

LAKSA: An MLIR-based Higher-Level Synthesis Compiler for Edge Inference

Jiahong Bi^{ID}, Lars Schütze^{ID}, Jeronimo Castrillon^{ID}

Chair for Compiler Construction, TU Dresden

{jiahong.bi, lars.schuetze, jeronimo.castrillon}@tu-dresden.de

Abstract—Driven by the increasing demand for low-latency and real-time processing, machine learning applications are steadily migrating toward edge platforms, where Field-Programmable Gate Arrays (FPGAs) are widely adopted due to their energy efficiency advantages compared to CPUs and GPUs. To generate high-performance and low-power FPGA designs, several frameworks built upon High Level Synthesis (HLS) vendor tools have been proposed, among which template-based (e.g., FINN) and compiler-based frameworks are gaining traction due to their ease of use. However, existing template-based frameworks often face challenges in extensibility, while frameworks leveraging Multi-Level Intermediate Representation (MLIR) frequently target devices with no stringent resource constraints. To address these limitations, we propose LAKSA, an MLIR-based framework that abstracts and automates the HLS design process for edge devices without relying on template-driven compilation. LAKSA adopts a streaming architecture with carefully managed buffers, specifically designed to accommodate resource constraints while ensuring low latency. Compared to recent MLIR-based frameworks, LAKSA is capable of generating designs that successfully undergo Place and Route (PnR). Even under highly constrained resources and large input sizes, LAKSA achieves a geometric mean of $3.32\times$ speedup compared to the best recent framework. Compared to FINN, LAKSA achieves an average of $1.74\times$ speedup for end-to-end network inference.

Index Terms—MLIR-based Compiler, High Level Synthesis, Edge Inference, FPGA

I. INTRODUCTION

Resource- and compute-intensive Machine Learning (ML) workloads such as Convolutional Neural Network (CNN) demand changes in Internet of Things (IoT) infrastructures, that historically relied on communication technologies and cloud computing. To meet the requirements of low-latency and real-time tasks, an increasing number of applications are shifting towards edge inference, where FPGAs are emerging as a highly promising hardware platform due to their higher energy efficiency [1]–[3].

However, unlike programming for CPUs or GPUs, FPGA programming is inherently challenging. Coding in a Hardware Description Language (HDL) requires expert knowledge and is known to be error-prone due to its low-level concurrent nature. To address this, HLS tools such as AMD’s Vitis [4] and Intel’s Quartus [5] instead generate hardware designs from a high-level program, e.g., in C/C++. However, effectively using them for high-performance designs still requires low-level knowledge, expert use of pragma directives, and an intuition of how the way the source code is written impacts the final resource demands of the design.

To further ease FPGA programming, several frameworks operate at higher levels of abstraction [6]–[16]. Some frameworks are explicitly designed for specific workloads, such as FINN and hls4ml [6]–[8] for ML inference and AutoSA [10] for Systolic Arrays (SAs). Other frameworks are general-purpose, advancing aspects such as polyhedral compilation [14], programmability [9], [13], and dataflow architectures [11], [12], [15], and demonstrate strong performance on ML workloads dominated by large, deeply nested loops.

However, frameworks that rely heavily on pre-defined **templates** [6]–[8] face extensibility challenges. Other **compiler**-based designs do not necessarily suffer from high resource usage, but often lack automated design space exploration (e.g., works [11]–[13]) or do not explicitly account for severely resource-constrained devices (e.g., StreamHLS [15]), particularly with respect to on-chip Block RAM (BRAM) resources.

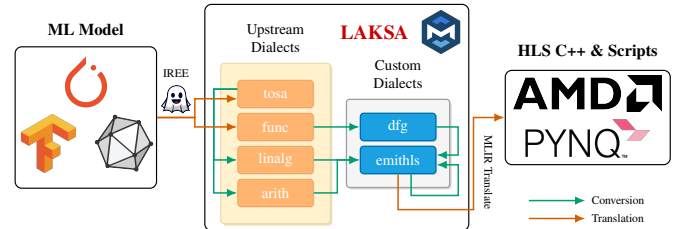


Fig. 1: LAKSA framework overview

To address these challenges, we introduce **LAKSA** (Linear Algebra Kernel Streaming Architecture) for CNN inference, an MLIR-based extensible compilation framework that performs hardware-aware optimizations tailored for highly resource-constrained edge devices. LAKSA combines the resource-awareness of template-based approaches with the high-level abstraction and automation of compiler-based ones. A streaming dataflow architecture is paired with resource optimization techniques under an analytical model, and HLS implementations of a given CNN model are generated automatically. Figure 1 depicts an overview of LAKSA, and its concrete contributions are as follows:

- **Template-free, annotation-free optimizations.** LAKSA automatically classifies MLIR operations as different types directly from its affine indexing maps, and uses this classification to drive all streaming connection and on-chip buffering decisions. Unlike approaches relying on predefined templated layers or user-specified dataflow an-

TABLE I: LAKSA Comparison with Prior Frameworks

Frameworks	Front-end	Streaming Dataflow	Automatic Pragma Insertion	Adaptive Parallelism	Hardware-aware Optimization
HIDA [12]	C/C++/PyTorch	✗	✓	Manual	✗
Allo [13]	Python/PyTorch	Manual	Manual	Manual	✗
StreamHLS [15]	C/C++/PyTorch	✓	✓	✓	Partially
FINN [6], [7]	QONNX	✓	Templates	Partially	✓
LAKSA (this work)	ONNX/TensorFlow/PyTorch	✓	✓	✓	✓

notations, LAKSA infers every implementation decision from its MLIR dialect stack with no user intervention.

