

# FIFOAdvisor: A DSE Framework for Automated FIFO Sizing of High-Level Synthesis Designs

Stefan Abi-Karam<sup>a</sup>, Rishov Sarkar<sup>a</sup>, Suhail Basalama<sup>b</sup>, Jason Cong<sup>b</sup>, Callie Hao<sup>a</sup>

<sup>a</sup>*Georgia Institute of Technology, Atlanta, GA, United States*

<sup>b</sup>*University of California, Los Angeles, Los Angeles, CA, United States*

---

## Abstract

Dataflow hardware designs are important for efficient algorithm implementations on FPGAs across various domains using high-level synthesis (HLS). However, these designs pose a challenge: correctly and optimally sizing first-in-first-out (FIFO) channel buffers. FIFO sizes are user-defined parameters, introducing a trade-off between latency and area—undersized FIFOs cause stalls and increase latency, while oversized FIFOs waste on-chip memory. In many cases, insufficient FIFO sizes can also lead to deadlocks. Deciding the best FIFO sizes is non-trivial. Existing methods make limiting assumptions about FIFO access patterns, overallocate FIFOs conservatively, or use time-consuming RTL simulations to evaluate different FIFO sizes. Furthermore, we highlight that runtime-based analyses (i.e., simulation) are the only way to solve the FIFO optimization problem while ensuring a deadlock-free solution for designs with data-dependent control flow.

To tackle this challenge, we propose FIFOAdvisor, a framework to automatically decide FIFO sizes in HLS designs. Our approach is powered by LightningSim, a fast simulator that is 99.9% cycle-accurate and supports millisecond-scale incremental simulations with new FIFO configurations. We formulate FIFO sizing as a dual-objective black-box optimization problem and explore various heuristic and search-based methods to analyze the latency–resource trade-off. We also integrate FIFOAdvisor with Stream-HLS, a recent framework for optimizing affine dataflow designs lowered from C++, MLIR, or PyTorch, enabling deeper optimization of the heavily-used FIFOs in these workloads.

We evaluate FIFOAdvisor on a suite of Stream-HLS benchmarks, including linear algebra and deep learning workloads, to demonstrate our approach’s ability to optimize large and dynamic dataflow patterns. Our results show Pareto-optimal latency–memory usage frontiers for FIFO configurations generated via different optimization strategies. Compared to baseline designs with naïvely-sized FIFOs, FIFOAdvisor identifies configurations with much lower memory usage and minimal delay overhead. Additionally, we measure the runtime of our optimization process and demonstrate significant speedups compared to traditional HLS/RTL co-simulation-based approaches, making FIFOAdvisor practical for rapid design space exploration. Finally, we present two case studies using FIFOAdvisor to optimize a complex hardware accelerator with non-trivial data-dependent control flow as well as optimizing for FIFO depths under multiple kernel inputs representative of an application workload.

Code and results open-sourced at <https://github.com/sharc-lab/fifo-advisor>.

*Keywords:* High-Level Synthesis, Buffer Sizing, Design Space Exploration, FPGA

---

## 1. Introduction

Dataflow hardware designs play an important role in domain-specific accelerators [1, 2, 3, 4, 5, 6, 7] including systolic arrays [8, 9, 10, 11, 12], such as AutoSA [13] and SuSy [14]. They allow different tasks to execute simultaneously while communicating through first-in, first-out (FIFO) channels and

---

*Email addresses:* stefanabikaram@gatech.edu (Stefan Abi-Karam), rishov.sarkar@gatech.edu (Rishov Sarkar), basalama@cs.ucla.edu (Suhail Basalama), cong@cs.ucla.edu (Jason Cong), callie.hao@gatech.edu (Callie Hao)

thus improve overall design performance such as latency and throughput.

In designing dataflow architectures using high-level synthesis (HLS), users typically define parallel tasks as functions, which are synthesized into hardware modules, and use FIFO streams to pass data between them. One important consideration is to correctly and optimally decide the depth of each FIFO stream, which has a significant impact on the overall design quality. Small FIFO sizes may result in deadlocks or severe stalls that largely hurt design performance, while large FIFO sizes require large on-chip memory capacity with possible resource and area waste.

Determining the optimal sizes of FIFOs is very challenging. First, the design space can be exponentially large with the growing number of FIFOs in the design, especially given the intertwined effects of multiple FIFOs that cannot be statically analyzed. For example, increasing the sizes of one or more FIFOs does not necessarily improve the design performance. Second, exploring the trade-off between latency and memory is nonlinear and difficult to model. Without proper simulation and synthesis, it is impossible to accurately estimate the performance and memory change because of FIFO size change.

Prior works address the FIFO sizing problem by enforcing producer-consumer rate matching by design, as in systolic array architectures (e.g., PolySA [12] and AutoSA [13]), which assume constant FIFO throughput [15]. Meanwhile, other buffer sizing strategies remain incompatible with traditional HLS design flows (e.g., dynamically scheduled HLS [16]). More fundamentally, static-analysis-based methods are inherently restricted to designs that *do not* include data-dependent control flow, a characteristic common in many modern HLS designs, which limits the overall applicability [17] of prior approaches.

Instead, existing tools and other prior work usually sidestep this problem either by overallocating FIFOs or using time-consuming RTL simulations to test different FIFO sizes. For instance, StreamHLS [18] is a state-of-the-art work that automatically generates HLS dataflow designs for machine learning kernels, but it maximizes FIFO sizes to their respective write counts because of the complexity of choosing these sizes correctly. Meanwhile, Vitis HLS has a tool to repeatedly run RTL simulation for an HLS design with higher and higher FIFO sizes until it no longer deadlocks. As shown in Fig. 1

left, this makes it very challenging to find one feasible solution, let alone explore the latency/memory trade-off—a much harder problem, given the entangled effects of multiple FIFOs.

Addressing the complexity of the problem and limitations of existing tools, we propose **FIFOAdvisor**, a novel framework for design space exploration (DSE) of FIFO sizes for HLS dataflow designs, depicted in Fig. 1. Our work features:

- **Rapid Automated DSE:** FIFOAdvisor is the first work that can find a Pareto frontier of FIFO sizes for arbitrary HLS dataflow designs quickly with push-button ease of use—often in under one minute end-to-end. This is powered by LightningSim [19], which performs incremental simulations in under 1 ms on average per FIFO size change.
- **Comprehensive Evaluation with Multiple Search Algorithms:** We introduce several optimization methods to guide DSE, analyzing each optimizer’s efficiency and solution quality for latency/memory trade-offs. Our framework achieves search runtimes of seconds, versus hours or days for traditional HLS/RTL co-simulation. We also present a case study using FIFOAdvisor to optimize a non-trivial hardware accelerator with data-dependent control flow.
- **Open-Source and Integration with StreamHLS:** We integrate FIFOAdvisor with StreamHLS [18] to automatically optimize the FIFOs in the generated dataflow HLS kernels, and we evaluate FIFOAdvisor on 22 of these kernels, representative of real machine learning applications. Our tool can also be used in a standalone fashion, and we open-source all our code and results.

