congestion and facilitate HLS to adequately pipeline global interconnects.
- We analyze how the additional latency may affect the throughput of the design, and propose algorithms to offset the potential negative influence of the added latency.

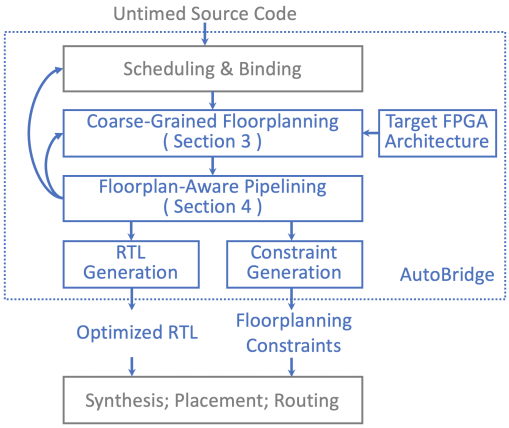

Figure 2: Overview of the AutoBridge Framework. Grey boxes represent the original software flow and blue boxes represent components of AutoBridge.

- Our framework, AutoBridge, interfaces with the commercial FPGA design toolflow, with a compile time overhead in the order of seconds. It improves the average frequency of 43 designs from 147 MHz to 297 MHz with and a negligible area overhead.

Figure 2 shows the overall flow of our proposed methodology. The rest of the paper is organized as follows: Section 2 introduces background information on modern FPGA architectures and shows motivating examples; Section 3 details our coarse-grained floorplan scheme inside the HLS flow; Section 4 describes our floorplan-aware pipelining methods; Section 5 presents experimental results; Section 6 provides related work, followed by conclusion and acknowledgements.

2 BACKGROUND AND MOTIVATING EXAMPLES

2.1 Multi-Die FPGA Architectures

Figure 3 shows three representative multi-die FPGA architectures, each of which is described in more details as follows.

- The Xilinx Alveo U250 FPGA is one of the largest FPGAs with four dies. All the I/O banks are located in the middle column and the four DDR controller IPs are positioned vertically in a tall-and-slim rectangle in the middle. On the right lies the Vitis platform region [11], which incorporates the DMA IP, the PCIe IP, etc, and serves to communicate with the host CPU.
- The Xilinx Alveo U280 FPGA is integrated with the latest High-Bandwidth Memory (HBM) [12, 13, 14], which exposes 32 independent memory ports at the bottom of the chip. I/O banks are located in the middle columns. Meanwhile, there is a gap region void of programmable logic in the middle.
- The Intel Stratix 10 FPGA [15] also sets the DDR controller and I/O banks in the middle of the programmable logic. The embedded multi-die interconnect bridges and the PCIe blocks are distributed at the two sides of the chip, allowing multiple FPGA chips to be integrated together. Although this paper uses the Xilinx FPGAs to demonstrate the idea, our methodology is also applicable to Intel FPGAs and other architectures.

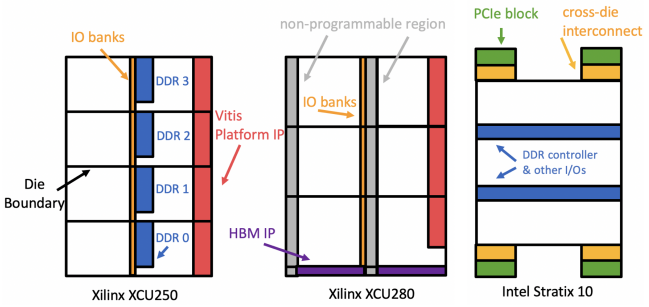

Figure 3: Block diagrams of three representative FPGA architectures: the Xilinx Alveo U250, U280 (based on the Xilinx UltraScale+ architecture) and the Intel Stratix 10.

Compared to previous generations, the latest multi-die FPGA architectures are divided into disjoint regions, where the region-crossing naturally incurs additional signal delay. In addition, the large pre-located IPs consume significant programmable resources near their fixed locations that may also cause local routing congestion. These characteristics can hamper the existing HLS flows from achieving a high frequency.

2.2 Motivating Examples

We show two examples to motivate our floorplan-guided HLS approach. First, Figure 4 shows a CNN accelerator implemented on the Xilinx U250 FPGA. It interacts with three DDR controllers, as marked in grey, pink, and yellow blocks in the figure. In the original implementation result, the whole design is packed close together within die 2 and die 3. To demonstrate our proposed idea, we first manually floorplan the design to distribute the logic in four dies and to avoid overlapping the user logic with DDR controllers. Additionally, we pipeline the FIFO channels connecting modules in different dies as demonstrated in the figure. The manual approach improves the final frequency by 53%, from 216 MHz to 329 MHz.

Second, Figure 5 shows a stencil computation design on the Xilinx U280 FPGA. It consists of four identical kernels in linear topology with each color representing a kernel. In the original implementation, the tool's choice of die-crossing wires are sub-optimal and one kernel may be divided among multiple regions. Instead in our approach, we pre-determine all the die-crossing wires during HLS compilation and pipeline them, so the die boundaries will not cause any problems for the placement and routing tool. For this example, we achieve 297 MHz while the design is originally *unroutable*.

3 COUPLING HLS WITH COARSE-GRAINED FLOORPLANNING

In this section, we present our coarse-grained floorplanning scheme that can be integrated with HLS. We assume that HLS preserves the hierarchy of the source code, and each ***function*** in the HLS source code will be compiled into an RTL ***module***.

Note that the focus of this work is not about improving floorplanning algorithms; instead, we intend to properly use coarse-grained floorplan information to guide HLS and placement.

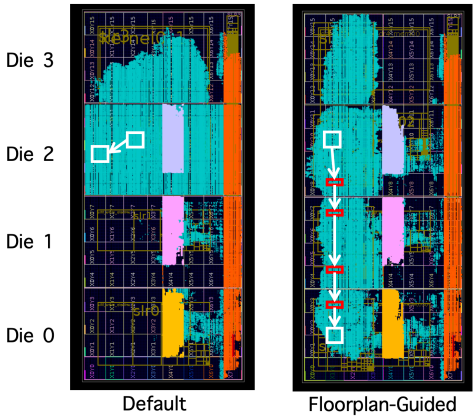

Figure 4: Implementation results of a CNN accelerator on the Xilinx U250 FPGA. Spreading the design across the device helps reduce local congestion, while the die-crossing wires are additionally pipelined.

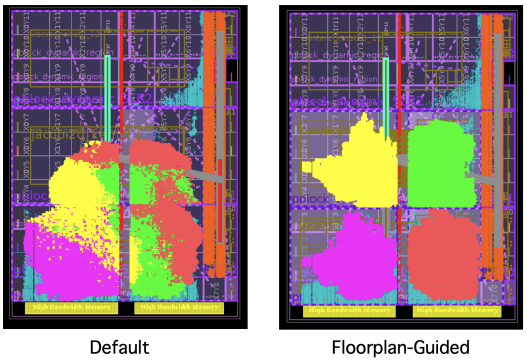

Figure 5: Implementation results of a stencil computing design on U280. Floorplanning during HLS compilation significantly benefits the physical design tools.