- **Joint BRAM and DSP modeling in an ILP-based DSE.** We formulate parallelism selection as an Integer Linear Program (ILP) that simultaneously bounds BRAM and Digital Signal Processor (DSP) usage for integer arithmetic. To the best of our knowledge, LAKSA is the first MLIR-based framework to model both resources jointly in its Design Space Exploration (DSE): the closest prior work, StreamHLS, constrains DSP only, and consequently overflows BRAM at larger input sizes.
- **Deadlock-free FIFO sizing from ILP cycle estimates.** We use the per-stage latency estimates from the ILP solution to size inter-stage First In First Out (FIFO) buffers, eliminating back-pressure deadlocks in the streaming pipeline without resorting to worst-case buffer allocation.
- **End-to-end automated flow.** Starting from a high-level specification (e.g., TensorFlow), LAKSA emits the streaming dataflow architecture as HLS code that completes the full vendor toolchain, including PnR, with no manual intervention.

We evaluate LAKSA against both template-based and compiler-based frameworks on a range of CNN workloads, LAKSA achieves a geometric mean speedup of $3.32\times$ over StreamHLS under highly constrained resources, and an average of $1.74\times$ over FINN in an end-to-end evaluation.

II. RELATED WORKS

Numerous compiler frameworks from academia and industry have been proposed to ease the translation of ML models into FPGA implementations, here we focus on frameworks for deploying CNN models onto FPGAs.

FINN [6], [7], initially proposed by Xilinx/AMD, adopts a hybrid template-based approach. HDL templates provide highly efficient implementations of instances such as data buffers and control logic, while the rich set of utilities and pragmas of the HLS toolflow simplifies and optimizes the kernel design. Similar design methodologies can be found in hls4ml [8], SAMO [17] (a FINN-based framework), and ONNX2MDC [18]. DOSA [19] adds flexibility by selecting and combining operator templates from several such frameworks to achieve more optimized designs.

In contrast to template-based frameworks, which require manually predefined templates for the target operator set, code-generation-based frameworks provide a higher degree of design automation. HeteroCL [9] and its successor Allo [13] follow a Domain Specific Language (DSL)-based approach offering a user-friendly programming interface with user-specified optimization directives, while ScaleHLS [11] and

HIDA [12] apply graph-level pipelining based on kernel parameters available to their MLIR-based compiler, such as loop unroll and tiling factors. However, these approaches still offload most of the optimization effort to users, who need considerable hardware expertise to set the directives. To overcome this limitation, StreamHLS [15] proposed a fully automated MLIR-based framework that adopts a streaming dataflow paradigm for efficient and deterministic inter-node communication, and integrates a DSE that optimizes stream utilization while adhering to the hardware constraints imposed by DSP resources.

There are several other frameworks tailored for specific scenarios. AutoSA [10], for instance, generates SA architectures on FPGA. POM [14] integrates polyhedral optimization [20] into ScaleHLS via isl [21]. The work in [22] adapts various release versions of the Low-Level Virtual Machine (LLVM) Intermediate Representation (IR) for compatibility with the older version that can be compiled in Vitis HLS. Sisyphus [23] employs Large Language Model (LLM)-based prompt understanding to automatically insert appropriate Merlin pragmas [24] into C/C++ source code. Finally, StreamTensor [16] generates energy-efficient hardware for LLMs using a custom tensor representation. These works are orthogonal to LAKSA. While they focus on hardware architecture design and compiler interface, LAKSA focuses on high-level code generation for HLS-based FPGA designs.

Table I compares the aforementioned template-based and compiler-based frameworks. All the listed frameworks can generate optimized FPGA implementations for a given CNN model, with HIDA, Allo and StreamHLS supporting more general inputs. However, existing MLIR-based frameworks place less emphasis on explicitly modeling and managing limited on-board resources, especially BRAM (as discussed in the following section), so their results can be suboptimal or too large to fit on resource-constrained edge devices.

III. BACKGROUND AND MOTIVATION

Deploying CNNs efficiently on FPGA involves navigating a complex design space where compiler infrastructure, hardware resources, and optimization strategies are deeply intertwined. This section examines the key challenges that arise in existing frameworks, introduces the MLIR infrastructure that LAKSA builds upon, and discusses the hardware-aware optimization techniques that are central to our approach.

A. Design Challenges

After critically examining the frameworks listed in Table I for hardware generation, we identify three key challenges that arise in optimization.

The template challenge: In frameworks like FINN, hardware generation relies on composing an operator from multiple predefined templates. While this simplifies design reuse, it constrains flexibility and limits adaptation to novel or unconventional inputs, such as that FINN maps each operator to a composition of templates based on the operator's type and attributes, rather than performing a fine-grained analysis of the underlying computational logic while applying the code transformations. In Section V-C, we show that, with our compiler-based approach, LAKSA outperforms FINN by $1.74\times$ on average (geomean).

The automation challenge: The challenge lies in how much manual input is required, e.g., in the form of pragmas or optimization directives, to enable automatic code transformation and generation. In Allo and HIDA, for instance, domain-specific code is generated automatically from a high-level specification, but optimization is delegated to the user, who must specify parameters such as loop unrolling or tiling factors. Even in FINN's automated end-to-end pipeline, the application of specialized graph transformation passes must still be configured correctly by the user, posing a nontrivial barrier to effective deployment.