## 2. Motivation & Prior Work

### 2.1. Static vs. Runtime Analysis

To solve the deadlock-free FIFO sizing optimization problem for a broad class of real-world HLS designs, specifically those exhibiting data-dependent control flow, static analysis alone is insufficient [17]. Static methods cannot model the control flow that might conditionally enable FIFO read and write operations. Therefore, we argue that runtime analysis methods, such as the simulation-based approach proposed in this work, are the only way to find deadlock-free solutions for designs with data-dependent control flow. This is in contrast to prior

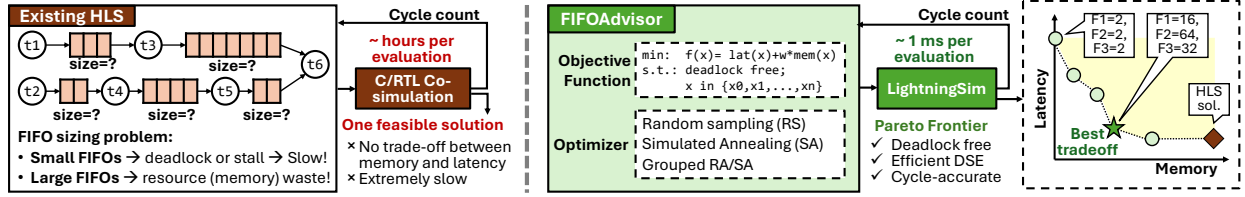


Figure 1: Overview of **FIFOAdvisor**, highlighting the limitations of previous FIFO sizing techniques and the advantages of our multi-objective optimization approach using LightningSim for fast, cycle-accurate simulation. FIFOAdvisor generates a Pareto frontier of candidate solutions that trade off between memory usage reduction and latency overhead.

```
#include <hls_stream.h>
using stream = hls::stream<int>;
void producer(stream &x, stream &y, int n) {
    for (int i = 0; i < n; i++) x.write(1);
    for (int i = 0; i < n; i++) y.write(1);
}
void consumer(int *out, stream &x, stream &y, int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += x.read() + y.read();
    *out = sum;
}
void mult_by_2(int *out, int n) {
    #pragma HLS dataflow
    stream x, y;
    producer(x, y, n);
    consumer(out, x, y, n);
}
```

Figure 2: HLS designs with FIFOs that can not be sized optimally or in a deadlock-free manner without runtime analysis (e.g. simulation)

efforts in FIFO and buffer sizing for HLS as we note in Table 1.

As a supporting example, consider the design in Figure 2. The only way to choose optimal depths for FIFOs  $x$  and  $y$  without possibly incurring a deadlock is to know the value of input argument,  $n$ , passed into the kernel at runtime, information that is not present during static analysis. This example motivates a larger family of common HLS design patterns found in real-world HLS accelerator designs in which data-dependent control flow interacts with FIFO operations, and where only a dynamic, simulation-driven approach can guarantee correct, minimal FIFO sizing.

## 2.2. Prior Work

The problem of optimizing buffer and FIFO sizes in HLS has previously been addressed, but with major limitations that prevent their application to a wide range of practical “real world” HLS designs.

**Static Analysis with Synchronous Dataflow (SDF) Models.** Existing work

| Prior Work            | Method                                     | CT  | IR/W | DDCF |
|-----------------------|--------------------------------------------|-----|------|------|
| Josipovic et al. [16] | Static Analysis + Dataflow Rewrites + MILP | N/A | N/A  | N/A  |
| Cong et al. [15]      | Static Analysis + SDC                      | ✓   | ✗    | ✗    |
| <b>FIFOAdvisor</b>    | Simulation + Black-Box Opt.                | ✓   | ✓    | ✓    |

Table 1: Comparison of Prior Works Related to Buffer and FIFO Sizing. **CT** indicates support for optimizing FIFOs with constant throughput. **IR/W** indicates support for irregular FIFO read/write patterns. **DDCF** indicates support for data-dependent control flow.

on SDF buffer sizing relies on static analysis combined with a system of difference constraints (SDC) optimization model to determine optimal FIFO sizes for HLS designs [15]. However, this approach assumes FIFOs with fixed read and write rates, resulting in constant throughput between dataflow processes. SDF-based analysis methods cannot handle irregular FIFO read and write patterns or data-dependent control flow, which are ubiquitous in modern HLS designs. For example, Stream-HLS designs exhibit highly irregular FIFO read and write timing patterns that violate the design assumptions required by SDF analysis.

**Dynamic HLS Buffer Optimization.** Recent dynamic HLS work uses static analysis, dataflow rewrites, and MILP to optimize buffer sizes [16]. While this appears to address the limitations of SDF analysis, it targets a fundamentally different problem. In dynamic HLS, buffers are implicit in a token-based dataflow protocol: reading from an empty buffer is a no-op, and only writes to full buffers cause stalls. In contrast, Vitis HLS streams have explicit, potentially blocking reads and writes, with timing controlled by data-dependent control flow. Thus, buffer sizing methods for dynamic HLS are not applicable to deadlock-free explicit FIFO sizing in traditional HLS designs, making these optimization techniques inapplicable to the scope of

this work.

**Need for Runtime Analysis.** Even if we ignore the shortcomings of previous methods and try to apply SDC or MILP-based optimization, these techniques still fail because they require static analysis of the HLS program to formulate the optimization problem. With data-dependent or irregular control flow, static analysis cannot capture FIFO access patterns, making such optimization approaches infeasible.

Given the limitations of static analysis and prior work, simulation-based runtime analysis is essential for optimizing FIFO sizes without deadlocks in designs with data-dependent control flow. This motivates our use of LightningSim [19] to enable rapid evaluation of candidate FIFO configurations with fast simulation and runtime program analysis, making it the only practical solution for a large class of HLS designs.

### 3. Methodology

The key challenge in optimizing FIFO sizes lies in the complex interdependencies that make it difficult to predict how changing one FIFO will impact the latency of downstream tasks and the overall design. Consider the dataflow design depicted in Fig. 1 left. Imagine that task  $\tau_1$  is stalling frequently due to a full output FIFO. Will increasing that FIFO’s depth improve the design’s latency? It depends on other tasks in the design—for instance, maybe the true bottleneck is  $\tau_5$ ’s output FIFO.

These inter-task effects are difficult to determine analytically, and doing so may even be impossible if the design has data-dependent control flow [17]. Instead, we formulate the FIFO sizing problem as a black-box multi-objective optimization task. Fig. 1 depicts our optimization flow.

Each candidate solution  $\mathbf{x}$  is a *FIFO configuration*—a set of positive integer FIFO sizes, with a lower bound of 2<sup>1</sup>. Upper bounds, noted as  $u$ , are either the sizes defined in the design, the total number of writes observed during kernel execution, or user-specified.

Given  $\mathbf{x}$ , FIFOAdvisor evaluates two objectives: kernel latency (in cycles),  $f_{lat}$ , using LightningSim, to be described in §3.1, and FIFO BRAM usage,

$f_{bram}$ , using our model defined in §3.2, both as positive integers. These “black-box” functions are inexpensive to evaluate: each run takes milliseconds in single-evaluation mode, and under 1 ms amortized in parallel mode.