3.1 Coarse-Grained Floorplanning Scheme

Instead of finding a dedicated region with a detailed aspect ratio for each module, we choose to view the FPGA device as a ***grid*** that is formed by the die boundaries and the large IP blocks. These physical barriers split the programmable fabric apart into a series of disjoint ***slots*** in the grid where each slot represents a sub-region of the device isolated by die boundaries and IP blocks. Using our coarse-grained floorplanning, we will assign each function of the HLS design to one of these slots.

For example, for the Xilinx Alveo U250 FPGA, the array of DDR controllers forms a vertical split in the middle column; and there are three horizontal die boundaries. Thus the device can be viewed as a grid of 8 slots in 2 columns and 4 rows. Similarly, the U280 FPGA can be viewed as a grid of 6 slots in 2 columns and 3 rows.

In this scheme, each slot contains about 700 BRAM_18Ks, 1500 DSPs, 400K Flip-Flops and 200K LUTs. Meanwhile, to reduce the resource contention in each slot, we set a maximum utilization ratio for each slot to guarantee enough blank space. Experiments show that such slot sizes are suitable, and HLS has a good handle of the timing quality of the local logic within each slot, as in Section 5.

3.2 Problem Formulation

We first assume the HLS design adopts a dataflow programming model, where each function corresponds to one dataflow process, and each function will be compiled into an RTL module. Functions communicate with each other through FIFO channels.

Given: (1) a graph $G(V, E)$ representing the HLS design where V represents the set of functions[1] of the dataflow design and E represents the set of FIFO channels between vertices; (2) the number of rows R and the number of columns C of the grid representation of the target device; (3) maximum resource utilization ratios for each slot; (4) location constraints such that certain IO modules must be placed nearby certain IP blocks. In addition, we may have constraints that certain vertices must be assigned to the same slot. This is for throughput concerns and will be explained in Section 4.

Goal: Assign each $v \in V$ to one of the slots such that (1) the resource utilization ratio[2] of each slot is below the given limit; (2) the cost function is minimized. We choose the total number of slot-crossings as the cost instead of the total estimated wire lengths. Specifically, the cost function is defined as

$$\sum_{e_{ij} \in E} e_{ij}.width \times (|v_i.row - v_j.row| + |v_i.col - v_j.col|) \quad (1)$$

where $e_{ij}.width$ is the bitwidth of the FIFO channel connecting v_i and v_j and module v is assigned to the $v.col$-th column and the $v.row$-th row. The physical meaning of the cost function is the sum of the number of slot boundaries that every wire crosses.

3.3 Solution

Our problem is small in size as HLS-level FPGA designs seldom have more than a few hundred functions. We adopt the main idea of top-down partitioning-based placement algorithms [16, 17, 18] to solve our problem. Meanwhile, due to the relatively small problem size, we plan to pursue an exact solution in each partitioning.

Figure 6 demonstrates the floorplanning of an example design through three iterations of partitioning. The top-down partitioning-based approach starts with the initial state where all modules are assigned to the same slot, iteratively partitions the current slots in half into two ***child slots*** and then assigns the modules into the child slots. Each partitioning involves splitting all of the current slots in half either horizontally or vertically.

Since the problem size is relatively small, we formulate the partitioning process of each iteration using integer linear programming (ILP). In every partitioning iteration, all current slots need to be divided in half. Since some of the modules in a slot may be tightly connected to modules outside of the slot, ignoring such connections can adversely affect the quality of the assignment. Therefore our ILP formulation considers the partitioning of all slots together for an exact solution which is possible due to the small problem size. Experiments in Section 5 show that our ILP formulation is solvable within a few seconds or minutes for designs of hundreds of modules.

Performing an N-way partitioning is another potential method. However, compared to our iterative 2-way partitioning, experiments show that it is much slower than iterative 2-way partitioning.

[1]Inlined functions will be merged accordingly in the C++ front-end processing.
[2]Based on the estimation of resource utilization by HLS.

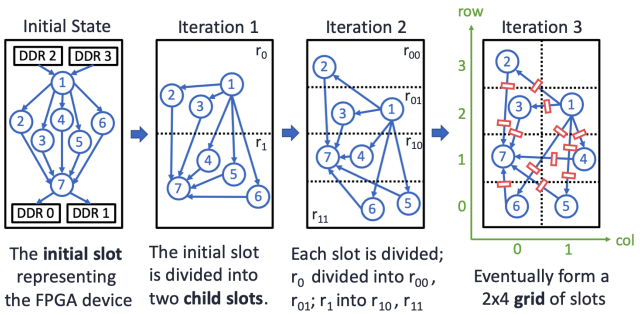

Figure 6: Generating the floorplan for a target 2×4 grid. Based on the floorplan, all the cross-slot connections will be accordingly pipelined (marked in red) for high frequency.

ILP Formulation of One Partitioning Iteration.

The formulation declares a binary decision variable v_d for each v to denote whether v is assigned to the left or the right child slot during a vertical partitioning (or to the upper or the lower child slot for a horizontal one). Let R denote the set of all current slots. For each slot $r \in R$ to be divided, we use r_v to denote the set of all vertices that r is currently accommodating. To ensure that the child slots have enough resources for all modules assigned to them, the ILP formulation imposes the resource constraint for each child slot r_{child} and for each type of on-chip resource.

$$\sum_{v \in r_v} v_d \times v_{area} < (r_{child})_{area}$$

where v_{area} is the resource requirement of v and $(r_{sub})_{area}$ represents the available resources in the child slot divided from r.

To express the cost function that is based on the coordinates of each module, we first need to express the new coordinates $(v.row, v.col)$ of v based on the previous coordinates $((v.row)_{prev}, (v.col)_{prev})$ and the decision variable v_d. For a vertical partitioning, the new coordinates of v will be

$$v.col = (v.col)_{prev} \times 2 + v_d$$
$$v.row = (v.row)_{prev}$$

And for a horizontal partitioning, the new coordinates will be

$$v.row = (v.row)_{prev} \times 2 + v_d$$
$$v.col = (v.col)_{prev}$$

Finally, the objective is to minimize the total slot-crossing shown in Formula (1) for each partitioning iteration.

For the example in Figure 6, Table 1 shows the *row* and *col* indices of selected vertices in each partitioning iteration.

Table 1: Coordinates of selected vertices in Figure 6

	v_2	v_1	v_4	v_5
Init	row = 0; col = 0			
iter-1	v_d = 1; row = 0 × 2 + 1 = 1		v_d = 0; row = 0 × 2 + 0 = 0	
iter-2	v_d = 1; row = 1 × 2 + 1	v_d = 0; row = 1 × 2 + 0	v_d = 1; row = 0 × 2 + 1	v_d = 0; row = 0 × 2 + 0
iter-3	v_d = 0; col = 0 × 2 + 0	v_d = 1; col = 0 × 2 + 1		

4 FLOORPLAN-AWARE PIPELINING

Based on the generated floorplan, we aim to *pipeline every cross-slot connection* to facilitate timing closure.

Although HLS has the flexibility to pipeline them to increase the final ***frequency***, the additional latency could potentially lead to large increase of the ***execution cycles***, which we need to avoid. This section presents our methods to pipeline slot-crossing connections without hurting the overall throughput of the design.

We will first focus on pipelining the dataflow designs, then extend the method to other types of HLS design. In Section 4.1 we introduce our approach of pipelining with latency balancing; and Section 4.2 presents the detailed algorithm. In Sections 4.3 we present how to utilize the internal computation pattern to construct loop-level dataflow graphs that allow more pipelining opportunities. In Section 4.4 we discuss pipelining other types of HLS designs.