The resource challenge: Another critical aspect of optimization for FPGA hardware is adhering to resource constraints. To address this, FINN incorporates fine-grained resource management strategies, for example, leveraging XOR-based logic to implement Multiply-Accumulate (MAC) operations and modeling BRAM utilization according to data storage and partitioning schemes. This capability represents one of the advantages of adopting vendor-specified templates.

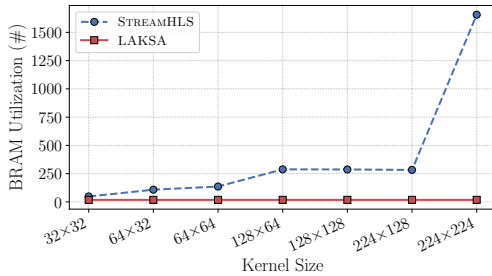


Fig. 2: BRAM utilization – single layer convolution

Among the other works listed in Table I, only StreamHLS provides explicit resource estimation and control through an ILP-based DSE, which models and allocates DSP blocks under resource constraints. However, StreamHLS does not explicitly manage BRAMs, whose utilization typically increases with loop-level optimizations in a manner similar to DSP usage.

Figure 2 illustrates the BRAM utilization as the input size scales for a single-layer convolution followed by a ReLU operator. StreamHLS exhibits significantly higher utilization than LAKSA as the input grows, due to intermediate tensors between dataflow nodes in the computation graph. LAKSA solves this issue with a compiler-based approach augmented with automated hardware-aware optimizations, as detailed in Section IV.

B. MLIR

MLIR [25] is a sub-module within the LLVM framework, designed to provide a comprehensive and reusable compiler infrastructure that supports extensibility and is particularly suited for building DSLs. Its core concept is the notion of *dialects*, which represent different abstraction levels and computational domains, for instance, `TOSA` for tensor operations, `linalg` for linear algebra, and `arith` for arithmetic, each comprising a collection of *operations*, *types*, and *attributes*. The CIRCT [26] project, for instance, defines a rich set of dialects spanning from high-level computation kernels down to circuit-level representations, effectively providing an MLIR-based infrastructure for hardware design.

Another fundamental feature of MLIR is the *pass* mechanism, where *transformations* are applied within the same dialect, while *conversions* translate the IR from one dialect to another, with a variety of built-in passes covering traditional compiler optimizations such as constant folding and common sub-expression elimination.

Motivated by the need for the abstraction of dataflow semantics with strong determinism guarantees, we adopt the `dfg` dialect [27], which is grounded in the dataflow Model of Computation (MoC). Within the MLIR infrastructure, the `dfg` dialect explicitly captures Kahn Process Network (KPN) dataflow primitives such as processes and FIFO channels. In LAKSA, `linalg.generic` operations are encapsulated as dataflow nodes and connected via streaming channels. The corresponding kernel and streaming mechanisms are described in Section IV-A and Section IV-B, respectively.

CIRCT, however, maintains a relatively low-level, Register Transfer Level (RTL)-like programming model and lacks higher-level abstractions and optimizations for HLS-targeted dataflow designs. Therefore, LAKSA adopts HLS as its backend, for which we introduce a new MLIR dialect, `emithls`, to bridge the dataflow representation and HLS code generation, as shown in Figure 1. A concrete example of `emithls` is presented later in Section IV, where we discuss the DSE optimization of LAKSA.

C. Hardware-Aware Optimization

When writing FPGA designs in an HDL, developers can explicitly control specific hardware resources, e.g., configurable Look Up Tables (LUTs), BRAMs, and DSP units. Similarly, in HLS C++, pragmas applied to variables, functions, and loops guide the HLS tool in allocating particular on-chip resources. Such resource-aware optimization is crucial for generating high-performance and energy-efficient implementations, but is not entirely straightforward. We highlight several considerations for achieving an efficient design in terms of both performance and hardware resource utilization.

Node- and graph-level optimizations: In HLS, loop unrolling increases loop-level parallelism by replicating the loop body, while pipelining enables instruction-level pipelined execution within a loop. Both are fundamental for high performance and are instantiated via the `UNROLL` and `PIPELINE` pragmas. Similarly, node-level pipelining is enabled by the

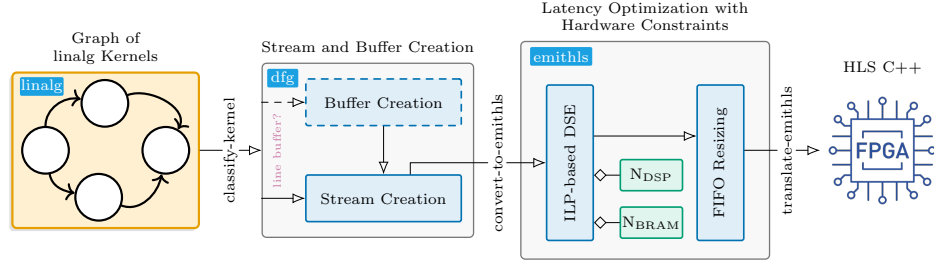


Fig. 3: LAKSA compilation flow

DATAFLOW pragma, which allows individual nodes in a dataflow graph to execute in a pipelined fashion.