We formalize the problem as:

$$\begin{aligned} \text{Minimize} \quad & (f_{lat}(\mathbf{x}), f_{bram}(\mathbf{x})) \in (\mathbb{Z}_{\geq 0}, \mathbb{Z}_{\geq 0}) \\ & \text{LightningSim}(\mathbf{x}) = (f_{lat}(\mathbf{x}), f_{bram}(\mathbf{x})) \\ \text{Subject to} \quad & \text{No Deadlock} \\ & \mathbf{x} = [x_1, x_2, \dots, x_n] \in \mathbb{Z}^n \quad (\text{FIFO sizes}) \\ & 2 \leq x_i \leq u_i, \quad \forall i = 1, \dots, n \end{aligned}$$

We then have the freedom to build any optimizer to evaluate a set of FIFO configurations and minimize the dual objective. Pareto points are then selected from the optimizer-evaluated configurations.

To solve this optimization problem, we integrate LightningSim into the search procedure to obtain  $f_{lat}$  (§3.1), formulate  $f_{bram}$  (§3.2), prune the search space whenever possible (§3.3), and adapt classic optimizers to our problem (§3.4).

#### 3.1. LightningSim and Incremental Simulation

As previously mentioned, hardware simulators like Vitis HLS’s built-in C/RTL co-simulation are far too slow for FIFOAdvisor: each simulation can take several minutes or even hours for large designs, meaning that evaluating a large number of design points would easily take hours or days.

Instead, we propose to utilize the state-of-the-art HLS simulator LightningSim [19]. LightningSim works by collecting an execution trace from the software execution of the HLS code testbenches and separately calculating its latency afterwards. This has the unique advantage that we can re-use the same execution trace to evaluate different FIFO sizes—usually in less than **1 ms** for the designs tested in this work outlined in Table 2.

Thus, at the first invocation, LightningSim collects a full execution trace of a user’s HLS design. We can then incrementally re-simulate the same design under any combination of FIFO sizes and obtain the changed latency  $f_{lat}$  in under 1 ms on average.

#### 3.2. Modeling Memory Usage

While LightningSim provides an accurate model for  $f_{lat}$ , we also need a model for the other major variable impacted by changing FIFO sizes, namely,  $f_{bram}$ . We use FPGA BRAM utilization under Vitis

<sup>1</sup>A FIFO size of 2 is the smallest practical FIFO size, as a size of 1 stalls after the first write—likely the reason Vitis HLS defaults to 2, which we also use.

---

**Algorithm 1** Compute BRAM count for FIFO

---

**Require:** FIFO depth  $d$  and FIFO bitwidth  $w$ **Ensure:** BRAM count  $n$ 

```
 $n \leftarrow 0$ 
if  $d \leq 2 \vee dw \leq 1024$  then return
for each supported BRAM depth  $d_i$  and bitwidth  $w_i$ 
do
   $n \leftarrow n + \lfloor w/w_i \rfloor \cdot \lceil d/d_i \rceil$  and  $w \leftarrow w \bmod w_i$ 
  if  $w > 0 \wedge d \leq d_i$  then
     $n \leftarrow n + 1$  and  $w \leftarrow 0$ 
```

---

HLS as our case study; extending this analysis to ASIC memory is trivial.

FIFOs with a depth of 2 or total size under 1K are implemented by shift registers and do not contribute to BRAM usage. Since BRAM resources are more scarce, we choose to primarily optimize for that metric while ignoring flip-flop overhead. Optimizing both BRAM and FF usage is in the scope of future work. For simplicity, we constrain our modeling to enforce that all FIFOs map exclusively to BRAMs, without initial support for URAM mapping. While extending our approach to URAMs is straightforward, we leave this as future work, with the same BRAM modeling methods directly applying to URAM modeling.

For other FIFOs, prior works [20, 21] propose techniques to calculate the BRAM allocation. However, validated against exhaustive HLS synthesis runs, those techniques overestimate the true number of BRAMs allocated. Instead, our more accurate calculation, validated against these synthesis runs, is shown in Algorithm 1.

BRAM\_18K units support several configurations of row count and bitwidth:  $1K \times 18$ ,  $2K \times 9$ ,  $4K \times 4$ ,  $8K \times 2$ , and  $16K \times 1$ . We first allocate  $1K \times 18$  BRAMs for bitwidths that are multiples of 18 or higher (or sizes under 1K), then  $2K \times 9$  BRAMs satisfy the remaining multiples of 9 (or sizes under 2K), and so on.

Our model targets the UltraScale+ family of Xilinx FPGAs, but should generalize to other device families that use the BRAM18K primitive. We also have test harnesses to quickly evaluate Vitis HLS’s BRAM mapping behavior and can easily derive corresponding models for other device families, tool versions, or memory primitives (e.g., URAM).

### 3.3. Pruning the Design Space

We notice that this function  $f_{\text{bram}}$  has a property that can drastically reduce the size of our search space. As noted in prior work [21], since the

BRAM usage increases in discrete steps at certain hard boundaries in FIFO depth, many FIFO sizes need not be considered. For instance, decreasing a FIFO’s depth from 2048 to 2047 will never change the number of BRAMs but could potentially negatively impact the latency of the HLS design; therefore, we can skip testing the FIFO with depth 2047. We compute these breakpoints for each FIFO in an HLS design and limit our DSE to only those FIFO sizes that maximally utilize their allocated BRAMs.

### 3.4. Included Optimizers

With a fast simulator for latency and a model for BRAM usage, FIFOAdvisor proposes the following optimizers to solve the black-box multi-objective FIFO sizing problem.

**Random Sampling:** The simplest optimizer in FIFOAdvisor randomly samples a user-defined number of FIFO configurations, evaluates each, and extracts the Pareto-optimal points. Sampling random sizes in the range  $2 \leq x_i \leq u_i$  is often ineffective, as only certain threshold sizes affect BRAM usage, as shown in §3.3. Instead, we use our BRAM usage model to suggest optimal sizes for each FIFO, from which the sampler uniformly selects.

**Grouped Random Sampling:** A common HLS pattern is to define FIFO arrays, e.g., `hls::stream<float> data[16]`, suggesting parallel operations across FIFOs with similar access patterns and scheduling. Stream-HLS adopts this computation style, so we modify the random sampling optimizer to select a single FIFO depth for such groups rather than for each FIFO individually. As before, we use the BRAM usage model to suggest candidate sizes—identical to those suggested for individual FIFOs—from which one is randomly selected and applied to the entire group.

**Simulated Annealing:** When the design space is too large for individual or grouped random sampling (even after pruning) or has many infeasible designs, a guided search is needed to better evolve the Pareto frontier. To this end, FIFOAdvisor employs a simulated annealing optimizer that, like the random approaches, selects an index for each FIFO corresponding to one of the sizes from the pruned space.

To address the multi-objective nature of the search, we define a new single objective function as a weighted sum:

$$f(\mathbf{x}) = (1 - \beta)f_{\text{lat}}(\mathbf{x}) + \beta f_{\text{bram}}(\mathbf{x})$$

for  $\beta = \{0, 1/N, 2/N, \dots, 1\}$

The user specifies  $N$ , the number of  $\beta$  values to search, which are linearly interpolated between 0 and 1. The optimizer launches  $N$  simulated annealing runs, one for each  $\beta$ , optimizing the corresponding weighted objective function. After all runs are completed, all the evaluated points across runs are aggregated, and the Pareto frontier is extracted.