4.1 Pipelining Followed by Latency Balancing for Dataflow Designs

In our problem, an HLS dataflow design consists of a set of concurrently executed functions communicating through FIFO channels, where each function will be compiled into an RTL module controlled by a finite-state machine (FSM) [19]. The rich expressiveness of FSM makes it difficult to statically determine how the additional latency will affect the total execution cycles. Note that our problem is different from other simplified dataflow models such as the Synchronous Data Flow (SDF) [20] and the Latency Insensitive Theory (LIT) [21], where the firing rate of each vertex is fixed. Unlike SDF and LIT, in our problem each vertex is an FSM and the firing rate is not fixed and can have complex pattern.

Therefore, we adopt a conservative approach, where we first pipeline all edges that cross slot boundaries, then balance the latency of parallel paths based on the *cut-set pipelining* [22]. A cut-set is a set of edges that can be removed from the graph to create two disconnected sub-graphs; and if all edges in a cut-set are of the same direction, we could add an equal amount of latency to each edge and the throughput of the design will be unaffected. Figure 7 (a) illustrates the idea. If we need to add one unit of latency to e_{13} (marked in red) due to the floorplan results, we need to find a cut-set that includes e_{13} and *balance* the latency of all other edges in this cut-set (marked in blue).

Since we can choose different cut-set to balance the same edge, we need to minimize the area overhead. For example, for e_{13}, balancing the `cut-set 2` in Figure 7 (b) costs smaller area overhead compared to `cut-set 1` in Figure 7 (a), as the width of e_{47} is smaller than that of e_{14}. Meanwhile, it is possible that multiple edges can be included in the same cut-set. For example, the edges e_{27} and e_{37} are both included in the `cut-set 3`, so we only need to balance the other edges in `cut-set 3` once.

Cut-set pipelining is equivalent to balancing the total added latency of every pair of ***reconvergent paths*** [22]. A path is defined as one or multiple concatenated edges of the same direction; two paths are reconvergent if they have the same source vertex and destination vertex. When there are multiple edges with additional latency from the floorplanning step, we need to find a global optimal solution that ensures all reconvergent paths have a balanced latency, and the area overhead is minimized.

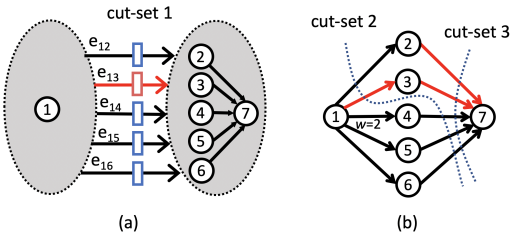

Figure 7: Assume that the edges e_{13}, e_{37} and e_{27} are pipelined according to some floorplan, and each of then carries 1 unit of inserted latency. Also assume that the bitwidth of e_{14} is 2 and all other edges are 1. In the latency balancing step, the optimal solution is adding 2 units of latency to each of e_{47}, e_{57}, e_{67} and 1 unit of latency to e_{12}. Note that edge e_{27} and e_{37} can exist in the same cut-set.

4.2 Latency Balancing Algorithm

Problem Formulation.

Given: A graph $G(V, E)$ representing a dataflow design that has already been floorplanned and pipelined. Each vertex $v \in V$ represents a function in the dataflow design and each edge $e \in E$ represents the FIFO channel between functions. Each edge $e \in E$ is associated with $e.width$ representing the bitwidth of the edge. For each edge e, the constant $e.lat$ represents the additional latency inserted to e in the previous pipelining step. We use the integer variable $e.balance$ to denote the number of latency added to e in the current latency balancing step.

Goal: (1) For each edge $e \in E$, compute $e.balance$ such that for any pair of reconvergent paths $\{p_1, p_2\}$, the total latency on each path is the same:

$$\sum_{e \in p_1} (e.lat + e.balance) = \sum_{e \in p_2} (e.lat + e.balance)$$

and (2) minimize the total area overhead, which is defined as:

$$\sum_{e \in E} e.balance \times e.width$$

Note that this problem is different from the min-cut problem [23] for DAG. One naïve solution is to find a min-cut for every pipelined edge, and increase the latency of the other edges in the cut accordingly. However, this simple method is suboptimal. For example in Figure 7, since edge e_{27} and e_{37} can be in the same cut-set, we only need to add one unit of latency to the other edges in the cut-set (e.g., e_{47}, e_{57} and e_{67}) so that all paths are balanced.

Solution.

We formulate the problem in a restricted form of ILP that can be solved in polynomial time. For each vertex v_i, we associate it with an integer variable S_i that denotes the maximum latency from pipelining between v_i and the sink vertex of the graph. In other words, given two vertices v_x and v_y, $(S_x - S_y)$ represents the maximum latency among all paths between the two vertices. Note that we only consider the latency on edges due to pipelining.

For each edge e_{ij}, we have

$$S_i \geq S_j + e_{ij}.lat$$

According to our definition, the additional balancing latency added to edge e_{ij} in this step can be expressed as

$$e_{ij}.balance = (S_i - S_j - e_{ij}.lat)$$

since we want every path from v_i to v_j have the same latency.

The optimization goal is to minimize the total area overhead, i.e. the weighted sum of the additional depth on each edge:

$$minimize \sum_{e_{ij} \in E} e_{ij}.balance \times e_{ij}.width$$

For example, assume that there are two paths from v_1 to v_2 where path p_1 has 3 units of latency from pipelining while p_2 has 1 unit. Thus from our formulation, we will select the edge(s) on p_2 and add 2 additional units of latency to balance the total latency of p_1 and p_2 so that the area overhead is minimized.

Our formulation is essentially a system of differential constraints (SDC), in which all constraints are in the form of $x_i - x_j \leq b_{ij}$, where b_{ij} is a constant and x_i, x_j are variables. Because of this restrictive form of constraints, we can solve SDC as a linear programming problem while the solutions are guaranteed to be integers. As a result, it can be solved in polynomial time [4, 24].

If the SDC formulation does not have a solution, there must be a dependency cycle in the dataflow graph [24]. This means that at least one of the edges in the dependency cycle are pipelined based on the floorplan. In this situation, we will feedback to the floorplanner to constrain those vertices into the same region and then re-generate a new floorplan.

4.3 Loop-Level Latency Balancing

In the previous subsection, we treat each function as a vertex in the dataflow graph and perform latency balancing. The limitation is that when there are dependency cycles in the graph, we must choose a conservative course and refrain from adding additional pipelining to the involved channels. As a result, the functions in a cycle must be floorplanned in the same slot. However, we could potentially resolve the dependency cycles if we treat each loop within a function as a vertex and construct a loop-level dataflow graph.

Figure 8 shows a motivating example. In this design, there are two functions `A` and `B`. We assume that the HLS scheduler has scheduled the loops to execute sequentially. If we view the topology of the design at the granularity of functions, there is a dependency cycle from `A` to `B` then back to `A`. To ensure that it works properly, we cannot add latency to either edge. However, if we look into the function and treat each loop as a vertex, we will find that the cycle is resolved and the edges now form a pair of reconvergent paths, which we can handle as presented in the previous subsection. To achieve such conversion, we make the former loop send a `start` signal after finishing to trigger the latter loop through a 1-bit-wide FIFO.