Explicit memory management: The HLS tool allows explicit memory allocation for variables such as arrays and FIFO streams by specifying the target memory resource type. When a loop is unrolled, arrays accessed within it must be partitioned along the corresponding dimension according to the unroll factor. Otherwise, memory access conflicts violate the intended unroll semantics and, within a pipelined loop body, can prevent the target Initial Interval (II) from being achieved. Respectively, the `BIND_STORAGE` and `ARRAY_PARTITION` pragmas specify memory types and partition arrays.

FIFO sizing: In a dataflow-based design, the stream connecting two nodes is typically modeled as an elastic circuit [28], which synchronizes producer and consumer through a *valid/ready* handshake protocol. When the filling and draining rates of a stream differ, the handshake may deadlock and stall further execution. A sufficient buffer depth must therefore be specified via the `STREAM` pragma when a variable is implemented as a stream.

The application of these compiler pragmas for improved design quality in LAKSA is further elaborated in Section IV.

IV. LAKSA OPTIMIZATION FLOW

To address the challenges identified in Section III-A, LAKSA transforms tensor operations into a memory-efficient streaming architecture with explicit memory control, and provides an automated DSE that optimizes nested loops while respecting hardware resource constraints. These optimizations are implemented through an analytical model combined with MLIR transformations.

Figure 3 illustrates the key stages of the compilation flow, where the proposed optimizations are applied. LAKSA first classifies the `linalg` kernels in the input graph into different categories based on their attributes. This classification is used to determine whether a line buffer is required and how it should be implemented (Section IV-A). Based on the classification results, LAKSA creates buffers and streams to represent the inputs in the `dfg` dialect, which is subsequently lowered to `emithls` (Section IV-B). Next, an ILP-based DSE transforms the IR toward a higher-performance design based on analysis of the IR and two user-specified resource constraints (Section IV-C). Afterwards, the FIFO depths are resized according to the transformation results to ensure that backpressure does not stall the dataflow. Finally, LAKSA generates HLS code.

A. Kernel Analysis and Classification

In MLIR, the `affine` dialect represents affine loop structures and supports several loop optimization transformations, such as loop unrolling and tiling, as well as dependence analysis based on the IR. All MLIR-based frameworks listed in Table I perform analysis and optimization starting at this level. LAKSA operates at a higher abstraction level using the `linalg` dialect, specifically the `generic` operation.

This operation represents perfectly nested affine loops with statically known bounds and access patterns (control flow branching is not permitted), and offers richer semantics than affine loops, such as explicit mappings between data structures and computations through `affine_maps`. It also explicitly annotates the type of each iterated dimension, e.g., *parallel* or *reduction*, allowing the compiler to better capture program semantics and apply optimizations.

Listing 1 MLIR Conv2D-ReLU example

```

1  #map0 = affine_map<(d0, d1, d2, d3, d4, d5, d6)->(d0,
   ↪ 2*d1+2*d4, 2*d2+2*d5, d6)>
2  #map1 = affine_map<(d0, d1, d2, d3, d4, d5, d6)->(d3,
   ↪  d4, d5, d6)>
3  #map2 = affine_map<(d0, d1, d2, d3, d4, d5, d6)->(d0,
   ↪  d1, d2, d3)>
4  #map3 = affine_map<(d0, d1, d2, d3)->(d0, d1, d2, d3)>
5  %conv_result = linalg.generic {
6    indexing_maps = [#map0, #map1, #map2],
7    iterator_types = ["parallel", "parallel", "parallel",
   ↪  "parallel", "reduction", "reduction",
   ↪  "reduction"]}
8  ins(%input, %weights : tensor<1x32x32x8xi8>,
   ↪  tensor<16x3x3x8xi8>)
9  outs(%empty0 : tensor<1x14x14x16xi32>) {
10   // elided payload operations
11 } -> tensor<1x14x14x16xi32>
12 %relu_result = linalg.generic {
13   indexing_maps = [#map3, #map3],
14   iterator_types = ["parallel", "parallel", "parallel",
   ↪  "parallel"]}
15 ins(%conv_result : tensor<1x14x14x16xi32>)
16 outs(%empty1 : tensor<1x14x14x16xi8>) {
17   // elided payload operations
18 } -> tensor<1x14x14x16xi8>

```

Listing 1 presents an example of Conv2D-ReLU operator written in `linalg.generic` operations with affine maps bounded to the operands. Each `affine_map` consists of two components: the left-hand side enumerates the dimensions of the iteration space (corresponding to the loop induction variables of the generic operation), while the right-hand side

specifies, as affine expressions over these dimensions, the indices used to access the associated operand.

Based on the indexing maps in the `generic` operation, operators in Neural Network (NN)-related workloads can be categorized into three types – Pure Parallel (PP), Sliding Window (SW), and Regular Reduction (RR). PP kernels represent computations in which all dimensions are mutually independent, admitting full parallelization across every dimension (e.g., elementwise operations). SW kernels are characterized by a sliding window memory access pattern (e.g., `Conv2D` and `Pool2D`). RR kernels involve a reduction dimension in the computation but do not exhibit a sliding window access pattern (e.g., `MatMul`), which leads to a distinct buffer allocation strategy compared to the SW category (Section IV-B). Based on these characteristics, the classification problem can further be reduced to recognizing the sliding window semantics while analyzing the kernels.