**Grouped Simulated Annealing:** Analogous to grouped random sampling, we extend the simulated annealing optimizer to operate on groups of FIFOs defined as arrays, assigning a shared FIFO depth to the entire group.

**Greedy Search:** FIFOs are often naïvely sized to match a large input or output buffer, or to connect faster upstream computation to slower downstream computations—cases where a minimal depth of 2 often incurs no latency penalty. In such scenarios, a simple heuristic can efficiently optimize FIFO sizes. INR-Arch [22] proposes iteratively reducing each FIFO’s depth to 2, ranked from largest to smallest observed depth during simulation, and simulating to check for deadlocks or excessive latency (defined as a fixed percentage over baseline). If a deadlock is triggered or an unreasonable increase in latency occurs, the original depth is restored. FIFOAdvisor adopts this as the greedy search optimizer.

## 4. Evaluations

To evaluate our proposed technique, we need a collection of complex, realistic HLS designs that make heavy use of FIFO streams. The recent state-of-the-art work Stream-HLS [18] presents a suitable case study for this purpose. We collect a set of 21 streaming designs open-sourced by the Stream-HLS authors, enumerated in Table 2, representing a wide variety of HLS kernels useful for machine learning.

We use AMD/Xilinx’s Vitis HLS 2023.2 targeting the Alveo U280 FPGA platform to synthesize the designs. We then run them with LightningSim [19, 23] version 0.2.6, generating an execution trace for each, which can then be used to generate a simulated cycle count for any FIFO depth. To check the accuracy of LightningSim and

| Design                | FIFOs | Co-Sim. | LightningSim | Diff  |
|-----------------------|-------|---------|--------------|-------|
| atax                  | 175   | 2180    | 2181         | +0.0% |
| Autoencoder           | 392   | 39178   | 40073        | +2.3% |
| bicg                  | 25    | 1112    | 1112         | ✓     |
| DepthSepConvBlock     | 84    | 134541  | 134542       | +0.0% |
| FeedForward           | 848   | 65997   | 65998        | +0.0% |
| gemm                  | 88    | 24051   | 24051        | ✓     |
| k2mm                  | 64    | 36352   | 36352        | ✓     |
| k3mm                  | 95    | 49092   | 49092        | ✓     |
| k7mmseq_balanced      | 112   | 5684    | 5684         | ✓     |
| k7mmseq_unbal.        | 108   | 10036   | 10036        | ✓     |
| k7mmtree_unbal.       | 128   | 8750    | 8750         | ✓     |
| mvt                   | 288   | 667     | 668          | +0.1% |
| ResidualBlock         | 64    | 2092531 | 2092532      | +0.0% |
| k15mmseq_imbal.       | 59    | 7802    | 7802         | ✓     |
| k15mmseq              | 188   | 61052   | 61052        | ✓     |
| k15mmseq_relu_imbal.  | 116   | 8504    | 8504         | ✓     |
| k15mmseq_relu         | 232   | 28838   | 28838        | ✓     |
| k15mmtree_imbal.      | 163   | 16237   | 16237        | ✓     |
| k15mmtree             | 192   | 20326   | 20326        | ✓     |
| k15mmtree_relu_imbal. | 340   | 16489   | 16490        | +0.0% |
| k15mmtree_relu        | 320   | 17277   | 17277        | ✓     |

Table 2: The 21 benchmarks we used from Stream-HLS [18], including the number of FIFOs in each design, cycle counts from co-simulation and LightningSim, and their difference. A checkmark (✓) indicates identical cycle counts.

ensure its validity for our case study, we compare its cycle count output with the result of C/RTL co-simulation in Vitis HLS. In all but one of our designs, the LightningSim-computed latency is within **one cycle** of the true cycle count; one edge case (**Autoencoder**) is still within 2.3% of the true cycle count.

For all results shown in Fig. 3 and Fig. 4, each optimizer is given the same budget of 1,000 samples.

### 4.1. Pareto Frontiers of Optimized Solutions

In Fig. 3, we illustrate the Pareto frontiers generated by different optimizers relative to two baseline points: **Baseline-Max** and **Baseline-Min**. The Baseline-Max point represents a design where each FIFO is naïvely maximally sized to fully buffer all the data that passes through it, which guarantees minimum latency. This is the default FIFO sizing that Stream-HLS adopts. The Baseline-Min point represents a design with all FIFOs set to **depth=2**, the default smallest depth, guaranteeing minimum resource usage. By construction, the Baseline-Max solution is always deadlock-free, while the Baseline-Min solution can potentially deadlock. As shown in Fig. 4 (indicated by ✗), only two designs in our evaluation have a Baseline-Min solution that deadlocks, with **k15mmtree** as one of these kernels (also indicated in Fig. 3).

Qualitatively, we see that grouped optimizers, especially grouped simulated annealing, always find Pareto frontiers as good as or better than other op-

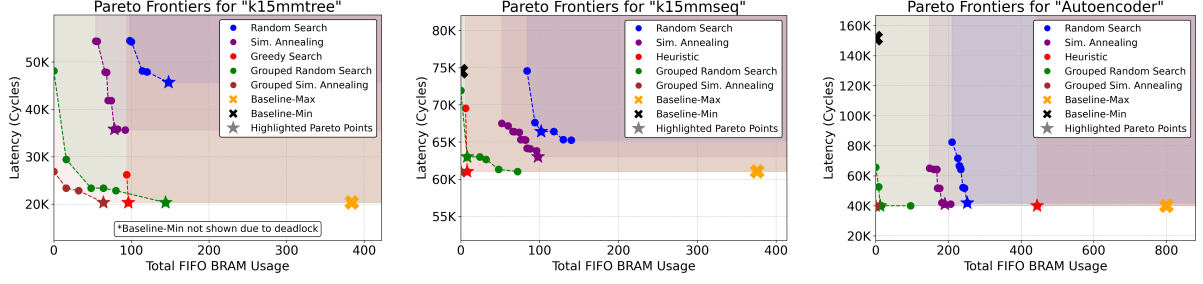


Figure 3: Pareto frontiers for the selected *k15mmtree*, *k15mmseq*, and *Autoencoder* designs. The “highlighted Pareto points” (★, using  $\alpha = 0.7$ ) are used for comparison against the Baseline-Min (✱) and Baseline-Max points (✱) as described in §4.1.