Note that the goal of loop-level analysis is not splitting the module, but determining whether it is safe to place `A` and `B` into different slots and pipeline the connections between them. We still consider floorplanning each function as a unit, not each loop. The loop-level dataflow graph allows more opportunities to produce a valid floorplan as many dependency cycles at function level can be resolved, thus allowing more flexibility for the floorplanning step.

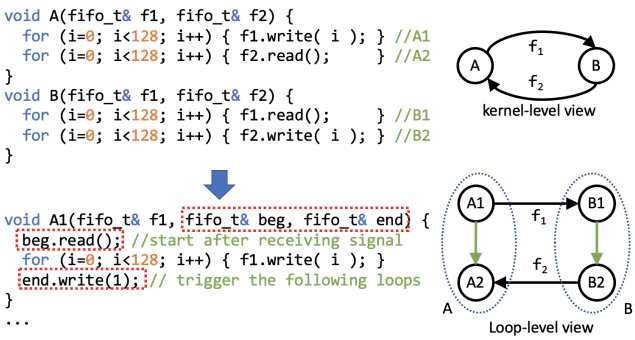

Figure 8: Motivating example for loop-level latency balancing. Assume the loops within a function are scheduled to execute sequentially. Black arrows represent the original data FIFOs in the dataflow design; green arrows represent our conceptual 1-bit-wide FIFO used to pass control signals. `A1` refers to the first loop in the function `A`. `f1` and `f2` represent FIFO connections between the three functions.

4.4 Extension to Non-Dataflow Designs

In the previous subsections, we focus on pipelining and latency balancing for dataflow designs as they can be easily pipelined. However, our methodology applies to other types of HLS designs as well. Since most interface protocols of HLS-generated modules have pre-determined operation latency, we can accurately predict at compile time whether the additional latency on certain interface will cause throughput degradation, in which case we will adjust the constraints for the floorplanning step.

When a C++ function is compiled into an RTL module, the arguments to the function become ports of the module with IO protocols according to the type of C++ arguments. Here we discuss how to add latency to interfaces with Vivado HLS [25] designs, but the concept and implementation is similar in other HLS compilers. Besides the FIFO interface, there are four major types of ports on RTL modules generated by HLS:

- `Control signals`. These include `start`, `ready` and `done` which indicate when the module starts executing and whether it has finished. They can be directly pipelined without influencing functionality. We require that the function is not invoked inside a loop to prevent the added latency from increasing the initiation interval of the loop.
- `Scalar` or `input pointer`. By default the pass-by-value input arguments and pointers are implemented as simple input wire ports. They can be directly pipelined, and the `start` should be pipelined accordingly.
- `Output pointers`. These are implemented with an associated output valid signal to indicate when the output data is valid. We can directly pipeline the output signals along with the valid signals.
- `Array` arguments. The compiler will compile them into a standard block RAM interface with data, address, chip-enable, and write-enable ports. For such interface, the configuration option specifies the read or write latency of the RAM resource driving the interface, which is known at compile time. Adding pipelining to all signals of the RAM interface will change the latency of

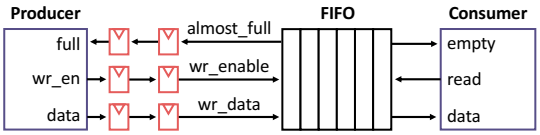

Figure 9: Pipelining FIFO interfaces using almost-full FIFOs.

RAM access operations, thus we require that the array should only be accessed inside a pipelined loop, where increasing the latency of the RAM operation will not increase the initiation interval of the pipeline.

In addition, we must re-run HLS synthesis for each function after annotating the new interface latency in the source code. In comparison, this is not needed for dataflow designs with latency-insensitive interfaces.

5 EXPERIMENTS

5.1 Implementation Details

We implement our proposed methods in Python interfaced with the CAD flow for Xilinx FPGAs, including Vivado HLS, Vivado and Vitis (2019.2). We parse the scheduling and binding reports of dataflow HLS designs to create the graph representation of the design and obtain the resource utilization of each RTL module. We use the Python MIP package [26] coupled with Gurobi [27] to solve the various ILP problems introduced in previous sections. We generate TCL constraint files to be used by Vivado to enforce our high-level floorplanning scheme. Our RTL generator parses the RTL from Vivado HLS using PyVerilog [28], then traverses the AST to add the additional pipelining and regenerate the optimized RTL.

We mainly implement the AutoBridge prototype for Vivado HLS dataflow designs, where the top function instantiates all the dataflow processes and the FIFO connections. In addition, we support the TAPA compiler [29], which serves as a front-end to the existing HLS tools to enable more expressibility over task-level parallel programs. We also include tools to process non-dataflow designs and some manual help is necessary due to the limited access to the internals of the HLS compiler. A certain coding style is expected and we provide examples in our open-sourced repository.

Figure 9 shows how we add pipelining to a FIFO-based connection. We adopt FIFOs that assert their `full` pin before the storage actually runs out, so that we could directly register the interface signals without affecting the functionality.

Meanwhile, we turn off the hierarchy rebuild process during RTL synthesis [30] to prevent the RTL synthesis tool from introducing additional wire connections between RTL modules. The hierarchy rebuild step first flattens the hierarchy of the RTL design then tries to rebuild the hierarchy. As a result, hierarchy rebuild may create unpredictable new connections between modules. As a result, if two modules are floorplanned far apart, these additional wires introduced during RTL synthesis will be under-pipelined as they are unseen during HLS compilation. Note that disabling this feature may lead to slight differences in the final resource utilization.

We test out designs on the Xilinx Alveo U250 FPGA[3] with 4 DRAMs and the Xilinx Alveo U280 FPGA[4] with High-Bandwidth Memory (HBM). As the DDR controllers are distributed in the middle vertical column while the HBM controller lies at the bottom row, these two FPGA architectures present different challenges to the CAD tools. Thus it is worthwhile to test them separately.

To run our framework, users first specify how they want to divide the device. By default, we divide the U250 FPGA into a 2-column × 4-row grid and the U280 FPGA into a 2-column × 3-row grid, matching the block diagram of these two architectures shown in Figure 3. To control the floorplanning, users can specify the maximum resource utilization ratio of each slot. The resource utilization is based on the estimation by HLS. Users can also specify how many levels of pipelining to add based on the number of boundary crossings. By default, for each boundary crossing we add 2 levels of pipelining to the connection. The processed design is integrated with the Xilinx Vitis (2019.2) infrastructure to communicate with the host.