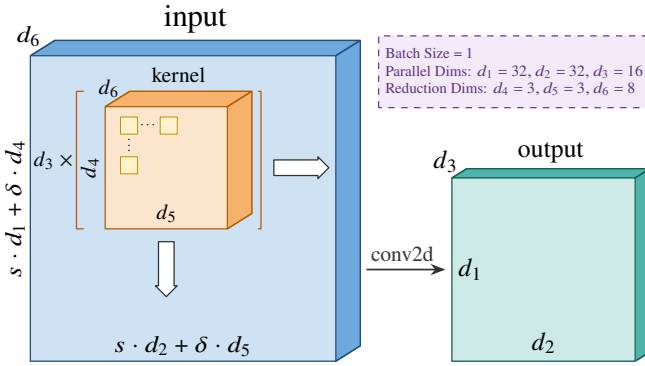


Fig. 4: Conv2D `linalg` kernel analysis

Figure 4 illustrates the `linalg.generic` convolution declared in Listing 1. The output is indexed by the parallel iterators (d_1, d_2), which select where a window is anchored in the input, while the reduction iterators (d_4, d_5) enumerate the taps inside that window. Consequently, in `#map0` the input's spatial coordinates $s \cdot d_1 + \delta \cdot d_4$ and $s \cdot d_2 + \delta \cdot d_5$ encode exactly a sliding window: varying (d_1, d_2) slides the window in steps of s (the stride), and varying (d_4, d_5) walks within it with spacing δ (the dilation).

Algorithm 1 Kernel classification

```

Require: linalg::GenericOp op
Ensure: ( $\mathcal{K}$ , stride, dilation), where  $\mathcal{K} \in \{\text{PP}, \text{RR}, \text{SW}\}$ 
1: if  $\forall t \in \text{op.iterator\_types} = \text{parallel}$  then
2:    $\mathcal{K} \leftarrow \text{PP}$  ▷ e.g., elementwise operation
3: end if
4: for each  $E_{rhs} \in \text{op.indexing\_maps}$  do
5:   if  $E_{rhs} = c_0 \cdot d_{parallel} + c_1 \cdot d_{reduction}$  then
6:      $\mathcal{K} \leftarrow \text{SW}$ , stride  $\leftarrow c_0$ , dilation  $\leftarrow c_1$ 
7:   end if
8: end for
9:  $\mathcal{K} \leftarrow \text{RR}$  ▷ e.g., matrix multiplication

```

We can therefore characterize a kernel with SW access patterns reads at least one input tensor using a linear combination of a spatial iterator and a local accumulation iterator. In

MLIR, such access patterns are canonically represented using an affine expression:

$$E = s \cdot d_{parallel} + \delta \cdot d_{reduction}$$

where $d_{parallel}$ and $d_{reduction}$ denote a parallel and a reduction iterator, respectively. The coefficient $s \in \mathbb{Z}_{>0}$ is the *stride* (step between adjacent windows along the spatial dimension), and $\delta \in \mathbb{Z}_{>0}$ is the *dilation* (spacing between elements inside a window along the reduction dimension).

Therefore, whenever a dimension in an affine map contains a linear combination of exactly one parallel iterator and one reduction iterator with non-zero coefficients (s, δ), the kernel exhibits SW semantics. Affine maps involving reduction iterators but not matching this pattern are classified as RR, while maps containing only parallel iterators are classified as PP. Based on these definitions, we design Algorithm 1. This analysis is also lightweight, operating in $O(\sum |E|)$ time, where $|E|$ is the number of right-hand side expressions across the inspected maps.

B. Stream and Buffer Creation

After classification, LAKSA determines whether a kernel requires a line buffer for storing input data to improve memory efficiency, specifically for kernels that involve a reduction dimension, i.e., SW and RR. If so, a line buffer structure is inserted into the IR in the subsequent step. A line buffer is a well-established hardware-aware optimization widely used in both HDL and HLS designs across various domains, including NNs and image processing. Instead of buffering the entire input, the line buffer stores only the subset of data required for computation over a limited window. This approach significantly reduces memory demand and is particularly beneficial for memory-constrained hardware systems. This is why in Figure 2 LAKSA's BRAM stays flat as input scales from 32×32 to 224×224 while StreamHLS grows approximately linearly. LAKSA applies this optimization automatically by statically analyzing the iterator types and affine maps of each `linalg.generic` operation, which explicitly expose how data is accessed and generated, enabling correct line buffer insertion without user intervention, in contrast to frameworks requiring manual control or template-based designs.

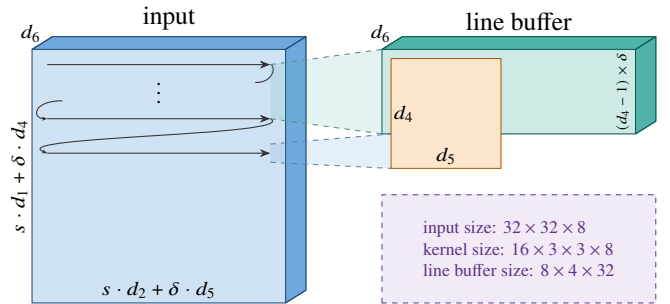


Fig. 5: Line buffer architecture

Figure 5 illustrates how the line buffer is constructed for an SW kernel, using the `Conv2D` generic operation of Listing 1 as a running example. The arrows on the input

cube depict the raster-scan order in which pixels arrive: the horizontal axis indicates one full input row is delivered before moving on, while the vertical axis advances by one row at a time. This naming is intentional — the parallel iterators d_1, d_2 choose where the window is anchored, and the reduction iterators d_4, d_5 walk inside it, so the same expressions that defined the sliding window in Figure 4 now describe the trajectory of incoming data.