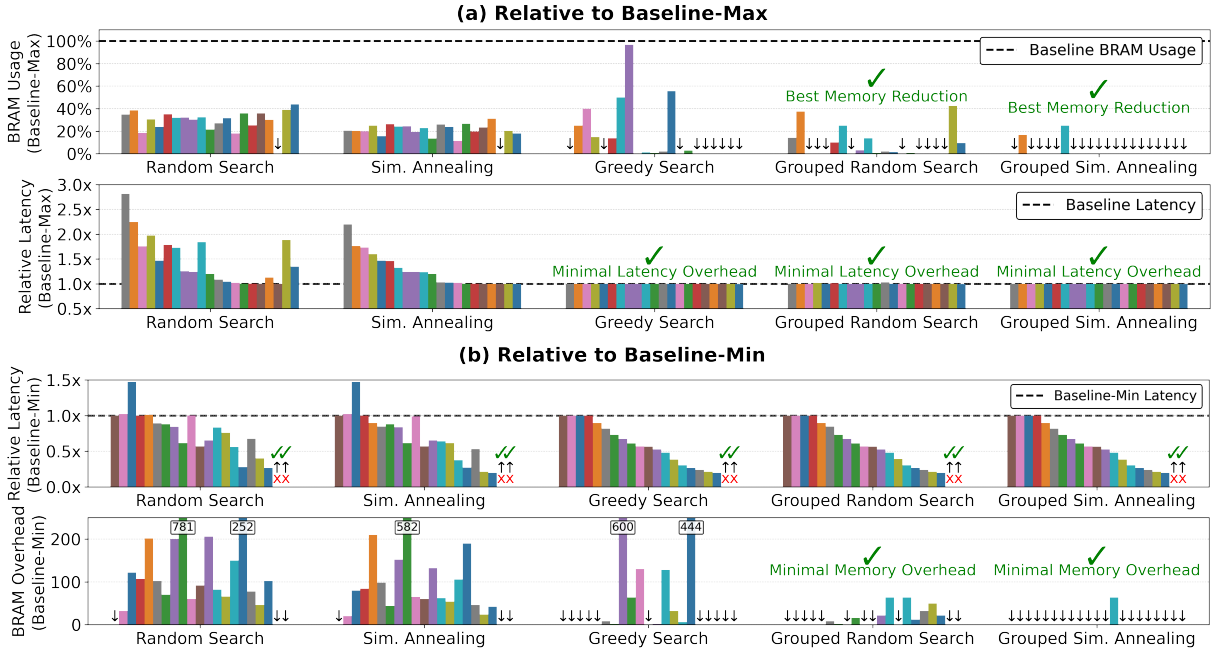


Figure 4: FIFO memory usage and relative latency of “highlighted Pareto points” (★) relative to the naïvely-sized baseline designs (✱ and ✱). Each bar represents a single design. For all metrics, lower is better. The down arrow ↓ indicates no BRAMs were used for all FIFOs in a given design. ✱→✓ indicates that a Baseline-Min design was *un-deadlocked* by FIFOAdvisor.

timizers and baselines. This is true for all designs we tested, including those not depicted.

#### 4.2. Optimized Design Improvement

The obvious question is, *does FIFOAdvisor really improve HLS dataflow designs?* However, since FIFOAdvisor generates a frontier of Pareto-optimal FIFO configurations, there is no one way to answer this question—we must pick a design point from the Pareto frontier and compare it against a baseline configuration. To do this, we devise a simple scoring metric:

$$\alpha \left( \frac{\text{latency}}{\text{baseline latency}} \right) + (1-\alpha) \left( \frac{\text{FIFO BRAMs}}{\text{baseline FIFO BRAMs}} \right)$$

Where  $\alpha$  provides a customizable knob to trade off latency and memory. To select our “optimal” configuration for comparison, we choose  $\alpha = 0.7$  relative to Baseline-Max, which prefers design points that better preserve the Baseline-Max latency even if they do not save as much memory. These selected points are highlighted as stars, ★, in Fig. 3, and will be used to compare against both baselines.

It is important to clarify that these comparisons to the selected “optimal” configuration based on  $\alpha = 0.7$  are for evaluation purposes only. FIFOAdvisor always solves for and provides the user with the full Pareto frontier of solutions, allowing the user to choose any configuration along this fron-

| Design                              | Co-Sim     | FIFO-Advisor              |                           |                           |                           |                           |
|-------------------------------------|------------|---------------------------|---------------------------|---------------------------|---------------------------|---------------------------|
|                                     | (PAR=32)   | Heuristic                 | Rnd.                      | Grp. Rnd.                 | SA                        | Grp. SA                   |
| atax                                | 0.61 days  | 10.68 s.                  | 2.10 s.                   | 3.17 s.                   | 8.62 s.                   | 11.08 s.                  |
| Autoencoder                         | 8.68 days  | 45.39 s.                  | 3.40 s.                   | 3.83 s.                   | 25.60 s.                  | 19.66 s.                  |
| bicg                                | 1.04 days  | 0.00 s.                   | 0.10 s.                   | 0.28 s.                   | 0.10 s.                   | 0.13 s.                   |
| DepthwiseSeparableConvBlock         | 2.06 days  | 7.33 s.                   | 1.76 s.                   | 1.31 s.                   | 18.48 s.                  | 14.82 s.                  |
| FeedForward                         | 16.11 days | 74.20 s.                  | 3.94 s.                   | 5.17 s.                   | 26.01 s.                  | 25.58 s.                  |
| gemm                                | 7.05 days  | 0.10 s.                   | 0.84 s.                   | 0.38 s.                   | 4.11 s.                   | 6.47 s.                   |
| gesummv                             | 0.46 days  | 0.00 s.                   | 0.08 s.                   | 0.15 s.                   | 0.15 s.                   | 0.08 s.                   |
| k15mmseq                            | 1.82 days  | 6.15 s.                   | 0.67 s.                   | 0.42 s.                   | 3.72 s.                   | 3.46 s.                   |
| k15mmseq_imbalanced                 | 0.43 days  | 0.55 s.                   | 0.29 s.                   | 0.28 s.                   | 1.52 s.                   | 1.34 s.                   |
| k15mmseq_relu                       | 1.33 days  | 11.90 s.                  | 0.78 s.                   | 0.82 s.                   | 10.29 s.                  | 7.15 s.                   |
| k15mmseq_relu_imbalanced            | 0.37 days  | 1.06 s.                   | 0.39 s.                   | 0.38 s.                   | 3.03 s.                   | 1.65 s.                   |
| k15mmtree                           | 1.59 days  | 2.35 s.                   | 0.48 s.                   | 0.42 s.                   | 2.32 s.                   | 3.04 s.                   |
| k15mmtree_imbalanced                | 1.42 days  | 1.98 s.                   | 0.41 s.                   | 0.49 s.                   | 2.56 s.                   | 4.09 s.                   |
| k15mmtree_relu                      | 1.36 days  | 13.37 s.                  | 1.00 s.                   | 0.81 s.                   | 7.99 s.                   | 7.54 s.                   |
| k15mmtree_relu_imbalanced           | 1.33 days  | 13.54 s.                  | 0.98 s.                   | 1.54 s.                   | 5.76 s.                   | 5.70 s.                   |
| k2mm                                | 8.46 days  | 0.88 s.                   | 0.79 s.                   | 0.62 s.                   | 5.44 s.                   | 7.24 s.                   |
| k3mm                                | 11.79 days | 5.98 s.                   | 1.01 s.                   | 0.78 s.                   | 7.94 s.                   | 7.21 s.                   |
| k7mmseq_balanced                    | 1.77 days  | 1.11 s.                   | 0.60 s.                   | 0.36 s.                   | 1.31 s.                   | 1.28 s.                   |
| k7mmseq_unbalanced                  | 2.59 days  | 0.87 s.                   | 0.78 s.                   | 0.44 s.                   | 1.42 s.                   | 1.45 s.                   |
| k7mmtree_balanced                   | 2.00 days  | 0.47 s.                   | 0.46 s.                   | 0.31 s.                   | 0.93 s.                   | 1.18 s.                   |
| k7mmtree_unbalanced                 | 2.23 days  | 1.32 s.                   | 0.34 s.                   | 0.31 s.                   | 1.41 s.                   | 1.41 s.                   |
| mvt                                 | 1.66 days  | 0.13 s.                   | 1.44 s.                   | 0.68 s.                   | 7.68 s.                   | 10.81 s.                  |
| ResidualBlock                       | 9.41 days  | 2.06 s.                   | 2.78 s.                   | 2.24 s.                   | 14.00 s.                  | 14.66 s.                  |
| ResMLP                              | 4.94 days  | 141.41 s.                 | 5.77 s.                   | 4.65 s.                   | 26.22 s.                  | 45.87 s.                  |
| <b>FIFO-Advisor Speedup Geomean</b> |            | <b>10<sup>6.53</sup>×</b> | <b>10<sup>6.88</sup>×</b> | <b>10<sup>6.91</sup>×</b> | <b>10<sup>6.20</sup>×</b> | <b>10<sup>6.19</sup>×</b> |