[3]The U250 FPGA contains 5376 BRAM18K, 12288 DSP48E, 3456K FF and 1728K LUT
[4]The U280 FPGA contains 4032 BRAM18K, 9024 DSP48E, 2607K FF and 434K LUT

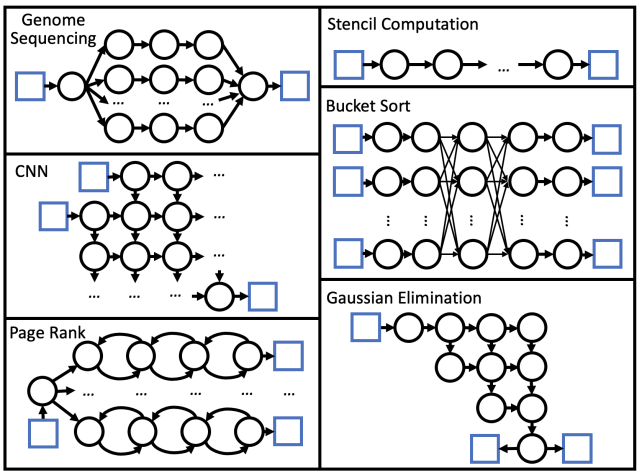

Figure 10: Topologies of the benchmarks. Blue rectangles represent external memory ports and black circles represent computation kernels of the design. In the genome sequencing design, the arrows represent BRAM channels; in other designs, the arrows represent FIFO channels.

5.2 Benchmarks

We use six representative benchmark designs with different topologies and change the parameter of the benchmarks to generate a set of designs with varying sizes on both the U250 and the U280 board. The six designs are all large-scale designs implemented and optimized by HLS experts. Figure 10 shows the topology of the benchmarks. Note that even for those benchmarks that seem regular (e.g. CNN), the location constraints from peripheral IPs can highly distort their physical layouts.

- The stencil designs created by the SODA [31] compiler have a set of kernels in linear topologies.
- The genome sequencing design [32] performing the Minimap2 overlapping algorithm [33] has processing elements (PE) in broadcast topology. This benchmark is based on shared-memory communication and all other benchmarks are dataflow designs.
- The CNN accelerators created by the PolySA [34] compiler are in a grid topology.

- The HBM graph processing design [29] performs the page rank algorithm. It features eight sets of processing units and one central controller. This design also contains dependency cycles, if viewed at the granularity of computing kernels.
- The HBM bucket sort design adapted from [35] which includes 8 parallel processing lanes and two fully-connected layers.
- The Gaussian elimination designs created by the AutoSA [36] compiler are in triangle topologies.

5.3 Frequency Improvements

By varying the size of the benchmarks, in total we have tested the implementation of 43 designs with different configurations. Among them, 16 designs failed in routing or placement with the baseline CAD flow, compared AutoBridge which succeeds in routing all of them and achieves an average of 274 MHz. For the other 27 designs, we improve the final frequency from 234 MHz to 311 MHz on average. In general, we find that AutoBridge is effective for designs that use up to about 75% of the available resources. We execute our framework on an Intel Xeon CPU running at 2.2GHz. Both the baseline designs and optimized ones are implemented using Vivado with the highest optimization level. The final checkpoints of all experiments are available in our open-sourced repository.

In some experiments, we may find that the optimized versions have even slightly smaller resource consumption. Possible reasons are that we adopt a different FIFO template and disable the hierarchy rebuild step during RTL synthesis. Also, as the optimization leads to very difference placement results compared to those of the original version, we expect different optimization strategies will be adopted by the physical design tools. The correctness of the code is verified by cycle-accurate simulation.

Next, we present the detailed results of each benchmark.

Stencil Computation.

For the stencil computing design, the kernels are connected in a chain format through FIFO channels. By adjusting the number of kernels, we can vary the total size of the design. We test anywhere from 1 kernels up to 8 kernels, and Figure 11 shows final frequency of the eight design configurations on both U250 and U280 FPGAs. In the original flow, many design configurations fail in routing due to routing resource conflicts. Those that are routed successfully still achieve relatively low frequencies. In comparison, with the help of AutoBridge, all design configurations are routed successfully. On average, we improve the timing from 86 MHz to 266 MHz on the U280 FPGA, and from 69 MHz to 273 MHz on the U250 FPGA.

Starting from the 7-kernel design, we observe a frequency decrease on the U280 FPGA. This is because each kernel of the design is very large and uses about half the resources of a slot; thus starting from the 7-kernel design on the relatively small U280, two kernels have to be squeezed into one slot which will cause more severe local routing congestion. Based on this phenomenon, we recommend that users avoid designing very large kernels and instead split the functionality into multiple functions to allow the tool more flexibility in floorplanning the design.

CNN Accelerator.

The CNN accelerator consists of identical PEs in a regular grid topology. We adjust the size of the grid from a 2×13 array up to a 16×13 array to test the robustness of AutoBridge. Figure 12 shows the result on both U250 and U280 FPGAs.

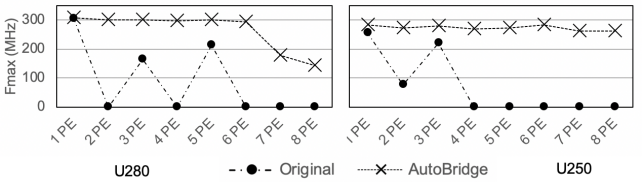

Figure 11: Results of the stencil computation designs.

Although the regular 2-dimensional grid structure is presumed to be FPGA friendly, the actual implementation results from the original toolflow is not satisfying. With the original toolflow, even small size designs are bounded at around 220 MHz when targeting U250. Designs of larger sizes will fail in placement (13×12) or routing (13×10 and 13×14). Although the final frequency is high when the design is small for the original toolflow targeting U280, the timing quality is steadily dropping as the designs become larger.

In contrast, AutoBridge improves from 140 MHz to 316 MHz on U250 on average, and from 214 MHz to 328 MHz on U280. Table 2 lists the resource consumption and cycle counts of the experiments on U250. Statistics on U280 are similar and are omitted here.

Table 2: Post-placement results of the CNN designs on U250. The design point of 13×12 failed placement and 13×10 and 13×14 failed routing with the original tool flow.

Size	LUT(%)		FF(%)		BRAM(%)		DSP(%)		Cycle	
	orig	opt	orig	opt	orig	opt	orig	opt	orig	opt
13x2	17.82	17.90	14.11	14.25	21.69	21.67	8.57	8.57	53591	53601
13x4	23.52	23.59	18.98	19.04	25.74	25.73	17.03	17.03	68630	68640
13x6	29.26	29.24	23.86	23.80	29.80	29.78	25.50	25.50	86238	86248
13x8	34.98	34.90	28.72	28.56	33.85	33.84	33.96	33.96	103882	103892
13x10	40.71	40.48	33.58	33.25	37.91	37.89	42.42	42.42	121472	121491
13x12	-	46.18	-	38.06	-	41.95	-	50.89	139098	139108
13x14	52.10	51.92	43.28	42.93	46.02	46.00	59.35	59.35	156715	156725
13x16	57.82	57.61	48.13	47.70	50.07	50.06	67.81	67.81	174377	174396

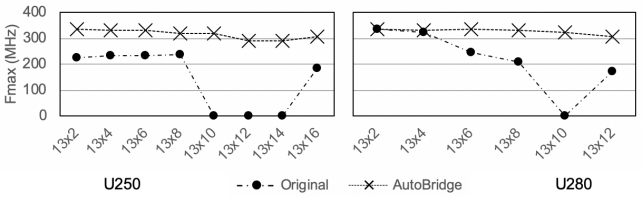

Figure 12: Results of the CNN accelerator designs.

Gaussian elimination.

The PEs in this design form a triangle topology. We adjust the size of the the triangle and test on both U250 and U280. Table 3 shows the results. On average, we improve the frequency from 245 MHz to 334 MHz on U250, and from 223 MHz to 335 MHz on U280.