To produce one output pixel, the kernel needs a $d_4 \times d_5$ neighborhood whose taps are spaced δ apart, which in the example spans $(d_4 - 1) \cdot \delta + 1 = 5$ input rows. Crucially, the bottom-most of those rows is the one currently being streamed in and is consumed on the fly, so it never has to be stored; only the $(d_4 - 1) \cdot \delta = 4$ rows above it must be retained for reuse with subsequent output pixels. The horizontal extent of each retained row equals the full input width (32), since any column may still be revisited as the window slides along them.

The depth of the buffer is dictated by the streaming interface: d_6 is the only reduction iterator that doesn't appear mixed with a parallel iterator in any indexing map of Listing 1, meaning every input pixel carries $d_6 = 8$ values that must be reduced at the same spatial position; we therefore feed them in parallel through 8 concurrent input streams, and the buffer is 8 deep so that one slot is reserved per stream. By the same reasoning, the parallel iterator d_3 appears only in the kernel and output maps but never in the input map, so it indexes values that are produced rather than consumed at each spatial step; this gives the quantity of $d_3 = 16$ concurrent output streams emitted from the kernel.

Algorithm 2 SW kernel parameter extraction

Require: `linalg::GenericOp op, dilation`

Ensure: $S_{buf}, \mathcal{N}_{in}, \mathcal{N}_{kernel}, S_{in}, S_{out}$

```

1:  $E_{kernel} \leftarrow \phi$ 
2: for each  $(M, val) \in (op.indexing\_maps, operands)$  do
3:    $E_{rhs} \leftarrow M.results, shape \leftarrow val.shape$ 
4:   for each  $(e_{rhs}, size) \in (E_{rhs}, shape)$  do
5:     if  $e_{rhs} = c_0 \cdot d_{parallel} + c_1 \cdot d_{reduction}$  then
6:        $\mathcal{N}_{in} \cup size, E_{kernel} \cup d_{reduction}$ 
7:     else if  $e_{rhs} = d_{reduction}$  then
8:       if  $d_{reduction} \in E_{kernel}$  then
9:          $\mathcal{N}_{kernel} \cup size$ 
10:      else
11:         $S_{in} \cup size$ 
12:      end if
13:    else if  $e_{rhs} = d_{parallel}$  then
14:       $S_{out} \cup size$ 
15:    end if
16:  end for
17: end for
18:  $S_{buf} \cup (\mathcal{N}_{kernel}[0] - 1) \cdot dilation$ 
19:  $S_{buf} \cup \mathcal{N}_{in}[1], S_{buf} \cup size_{in}$ 

```

Constructing this streaming architecture with line-buffer optimization for an SW `linalg.generic` operation therefore boils down to recovering five pieces of information from its indexing maps: the input spatial extents $\mathcal{N}_{in}$ that bound the raster scan, the kernel spatial extents $\mathcal{N}_{kernel}$ that fix the

window size, the input-stream count S_{in} and output-stream count S_{out} that dimension the streaming interface, and finally the line-buffer shape S_{buf} assembled from the previous four. Algorithm 2 performs this extraction by classifying every result expression of every indexing map and accumulating it into the corresponding set, with an auxiliary set E_{kernel} kept on the side to remember which reduction iterators have already been identified as kernel taps.

For RR kernels the same five-set abstraction applies, but extraction is much simpler because no indexing map mixes a parallel and a reduction iterator. In a `matmul generic`, for instance, each output element is obtained by sweeping a single reduction iterator over the inputs with no spatial window to slide, so a reduction iterator can only act as a channel along which values are accumulated, and its size directly fixes the input-stream count S_{in} . The output-stream count S_{out} is recovered exactly as before, from a parallel iterator that indexes the output but not all inputs. With no window to reconstruct across rows, the line buffer degenerates to a single slot per stream holding the operands currently being accumulated, i.e., S_{buf} reduces to $1 \times S_{in}$.

With the kernel type and these five quantities in hand, LAKSA optimizes the IR by allocating the line buffer and instantiating S_{in} inbound and S_{out} outbound streams for the streaming dataflow architecture. The resulting graph is then lowered into the `emithls` dialect, where the extracted parameters directly materialize as the loop bounds and stream declarations that drive the subsequent optimization steps.

C. Automatic DSE and FIFO Resizing

Listing 2 DSE input example

```

1  // Assume: %input: 8-channel streams
2  // and %output: 16-channel streams
3  emithls.for %idx0 = 0 to 32 step 1 {
4  emithls.for %idx1 = 0 to 32 step 1 { // Nin
5  %linebuf = emithls.variable as
6  ↪ !emithls.array<8x4x32xi8>
7  emithls.for %idx2 = 0 to 8 step 1 { // Sin
8  %read = emithls.stream.read %input[%idx2]
9  // elided: line buffer (Sbuf) construction
10 }
11 emithls.if (/* elided: compute condition based on
12 ↪ stride/dilation */) {
13 emithls.for %idx3 = 0 to 16 step 1 { // Sout
14 %accumulator = emithls.variable as i32 = 0
15 emithls.for %idx4 = 0 to 3 step 1 {
16 emithls.for %idx5 = 0 to 3 step 1 { // Nkernel
17 emithls.for %idx6 = 0 to 8 step 1 { // Sin
18 // elided: quantized convolution operations
19 }}}
20 emithls.stream.write %accumulator to %output[%idx3]
21 }}
22 }}

```

Listing 2 shows the `emithls` IR that LAKSA emits for the `Conv2D` running example: the five quantities extracted by Algorithm 2 serve as loop bounds and buffer dimensions — $\mathcal{N}_{in}$ on the outer spatial loops (lines 3–4), S_{in} on the inbound stream-read loop (line 6), S_{out} on the outbound parallel loop (line 11), $\mathcal{N}_{kernel}$ on the window loops (lines 13–14), and S_{buf} as the shape of the `%linebuf` variable (line 5).