Table 3: Comparison of search runtime of FIFOAdvisor vs. traditional HLS/RTL co-simulation with a given budget of 1,000 samples; Note that, even with best-case estimates and allowing 32 parallel workers for co-simulation, FIFOAdvisor is multiple orders of magnitude faster; **Rnd**: Random Search, **Grp. Rnd**: Grouped Random Search, **SA**: Simulated Annealing, **Grp. SA**: Grouped Simulated Annealing, **PAR=32**: 32 parallel workers.

tier based on their specific design and application trade-offs.

We first compare against Baseline-Max. Results are shown in Fig. 4(a). The random search and simulated annealing perform the worst: both optimizers frequently pick design points with substantial slowdowns (geomeans of 1.40× and 1.23× baseline latency, respectively) and less memory savings than other methods (average of 70.6% and 79.4% BRAM reduction). In contrast, our greedy search optimizer is able to preserve latency (0.9995×<sup>2</sup>) and attain greater memory savings (85.6%). The grouped optimizers identify many ideal configurations with zero BRAMs used by FIFOs and virtually no slowdown (1.0026× baseline latency for grouped random search, and 0.9994× for grouped simulated annealing). This supports the idea that arrays of streams behave similarly and can thus be optimized together.

We then compare against Baseline-Min; results

<sup>2</sup>Astute readers may notice that this geomean is less than 1× of Baseline-Max, which we claimed “guarantees minimum latency.” This is actually because by reducing FIFO sizes from Baseline-Max, some BRAM-backed FIFOs become shift registers instead, and shift register FIFOs have one fewer clock cycle of read delay than BRAM FIFOs, which can sometimes slightly decrease overall latency. This has no impact on the efficacy of FIFOAdvisor or our conclusions.

are in Fig. 4(b). Foremost, we highlight that in all cases where Baseline-Min designs deadlock, FIFOAdvisor finds a *non-deadlocked design with zero BRAM usage*; this alone is novel to FIFOAdvisor. Again, the random search and simulated annealing perform the worst: both optimizers can pick designs with some latency reduction (0.71× and 0.63×) but often with a high BRAM overhead (average of 131.0 and 97.7 BRAMs). In contrast, our greedy search optimizer is able to pick more designs with higher latency reduction (0.53×), but still incurs large BRAM overhead for some designs (67.4 BRAMs). As with before, the grouped optimizers identify many ideal configurations with significant latency reduction (0.53× and 0.52×) and virtually no BRAM overhead (13.9 and 3.0 BRAMs), with grouped simulated annealing as the best choice.

#### 4.3. Optimization Runtime

Across the board, we show that FIFOAdvisor has superior runtime performance solving the FIFO optimization problem compared to HLS/RTL co-simulation. To quantify this speedup, we estimate the co-simulation-based search runtime.

Running co-simulation for hundreds of FIFO configurations is prohibitively long, so we compute a conservative lower bound. We first run co-simulation once per design using maximum FIFO

sizes (i.e. corresponding to Baseline-Max), which minimizes execution cycles due to reduced stalling. Since co-simulation runtime scales with cycle count, this configuration yields the fastest co-simulation runtime as our best-case estimate. We then multiply this best-case runtime by the number of configurations FIFOAdvisor explores for a given search technique to compute the total runtime for an equivalent co-simulation-based search. Results are also reported assuming perfect scaling with 32 parallel workers for co-simulation, with no overhead for task distribution or optimizer logic.

In contrast, we directly measure the runtime of FIFOAdvisor, which evaluates hundreds of FIFO configurations in seconds. As before, all optimizers are limited to exactly 1,000 samples, except for the heuristic approach, which deterministically chooses its own stopping point (ranging from 10 to 2,200 samples across designs in this evaluation).

Table 3 summarizes runtime comparisons across all evaluated Stream-HLS designs, with speedups ranging from  $10^5\times$  to  $10^7\times$ . Even under the most optimistic estimate, co-simulation-based search has no practical advantage. Critically, for designs with data-dependent control flow, only a simulation-based technique can solve the FIFO optimization problem with deadlock-free guarantees. With these runtime results, the choice between FIFOAdvisor and co-simulation is clear for all designs.

We present additional search runtime results in Fig. 5 to illustrate the convergence behavior of different optimizers on a selected design, **k15mmtree**, relative to the Baseline-Max FIFO configuration. Both grouped optimizers converge to the best solutions observed within 6 seconds, which we attribute to effective design space pruning. Notably, the heuristic optimizer also achieves near-optimal results within 2 seconds. These results demonstrate that FIFOAdvisor and the proposed optimizers consistently converge to high-quality solutions within seconds, making FIFOAdvisor ideal for designers who wish to iteratively prototype and optimize domain-specific HLS accelerators.

#### 4.4. Case Study: Data Dependent Control Flow

As discussed in §2, supporting FIFO optimization of designs with data-dependent control flow is crucial for supporting real-world hardware accelerator designs. Therefore, we demonstrate FIFOAdvisor’s ability to optimize FIFOs in a complex hardware accelerator for graph neural networks from a prior work called FlowGNN [7]. These designs contain

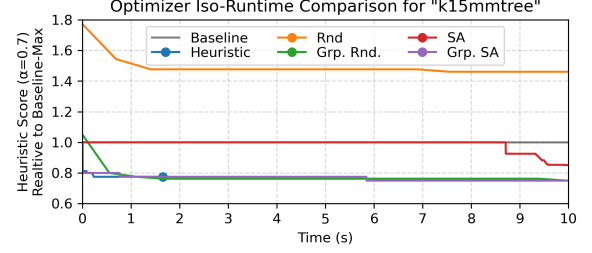


Figure 5: Iso-runtime comparison of different optimizers on the **k15mmtree** design. Runtime includes optimizer logic overhead and calls to LightningSim to evaluate new FIFO configurations.

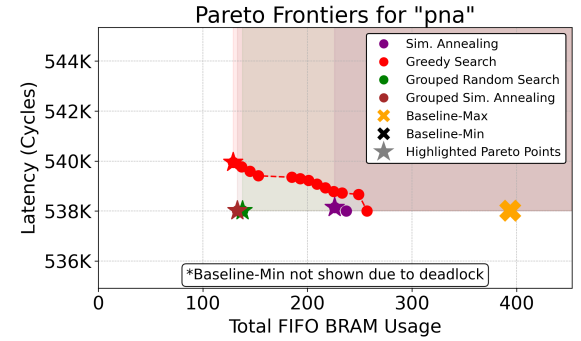


Figure 6: Pareto frontier for the selected **pna** design with data-dependent control flow. The “highlighted Pareto points” (★, using  $\alpha = 0.7$ ) are used to compare against the Baseline-Max point (✱).