Table 3: Results of Gaussian Elimination Designs on U250.

Size	LUT(%)		FF(%)		BRAM(%)		DSP(%)		Cycle	
	orig	opt	orig	opt	orig	opt	orig	opt	orig	opt
12×12	18.58	18.69	13.05	13.14	13.24	13.21	2.79	2.79	758	781
16×16	26.62	26.68	17.36	17.30	13.24	13.21	4.99	4.99	1186	1209
20×20	38.55	38.28	23.46	23.38	13.24	13.21	7.84	7.84	1728	1738
24×24	54.05	53.59	32.16	32.06	13.24	13.21	11.34	11.34	2361	2375

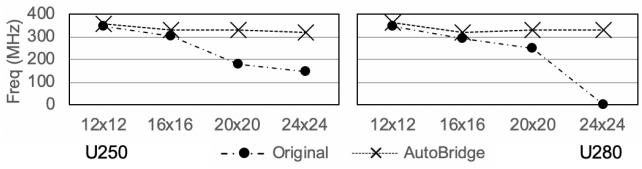

Figure 13: Results of the Gaussian elimination designs.

Genome Sequencing.

The genome sequencing design contains eight parallel PEs that communicate with the external memory through local buffers of a BRAM interface. The design provides parameters to adjust the computation accuracy of each PE and higher accuracy will result in larger area. Thus, we test three design configurations where each PE is of 1×, 1.5× and 2× the original size. For this non-dataflow design, AutoBridge first performs floorplanning and creates a wrapper for each PE to pipeline all I/O signals. Then we manually add pragmas to the source code to specify the modified latency on shared memory blocks and re-run HLS to update the internals of each PE. On average, we improve the frequency from 132 MHz to 248 MHz as in Table 4. When the original size of the PE is small Vivado performs well, but AutoBridge outperforms Vivado with larger PEs.

Table 4: Experiment result of genome sequencing on U250

	Fmax (MHz)	LUT %	FF %	BRAM %	DSP %	Cycle (K)
Orig, Size=1	265	25.43	16.12	17.21	4.38	11710
Opt, Size=1	267	25.48	16.29	17.21	4.38	11830
Orig, Size=1.5	-	-	-	-	-	12350
Opt, Size=1.5	272	31.73	19.39	15.14	6.46	12470
Orig, Size=2	131	38.89	23.11	17.21	8.54	12990
Opt, Size=2	206	38.91	23.31	17.21	8.54	13110

HBM Bucket Sort.

The bucket sort design has two complex fully-connected layers. Each fully-connected layer involves a 8×8 crossbar of FIFO channels, with each FIFO channel being 256-bit wide. AutoBridge pipelines the FIFO channels to alleviate the routing congestion. Table 5 shows the frequency gain, where we improve from 255 MHz to 320 MHz on U280. As the design requires 16 external memory ports and U250 only has 4 available, the test for this design is limited to U280 only.

Because the original source code have enforced a BRAM-based implementation for some small FIFOs, which results in wasted BRAM resources, the results of AutoBridge has slightly lower BRAM and flip-flop consumption than the original implementation. In comparison, we use a different FIFO template that chooses the implementation style (BRAM-based or shift-register-based) based on the area of the FIFO. Cycle accurate simulation has proven the correct functionality of our optimized implementation.

Table 5: Results of the Bucket Sort Design on U280.

	Fmax (MHz)	LUT %	FF %	BRAM %	DSP %	Cycle
Original	255	28.44	19.11	16.47	0.04	78629
Optimized	320	29.39	16.66	13.69	0.04	78632

5.4 Loop-Level Latency Balancing

We employ the HBM Page-Rank design based on the TAPA compiler to demonstrate our loop-level analysis technique presented in Section 4.3. This design incorporates eight sets of processing units, each interfacing with two HBM ports. There are also centralized control units that exchange control information with five HBM ports. As can be seen from the block diagram of this design in Figure 10, if we treat each function as a vertex in the dataflow graph, there will be many dependency loops which results in no valid floorplan solution. However, all loop structures are similar to Figure 8. Therefore, by analyzing at loop level of each function we are able to determine that it is safe to add additional pipelining to the edges in those cycles. This enables us to find feasible floorplan solutions. Table 6 shows the experiment results and we improve final frequency from 136 MHz to 210 MHz on U280.

Table 6: Results of the Graph Processing Design on U280.

	Fmax (MHz)	LUT %	FF %	BRAM %	DSP %	Cycle
Original	136	38.56	26.97	26.74	14.43	120458
Optimized	210	39.49	27.53	30.08	14.43	120495

5.5 Control Experiments

First, we test whether the frequency gain comes from the combination of pipelining and HLS-floorplanning, or simply pipelining alone. To do this, we set a control group where we perform floorplanning and pipelining as usual, but we do not pass the floorplan constraints to the physical design tools. The blue curve with triangle markers in Figure 14 shows the results. As can be seen, the control group has lower frequency than the original design for small sizes and has limited improvements over the original designs for large sizes. In all experiments the group with both pipelining and floorplan constraints (green curve with crossing markers) has the highest frequency. This experiment proves that the frequency gain is not simply a result of more pipelining.

Meanwhile, if we only do floorplanning without pipelining, obviously the frequency will be much degraded, as visualized by Fig. 4.

Second, we test the effectiveness of setting a slot boundary based on the DDR controllers. We run a set of experiments where we only divide the FPGA into four slots based on the die boundaries, minus the division in the middle column. The yellow curve with diamond markers in Figure 14 shows the results. As can be seen, it achieves lower frequency compared to our default eight-slot scheme.

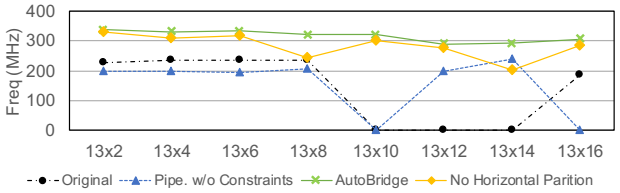

Figure 14: Control experiments with the CNN accelerators.

5.6 Scalability

To show that the tool works well on designs with large numbers of small functions, we utilize the CNN experiments to test the scalability of our algorithms, as the CNN designs have the most vertices (HLS functions) and edges. Table 7 lists The compile time overhead for the floorplanning and the latency balancing when

using Gurobi as the ILP solver[5]. For the largest CNN accelerator that has 493 modules and 925 FIFO connections, the floorplan step only takes around 20 seconds and the latency balancing step takes 0.03s. Usually FPGA designs are not likely to have this many modules and connections [37] [38], and our method is fast enough.

Table 7: Computing time for the CNN test cases targeting the U250 FPGA. *Div-1* and *Div-2* denote the first and the second vertical decomposition, and *Div-3* denotes the first horizontal decomposition. *Re-balance* denotes the delay balancing.