The `emithls` dialect adopts a design similar to `emitc` in MLIR (abstracting C/C++ syntax), additionally capturing the pragma- and dataflow-oriented HLS syntax needed to express such streaming architectures, and thereby serves as a common entry point for translation into HLS code.

As discussed in Section III-C, achieving a design that is both resource-efficient and **performance-competitive** on constrained hardware requires careful application of several hardware-aware compiler pragmas. LAKSA regulates hardware resource usage after the line buffer and stream optimizations directly in the IR, as in Listing 2. In LAKSA, line buffers are mapped to BRAM to ensure sufficient storage capacity for large input shapes. Constant arrays are scheduled to use LUT-based memory to provide faster access latency. Finally, multiplication operations are mapped to DSP blocks, which follows the default HLS behavior. Although DSP-based implementations are not always faster than LUT-based ones, this choice provides a better balance among hardware resources and helps avoid potential issues during PnR.

$$\min \left(\max_{p \in \mathcal{P}} \sum_{v \in p} \text{Cycles}(v) \right) \quad (\text{Objective}) \quad (1a)$$

$$\forall \ell \in \mathcal{L} : u_\ell \mid \text{trip}(\ell) \quad (\text{Unroll Constr}) \quad (1b)$$

$$\forall \ell \in \mathcal{L}_{\text{pipe}} : u_\ell = \text{trip}(\ell) \quad (\text{Pipeline Constr}) \quad (1c)$$

$$\sum_{b \in \mathcal{B}} \sum_{\ell \in \mathcal{L}(b)} u_\ell \eta_{\ell b} \leq B_{\text{total}} \quad (\text{BRAM Constr}) \quad (1d)$$

$$\sum_{d \in \mathcal{D}} \sum_{\ell \in \mathcal{L}(d)} u_\ell \eta_{\ell d} \leq D_{\text{total}} \quad (\text{DSP Constr}) \quad (1e)$$

$$\forall s \in \mathcal{S} : \kappa_{\text{src}(s)} = \kappa_{\text{dst}(s)} \quad (\text{Stream Shape}) \quad (1f)$$

$$\forall \ell \in \mathcal{L}, s \in \mathcal{S}(\ell) : \kappa_{\ell(s)} = u_\ell \quad (\text{Stream Access}) \quad (1g)$$

For performance optimization, LAKSA applies loop unrolling and pipelining within the IR, similar to StreamHLS. As discussed before, these are essential techniques for optimizing nested loops in HLS designs. In addition, since LAKSA generates a dataflow-based architecture, we also exploit graph-level pipelining by automatically applying the corresponding pragma in the identified top-level graph. Since these loop optimizations directly affect the utilization of BRAM and DSP resources, LAKSA integrates the following ILP model as Equation (1) to automatically steer the transformations without user-defined parameters while respecting resource constraints, with BRAM and integer arithmetic DSP estimation extending beyond the scope of StreamHLS.

For a given dataflow graph $G = (\mathcal{V}, \mathcal{S})$ with $\mathcal{P}$ different paths, LAKSA's ILP model minimizes the latency, modeled as the cumulated number of clock cycles of the nodes in the graph that construct the longest critical path (Equation (1a)). LAKSA estimates execution cycles in a manner similar to Vitis HLS, determining the number of cycles consumed per loop iteration and then scales this value according to the total iteration count, while accounting for the effects of loop unrolling and pipelining. While Vitis HLS does this at the more accurate RTL level, LAKSA does so at the level of MLIR operations in `emithls` dialect. Although our approach is more coarse-grained, it suffices to steer the optimization

without having to recompile the kernels. Note that LAKSA does not consider loop interchange as StreamHLS does, as it would disrupt the streaming dataflow structure required by the line-buffered streaming design. The constraints of the ILP formulations are described in the following.

Unroll and pipeline constraints: $\mathcal{L}$ and $\mathcal{L}_{\text{pipe}}$ denote the set of loops in the IR (e.g., lines 6, 11, 13–15 of Listing 2) and the subset of loops that reside within a region identified to be placed with the `PIPELINE` pragma, respectively. u_ℓ denotes the unroll factor of a loop ℓ . The unroll factor must divide the total trip count of the loop (Equation (1b)). Since we aim at generating pipelines with an II of one, loops within a `PIPELINE` region are fully unrolled (as required by Vitis HLS). This is represented by $u_\ell = \text{trip}(\ell)$ (Equation (1c)).

BRAM and DSP constraints: On FPGA devices, BRAM resources are typically implemented as BRAM18K blocks storing up to 18,432 bits each. The line buffer in Figure 5 with shape $8 \times 4 \times 32$ and 8-bit integers, for example, requires a single block when the loop at line 6 is not unrolled. Similarly, a DSP block multiplies a 27-bit by an 18-bit integer, so multiplying two 32-bit integers maps each operand to two DSPs, requiring four in total. However, actual resource usage during HLS is more complex than implied in our model, but this estimation provides a theoretical upper bound, ensuring that utilization does not exceed the specified constraints.