FIFOs for scattering and gathering data between nodes on a graph according to the graph connectivity passed in at runtime; this makes FlowGNN an ideal showcase for sizing many FIFOs with non-trivial deadlocking constraints and latency trade-offs.

Without loss of generality, we use FlowGNN’s Principal Neighborhood Aggregation (PNA) design for our case study. We allocate a fixed budget of 5,000 samples per optimizer and all optimizer runs execute in less than 10s.

For this evaluation, the Baseline-Max configuration is modified to match the actual FIFO sizes used in the original FlowGNN PNA accelerator. These sizes were heuristically chosen by the HLS designer to prevent deadlock and minimize resource usage. Unlike Stream-HLS designs, the PNA FIFOs are not maximally sized, but the user-defined Baseline-Max configuration may still be suboptimal in balancing latency and resource utilization.

Fig. 6 illustrates the Pareto frontiers generated

by FIFOAdvisor’s different optimizers. We can see that, like with our previous evaluations on designs with static control flow, our optimizers are able to find practical Pareto-optimal points relative to the user-sized Baseline-Max configuration.

#### 4.5. Case Study: Jointly Optimizing Over Multiple Kernel Inputs

A limitation of FIFOAdvisor as described so far is that it optimizes FIFO depths using only one set of kernel inputs from the testbench. In the current workflow, we invoke the kernel once on a single input, and the resulting simulation trace becomes the sole basis for FIFO-depth optimization.

A configuration that avoids deadlock for one input may still deadlock for another, and a depth that appears optimal for a single input may not remain optimal across a broader distribution of kernel inputs representative of an application workload. Although it is infeasible to optimize over all possible inputs or arbitrary input distributions, most users already possess a representative set of testbench inputs or can generate samples from relevant input distributions for their application. To support this common use case, we extended FIFOAdvisor by creating multi-input variants of FIFO optimizers that operate over a user-provided collection of kernel inputs.

**Implementation:** Referring back to the optimization problem description in Equation 3, two modifications are required in the optimization formulation: (1) aggregating latency and BRAM objectives, and (2) aggregating deadlocking conditions.

Aggregating latency and BRAM objectives only requires the user to supply an aggregation function to aggregate these values over multiple inputs for a single FIFO depth configuration. Although both the `max` and `mean` functions are reasonable candidates, we find that the `mean` function aligns better with FIFO optimization. In practice, one input set almost always produces the highest latency and resource usage for any given configuration. Using `max` causes this single worst-case input to dominate the objective and forces the optimizer to tune FIFO depths relative to that input case alone. By contrast, the `mean` function incorporates all input cases and avoids this domination. While worst-case inputs do not always occur, in our evaluation they are common.

Aggregating deadlocking conditions in the multi-input setting is simpler. A FIFO configuration is

valid only if it avoids deadlock for all user-provided inputs. If any input causes deadlock under a given configuration, that configuration is infeasible even if it succeeds on all other inputs. This rule ensures that any configuration found during optimization is guaranteed not to deadlock for any of the input cases the user provided.

With these two modifications, FIFOAdvisor provides "Multi" variants of all previously described optimizers ("Multi Random Search", "Multi Simulated Annealing", "Multi Greedy", and their grouped variants). These optimizers operate over a user-defined collection of kernel inputs representative of the target application. In practice, users synthesize their design once, create copies of the resulting project, and attach a different testbench to each copy so that each project instance corresponds to one input set. FIFOAdvisor then loads each project copy with its corresponding testbench, traces each using LightningSim, and supplies the resulting list of traces to the "multi" optimizer variants. These optimizers also support parallel evaluation, allowing each configuration to be evaluated concurrently across all LightningSim traces.

**Evaluation:** We evaluate the multi-input optimization approach using the same PNA design from FlowGNN described in §4.4, with the addition of optimizing FIFO depths over multiple kernel inputs. Each input to the PNA kernel is a graph with node and edge features. For this study, we use the first 20 graphs from the `hep10k` dataset [24], a high-energy physics graph dataset also used in FlowGNN evaluations. These graphs vary in size, connectivity, and structural properties, which produce different data-dependent control-flow behaviors. This variability makes the dataset well suited for demonstrating FIFO-depth optimization across multiple inputs.

We evaluate the Multi Grouped Random, Multi Grouped Simulated Annealing, and Multi Greedy optimizers, limiting each optimizer to 5000 evaluations. Although we cap each optimizer at 5000 configurations, each configuration now triggers 20 LightningSim queries—one for each input trace—rather than a single query as in the single-input case. In practice, the overhead of invoking LightningSim remains negligible, and the optimizers benefit from parallelizing queries across the multiple input traces.

Figure 7 shows the average Pareto frontier formed by the FIFO configurations discovered by FIFOAdvisor’s multi-input optimizers for the PNA kernel

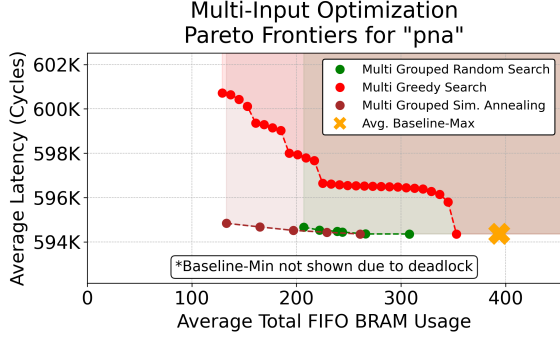


Figure 7: Pareto frontier for the selected **pna** design optimized over 20 different kernel inputs. Optimizers are able to find configurations with reduced BRAM usage given minimal latency overhead relative to the Baseline-Max point (✕).

when jointly optimizing over 20 kernel inputs. To construct this frontier, we first remove any configuration that deadlocks on any input, leaving only configurations that succeed on all 20 designs. For each remaining configuration, we compute its average latency and average BRAM usage across the inputs, and then form the Pareto frontier with respect to these averages. The points in the figure represent this same set of Pareto-optimal configurations, expressed in terms of their aggregated performance across all inputs. We emphasize that all points shown in Figure 7 are valid, non-deadlocking configurations for every input case but optimized for the average input case.

As with the previous PNA evaluation in §4.4, multi-grouped simulated annealing finds the best Pareto frontier in the multi-input setting, with multi-grouped random search performing second best and multi-greedy third. In all cases, the multi-input optimizers produce frontiers that include FIFO configurations with significantly lower BRAM usage than the naively sized average-baseline-max configuration, while incurring only minimal latency overhead.

Lastly, Figure 8 shows that even when we optimize for average latency and resource usage (the "average" input case) we still obtain high-quality Pareto frontiers for all input cases. This result demonstrates that modifying the FIFO-depth optimization objective to use mean aggregation ensures that the optimizer searches for configurations that perform well for each individual input and jointly across all inputs.