Size	# V	# E	Div-1	Div-2	Div-3	Re-balance
13 × 2	87	141	0.02 s	0.02 s	0.01 s	<0.01 s
13 × 4	145	253	0.05 s	0.02 s	0.20 s	<0.01 s
13 × 6	203	365	0.07 s	1.02 s	0.56 s	<0.01 s
13 × 8	261	477	0.07 s	1.07 s	3.58 s	0.01 s
13 × 10	319	589	3.17 s	1.61 s	2.63 s	0.01 s
13 × 12	377	701	3.42 s	1.43 s	9.84 s	0.01 s
13 × 14	435	813	3.54 s	1.55 s	6.18 s	0.03 s
13 × 16	493	925	4.95 s	2.02 s	12.56 s	0.03 s

6 RELATED WORK

Layout-Aware HLS Optimization. Previous works have studied how to couple physical design process with HLS in a *fine-grained* manner. Zheng *et al.* [1] propose to iteratively run placement and routing for fine-grained calibration of the delay estimation of wires. The long running time of placement and routing prohibits their methods from benefiting large-scale designs, and their experiments are all based on small examples (1000s of registers and 10s of DSPs in their experiments). Cong *et. al.* [6] presented placement-driven scheduling and binding for multi-cycle communications in an island-style reconfigurable architecture. Xu *et. al.* [7] proposed to predict a register-level floorplan to facilitate the binding process. Some commercial HLS tools [8, 9] have utilized the results of logic synthesis to calibrate HLS delay estimation, but they do not consider the interconnect delays. In contrast, we focus on a coarse-grained approach that only pipelines the channels that span long distances and guides the detailed placement.

Other works have studied methods to predict delay estimation at the behaviour level. Guo *et al.* [3] proposed to calibrate the estimated delay for operators with large broadcast factors by pre-characterizing benchmarks with different broadcast factors. Tan *et al.* [2] showed that the delay prediction of logic operations (e.g., `AND`, `OR`, `NOT`, etc) by HLS tools is too conservative. Therefore they consider the technology mapping for logic operations. These works mainly target local operators and have limited effects for global interconnects. Zhao *et al.* [39] used machine learning to predict how the manual pragmas affect routing congestion.

In addition, Cong *et al.* [40] presented tools to allow users to insert additional buffers to the designated datapath. Chen *et al.* [41] proposed to add additional registers to the pipeline datapath during HLS synthesis based on the profiling results on the CHStone benchmark. [42] proposes to generate floorplanning constraints only for systolic array designs, and their method does not consider the interaction with peripheral IPs such as DDR controllers. In comparison, our work is fully-automated for general designs and our register insertion is accurate due to HLS-floorplan co-design.

[5]Meanwhile, we observed that many open-sourced ILP solvers are much slower.

Optimization for Multi-Die FPGAs. To adapt to multi-die FPGAs, previous works have studied how to partition the entire design or memories among different dies [43, 44, 45, 46, 47, 48, 49]. These methods are all based on RTL inputs, thus the partition method must observe the cycle-accurate specification. [46, 47] try to modify the cost function of placement to reduce die-crossing. This will lead to designs confined in fewer dies with higher level of local congestion. Zha *et al.* [50] propose methods to virtualize the FPGA and let different applications execute at different partitions.

Floorplanning Algorithms. Floorplanning has been extensively studied [51, 52, 53, 54]. Conventionally, floorplanning consists of 1) feasible topology generation and 2) determining the aspect ratios for goals such as minimal total wire length. , the floorplanning step works on RTL input. In contrast, we propose to perform a coarse-grained floorplanning during the HLS step to help gain layout information for the HLS tool. Similar to [55, 56, 57], our algorithm adopts the idea of the partitioning-based approach. As our problem size is relatively small, we use ILP for each partitioning.

Throughput Analysis of Dataflow Designs. Various dataflow models have been proposed in other literature, such as the Kahn Process Network (KPN) [58], Synchronous Data Flow (SDF) [20], among many others. The more simplified the model is, the more accurately we can analyze its throughput. In the SDF model, it is restricted that the number of data produced or consumed by a process for each firing is fixed and known. Therefore, it is possible to analytically compute the influence of additional latency on throughput [59]. The latency insensitive theory (LIT) [60, 61, 62, 63, 64] also enforces similar restrictions as SDF. [65] proposes methods to insert delays when composing IP blocks of different latency. [66] studies the buffer placement problem in dataflow circuits [67, 68].

In our situation, each function will be compiled into an FSM that can be arbitrarily complex, thus it is difficult to quantitatively analyze the effect of the added latency on the total execution cycles. Therefore, we adopt a conservative approach to balance the added latency on all reconvergent paths.

7 CONCLUSIONS

We propose to couple coarse-grained floorplanning with pipelining to improves the frequency of the HLS designs on multi-die FPGAs. Our methodology has two key advantages: (1) it helps HLS identify and pipeline the long wires, especially those that will cross die boundaries; (2) it further reduces local routing congestion since early floorplanning can distribute the logic across multiple dies. According our evaluation on 43 realistic benchmarks, our framework effectively improves the average frequency from 147 MHz to 297MHz without compromising the throughput of the design.

ACKNOWLEDGMENTS

We would like to thank Luciano Lavagno, Gai Liu, Zixuan Jiang, Yifan Yuan and the anonymous reviewers for their valuable feedback. This work is partially supported by the CRISP Program, members from the CDSC Industrial Partnership Program, the Intel/NSF CAPA program, the NSF NeuroNex Award No. DBI-1707408 and the NIH Award No. U01MH117079. The authors acknowledge the valuable support of the Xilinx Adaptive Compute Clusters (XACC) Program. We thank Gurobi and GNU Parallel for their support to academia.

REFERENCES

[1] Hongbin Zheng, Swathi T Gurumani, Kyle Rupnow, and Deming Chen. "Fast and effective placement and routing directed high-level synthesis for FPGAs". *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 2014, pp. 1–10.

[2] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. "Mapping-aware constrained scheduling for LUT-based FPGAs". *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2015, pp. 190–199.

[3] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. "Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency". *57th ACM/IEEE Design Automation Conference*. 2020. DOI: 10.1109/DAC18072.2020.9218718.

[4] Charles E Leiserson and James B Saxe. "Retiming synchronous circuitry". *Algorithmica* 6.1-6 (1991), pp. 5–35.

[5] Xilinx. *Xilinx UltraScale Plus Architecture*. 2020. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html.

[6] Jason Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang. "Architecture and synthesis for on-chip multicycle communication". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.4 (2004), pp. 550–564.

[7] Min Xu and Fadi J Kurdahi. "Layout-driven RTL binding techniques for high-level synthesis using accurate estimators". *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 2.4 (1997), pp. 312–343.

[8] Cadence. 2020. URL: https://www.cadence.com/.

[9] Synopsys. 2020. URL: https://www.synopsys.com/.

[10] Charles E Leiserson, Flavio M Rose, and James B Saxe. "Optimizing synchronous circuitry by retiming (preliminary version)". *Third Caltech conference on very large scale integration*. Springer. 1983, pp. 87–116.

[11] Xilinx. *Xilinx Vitis Unified Platform*. 2020. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[12] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. "When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization". *arXiv preprint arXiv:2010.06075* (2020).

[13] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. "HBM Connect: High-Performance HLS Interconnect for FPGA HBM". *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2021.

[14] Xilinx-HBM. 2020. URL: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html.

[15] Intel. *Intel Stratix 10 FPGA*. 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-overview.pdf.

[16] Melvin A Breuer. "A class of min-cut placement algorithms". *Proceedings of the 14th Design Automation Conference*. 1977, pp. 284–290.

[17] Alfred E Dunlop, Brian W Kernighan, et al. "A procedure for placement of standard cell VLSI circuits". *IEEE Transactions on Computer-Aided Design* 4.1 (1985), pp. 92–98.

[18] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. "Fast timing-driven partitioning-based placement for island style FPGAs". *Proceedings of the 40th annual design automation conference*. 2003, pp. 598–603.

[19] Wuxu Peng and S Puroshothaman. "Data flow analysis of communicating finite state machines". *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.3 (1991), pp. 399–442.

[20] Edward A Lee and David G Messerschmitt. "Synchronous data flow". *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[21] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. "Theory of latency-insensitive design". *IEEE Transactions on computer-aided design of integrated circuits and systems* 20.9 (2001), pp. 1059–1076.

[22] Keshab K Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

[23] Minimum-Cut. 2020. URL: https://en.wikipedia.org/wiki/Minimum_cut.

[24] Jason Cong and Zhiru Zhang. "An efficient and versatile scheduling algorithm based on SDC formulation". *2006 43rd ACM/IEEE Design Automation Conference*. IEEE. 2006, pp. 433–438.

[25] Xilinx. *Vivado High-Level Synthesis*. 2020. URL: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[26] H.G. Santos and T.A.M. Toffolo. *Python MIP (Mixed-Integer Linear Programming) Tools*. 2020. URL: https://pypi.org/project/mip/.

[27] Gurobi. 2020. URL: https://www.gurobi.com/.

[28] Shinya Takamaeda-Yamazaki. "Pyverilog: A python-based hardware design processing toolkit for verilog hdl". *International Symposium on Applied Reconfigurable Computing*. Springer. 2015, pp. 451–460.

[29] Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, and Jason Cong. "Extending High-Level Synthesis for Task-Parallel Programs". *arXiv preprint arXiv:2009.11389* (2020).

[30] Xilinx. *Vivado Design Suite*. 2020. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[31] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. "SODA: stencil with optimized dataflow architecture". *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.

[32] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between FPGA and GPU". *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 127–135.

[33] Heng Li. "Minimap2: pairwise alignment for nucleotide sequences". *Bioinformatics* 34.18 (2018), pp. 3094–3100.

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

[35] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. "Bonsai: High-Performance Adaptive Merge Tree Sorting". *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE. 2020, pp. 282–294.

[36] Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA". *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*. 2021.

[37] Xilinx-Vitis-Library. 2020. URL: https://github.com/Xilinx/Vitis_Libraries.

[38] Intel-OpenCL-Examples. 2020. URL: https://www.intel.com/content/www/us/en/programmable/products/design-software/embedded-software-developers/opencl/support.html.

[39] Jieru Zhao, Tingyuan Liang, Sharad Sinha, and Wei Zhang. "Machine learning based routing congestion prediction in FPGA high-level synthesis". *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1130–1135.

[40] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. "Latte: Locality aware transformation for high-level synthesis". *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2018, pp. 125–128.

[41] Yu Ting Chen, Jin Hee Kim, Kexin Li, Graham Hoyes, and Jason H Anderson. "High-Level Synthesis Techniques to Generate Deeply Pipelined Circuits for FPGAs with Registered Routing". *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2019, pp. 375–378.

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

[43] Kalapi Roy and Carl Sechen. "A timing driven N-way chip and multi-chip partitioner". *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE. 1993, pp. 240–247.

[44] Raghava V Cherabuddi and Magdy A Bayoumi. "Automated system partitioning for synthesis of multi-chip modules". *Proceedings of 4th Great Lakes Symposium on VLSI*. IEEE. 1994, pp. 15–20.

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

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

[47] Ehsan Nasiri, Javeed Shaikh, Andre Hahn Pereira, and Vaughn Betz. "Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs". *IEEE Transactions on Very Large Scale Integration Systems* 24.5 (2015), pp. 1821–1834.

[48] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. "Multilevel hypergraph partitioning: applications in VLSI domain". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79.

[49] Nils Voss, Pablo Quintana, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. "Memory Mapping for Multi-die FPGAs". *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 78–86.

[50] Yue Zha and Jing Li. "Virtualizing FPGAs in the Cloud". *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 845–858.

[51] Charles J Alpert, Dinesh P Mehta, and Sachin S Sapatnekar. *Handbook of algorithms for physical design automation*. CRC press, 2008.

[52] Lei Cheng and Martin DF Wong. "Floorplan design for multimillion gate FPGAs". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.12 (2006), pp. 2795–2805.

[53] Pritha Banerjee, Susmita Sur-Kolay, and Arijit Bishnu. "Fast unified floorplan topology generation and sizing on heterogeneous FPGAs". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.5 (2009), pp. 651–661.

[54] Kevin E Murray and Vaughn Betz. "HETRIS: Adaptive floorplanning for heterogeneous FPGAs". *2015 International Conference on Field Programmable Technology (FPT)*. IEEE. 2015, pp. 88–95.

[55] Ulrich Lauther. “A min-cut placement algorithm for general cell assemblies based on a graph representation”. *Papers on Twenty-five years of electronic design automation*. 1988, pp. 182–191.

[56] David P La Potin and Stephen W Director. “Mason: A global floorplanning approach for VLSI design”. *IEEE transactions on computer-aided design of integrated circuits and systems* 5.4 (1986), pp. 477–489.

[57] H Modarres and A Kelapure. “AN AUTOMATIC FLOORPLANNER FOR UP TO 100,000 GATES”. *VLSI Systems Design* 8.13 (1987), p. 38.

[58] KAHN Gilles. “The semantics of a simple language for parallel programming”. *Information processing* 74 (1974), pp. 471–475.

[59] Amir Hossein Ghamarian, Marc CW Geilen, Sander Stuijk, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, Arno JM Moonen, and Marco JG Bekooij. “Throughput analysis of synchronous data flow graphs”. *Sixth International Conference on Application of Concurrency to System Design (ACSD’06)*. IEEE. 2006, pp. 25–36.

[60] Luca P Carloni and Alberto L Sangiovanni-Vincentelli. “Performance analysis and optimization of latency insensitive systems”. *Proceedings of the 37th Annual Design Automation Conference*. 2000, pp. 361–367.

[61] Ruibing Lu and Cheng-Kok Koh. “Performance optimization of latency insensitive systems through buffer queue sizing of communication channels”. *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)*. IEEE. 2003, pp. 227–231.

[62] Ruibing Lu and Cheng-Kok Koh. “Performance analysis of latency-insensitive systems”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.3 (2006), pp. 469–483.

[63] Rebecca L Collins and Luca P Carloni. “Topology-based optimization of maximal sustainable throughput in a latency-insensitive system”. *Proceedings of the 44th annual Design Automation Conference*. 2007, pp. 410–415.

[64] Mustafa Abbas and Vaughn Betz. “Latency insensitive design styles for FPGAs”. *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 360–3607.

[65] Girish Venkataramani and Yongfeng Gu. “System-level retiming and pipelining”. *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2014, pp. 80–87.

[66] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. “Buffer placement and sizing for high-performance dataflow circuits”. *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 186–196.

[67] Lana Josipović, Radhika Ghosal, and Paolo Ienne. “Dynamically scheduled high-level synthesis”. *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 127–136.

[68] Jianyi Cheng, Lana Josipovic, George A Constantinides, Paolo Ienne, and John Wickerson. “Combining Dynamic & Static Scheduling in High-level Synthesis”. *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 288–298.