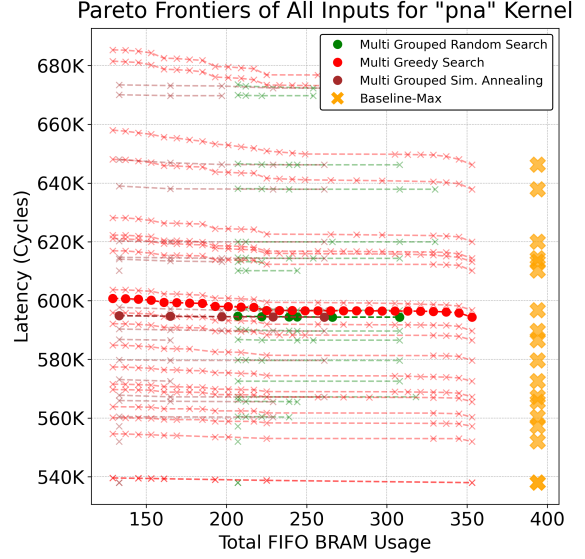


Figure 8: Pareto frontier for all 20 kernel input cases for selected **pna** design optimized over 20 different kernel inputs. The frontier for each input case is optimal for that case alone. We highlight the Pareto frontier optimized for the average case, the same shown in Figure 7. Even when optimizing for the average case, we find high quality frontiers for each input case.

## 5. Conclusion

This work introduces FIFOAdvisor, the first DSE tool for rapidly discovering optimal FIFO sizes in HLS dataflow designs. Combining a fast, cycle-accurate HLS simulator, a FIFO memory usage model, and a multi-objective optimization formulation, FIFOAdvisor efficiently identifies a Pareto-optimal frontier of FIFO depth configurations for a given design, balancing latency and FIFO memory usage. We demonstrate that several of FIFOAdvisor’s optimizers can significantly reduce BRAM usage in real-world dataflow designs with minimal latency impact. We also show that FIFOAdvisor achieves this with minimal runtime, for designs with complex data-dependent control flow, and in a multi-input problem setting. Finally, we open-source our work as a standalone tool for HLS designers and integrate it with Stream-HLS to enable further multi-level optimization of dataflow designs.

## Acknowledgments

This work was partially supported by NSF awards CCF-1937599 and CCF-2211557, the CDSC

industrial partners, the AMD<sup>3</sup> HACC Programs, the PRISM Center under the JUMP2.0 Program sponsored by Semiconductor Research Corporation (SRC) and DARPA, the Georgia Research Alliance based in Atlanta, Georgia under award GRA.25.039.GT.01.a, and NSF award CCF-2338365.

## References

- [1] L. Guo, Y. Chi, J. Lau, L. Song, X. Tian, M. Khatti, W. Qiao, J. Wang, E. Ustun, Z. Fang, Z. Zhang, and J. Cong, “TAPA: A scalable task-parallel dataflow programming framework for modern FPGAs with co-optimization of HLS and physical design,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 16, no. 4, pp. 63:1–63:31, Dec. 2023.
- [2] S. Abi-Karam, R. Sarkar, D. Xu, Z. Fan, Z. Wang, and C. Hao, “Inr-arch: A dataflow architecture and compiler for arbitrary-order gradient computations in implicit neural representation processing,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.
- [3] A. Lerner and G. Alonso, “Data flow architectures for data processing on modern hardware,” in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 5511–5522.
- [4] S. Basalama and J. Cong, “Stream-hls: Towards automatic dataflow acceleration,” in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2025, pp. 103–114.
- [5] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 416–429.
- [6] Y. Kim, S. Jadhav, and C. S. Gloster, “Dataflow to hardware synthesis framework on fpgas,” in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBACPADW)*. IEEE, 2016, pp. 91–96.
- [7] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao, “Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1099–1112.
- [8] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC ’17. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 1–6.
- [9] S. Basalama, J. Wang, and J. Cong, “A Comprehensive Automated Exploration Framework for Systolic Array Designs,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, Jul. 2023, pp. 1–6.
- [10] S. Basalama, A. Sohrabizadeh, J. Wang, and J. Cong, “A Versatile Systolic Array for Transposed and Dilated Convolution on FPGA,” in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2022, pp. 1–2.
- [11] J. Wang and J. Cong, “Search for Optimal Systolic Arrays: A Comprehensive Automated Exploration Framework and Lessons Learned,” Nov. 2021.
- [12] J. Cong and J. Wang, “PolySA: Polyhedral-Based Systolic Array Auto-Compilation,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.
- [13] J. Wang, L. Guo, and J. Cong, “AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 93–104.
- [14] Y.-H. Lai, H. Rong, S. Zheng, W. Zhang, X. Cui, Y. Jia, J. Wang, B. Sullivan, Z. Zhang, Y. Liang, Y. Zhang, J. Cong, N. George, J. Alvarez, C. Hughes, and P. Dubey, “SuSy: A

---

<sup>3</sup>Jason Cong has financial interest in AMD

- programming model for productive construction of high-performance systolic arrays on FPGAs,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD ’20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 1–9.
- [15] J. Cong, M. Huang, and P. Zhang, “Combining computation and communication optimizations in system synthesis for streaming applications,” in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 213–222. [Online]. Available: <https://doi.org/10.1145/2554688.2554771>
- [16] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, “Buffer placement and sizing for high-performance dataflow circuits,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 1, Nov. 2021. [Online]. Available: <https://doi.org/10.1145/3477053>
- [17] S. Ha and H. Oh, “Decidable signal processing dataflow graphs,” in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Cham: Springer International Publishing, 2019, pp. 907–937.
- [18] S. Basalama and J. Cong, “Stream-HLS: Towards automatic dataflow acceleration,” in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’25. New York, NY, USA: Association for Computing Machinery, Feb. 2025, pp. 103–114.
- [19] R. Sarkar and C. Hao, “LightningSim: Fast and accurate trace-based simulation for high-level synthesis,” in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Marina Del Rey, CA, USA: IEEE, May 2023, pp. 1–11.
- [20] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, “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)*. Irvine, CA: IEEE, Nov. 2017, pp. 430–437. [Online]. Available: <http://ieeexplore.ieee.org/document/8203809/>
- [21] A. Honorat, M. Dardaillon, H. Miomandre, and J.-F. Nezan, “Automated buffer sizing of dataflow applications in a high-level synthesis workflow,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, no. 1, pp. 1–26, Mar. 2024.
- [22] S. Abi-Karam, R. Sarkar, D. Xu, Z. Fan, Z. Wang, and C. Hao, “INR-Arch: A dataflow architecture and compiler for arbitrary-order gradient computations in implicit neural representation processing,” in *Proceedings of the 42nd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD ’23. San Francisco, CA, USA: Association for Computing Machinery, Nov. 2023.
- [23] R. Sarkar, R. Paul, and C. Hao, “LightningSimV2: Faster and scalable simulation for high-level synthesis via graph compilation and optimization,” in *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Orlando, FL, USA: IEEE, May 2024, pp. 104–114.
- [24] G. Kasieczka, T. Plehn, J. Thompson, and M. Russel, “Top Quark Tagging Reference Dataset.” [Online]. Available: <https://zenodo.org/record/2603256>