Loop optimizations also influence both resources. In an unrolled loop, the increased parallelism leads to an approximately linear increase in DSP usage. Similarly, parallel memory accesses increase with the unroll factor, which requires arrays mapped to BRAM to be properly partitioned to support concurrent accesses. For example, if the loop at line 6 is fully unrolled, the dimension of the line buffer in Figure 5 indexed by `%idx2` must be fully partitioned, resulting in 8 BRAM resources instead of one. In the ILP formulation, $\eta_{\ell b}$ and $\eta_{\ell d}$ denote the hardware resource usage of the operations within loop ℓ for BRAM and DSP, respectively. The total resource consumption is therefore modeled as the product of the loop unroll factor and these coefficients (Equations (1d) and (1e)).

Stream constraints: This constraint specifies that for a FIFO channel connecting two nodes in the dataflow graph, its producer and its consumer must have matching stream shapes. In the case of Listing 2, the producer node connected to it must provide 8 output streams and the consumer must accept 16 input streams. As in Equation (1f), κ denotes the shape of the streams used as a parameter of the node. Additionally, to ensure no performance degradation because of the introduction of streams, κ must also match the read/write access pattern, which is influenced by the loop unrolling configuration (Equation (1g)).

The solution of the LAKSA ILP yields several key variables, such as the loop unroll factors and the locations where `PIPELINE` pragmas should be applied. These variables serve as essential inputs for guiding the subsequent transformations. Additionally, LAKSA integrates a lightweight FIFO resizing mechanism based on the estimated clock cycles per iteration for each node, using the optimization parameters obtained from the ILP. After rewriting the graph according to the optimization parameters, LAKSA resizes the FIFO buffers

based on the number of cycles required by the producer and consumer, thereby eliminating potential race conditions in the synchronous streaming architecture. This step is particularly important for diamond-shaped dataflows, such as residual blocks commonly found in CNN workloads.

V. EVALUATION

In this section, we evaluate LAKSA against a vanilla implementation without specific optimizations, two MLIR-based frameworks, HIDA [12] and StreamHLS [15], as well as FINN [7]. These experiments demonstrate that LAKSA achieves competitive or better performance compared to well-established template-based frameworks such as FINN, despite operating at a high-level of abstraction, and remains competitive against other compiler-based frameworks such as StreamHLS and HIDA with considerably less user intervention.

A. Experimental Setup

To evaluate our work, we select several commonly used kernels from CNN models, specifically those previously evaluated by other frameworks. For convolution kernels, we generate two input sizes (32×32 and 224×224 , which are commonly used in edge inference tasks), while keeping the remaining dimensions identical, in order to highlight the BRAM utilization issue at increased input sizes observed in StreamHLS. This setup is designed to highlight the scalability challenges faced by other frameworks, reflected in the varying BRAM utilization. Some CNNs contain fully connected layers at the end, typically characterized by a small number of dimensions but large feature sizes. To account for this, we design two representative kernels, namely *Linear* and *FeedFwd*, comprising one and two fully connected layers, respectively.

At the edge, CNN models used for inference are typically quantized to reduce the computational overhead associated with floating-point arithmetic, which is particularly expensive on edge FPGAs. For this evaluation, all kernels are quantized to integer precision prior to compilation. For the Vanilla implementation and the MLIR-based frameworks, the kernels are evaluated using 8-bit precision, which is a typical bitwidth used for inference in practice. For FINN, which currently supports up to 4-bit precision, we perform the comparison using 4-bit precision.

Notably, for StreamHLS, which does not provide explicit control over DSP utilization for integer operations and may therefore lead to excessive DSP usage, we integrate the same estimation mechanism used in LAKSA while keeping the remaining framework unchanged. For HIDA, which requires user-specified optimization directives, we use its default configuration, since our comparison targets the benefit of automated DSE over manual tuning; StreamHLS is run with its own ILP-based DSE under a user-specified DSP budget as its best-effort automated configuration. When comparing to FINN, we generate designs using its required ONNX format, which is the same logic as the 8-bit version. Since the translation of 4-bit models to MLIR is not supported, we manually reduce the bitwidth from the corresponding 8-bit model to

4-bit in the IR for evaluation. We verify that the generated HLS C++ code differs only in the declared bitwidth, while the loop structure, pragma insertion, and dataflow topology remain identical to the 8-bit variant; furthermore, the datapath upcasts operands to 32-bit prior to multiplication and accumulation, ensuring no unintended precision loss. This evaluation focuses on resource utilization and latency rather than network-level inference accuracy. Thus, the discussions about network accuracy are irrelevant to this evaluation. Additionally, to eliminate potential variations caused by different MLIR versions, we generate identical inputs using IREE, which is capable of producing MLIR code in the `linalg` dialect.

All experiments are conducted on the KV260 Evaluation Board from AMD [29], which provides **288 BRAM18K** blocks and **1248 DSP** resources operating at 100 MHz using Vivado 2025.1. To further evaluate how LAKSA performs under severe resource constraints, we additionally conduct experiments where the available DSP resources are limited to **125 (10% of the total)**. LAKSA still achieves comparable results, demonstrating its ability to generalize across devices with drastically different resource budgets.

We note that the optimizations in LAKSA are interdependent rather than separable. Line buffering and the ILP jointly keep each design within the BRAM and DSP budgets, and disabling either yields a design that does not fit the device or deadlocks. Rather than an isolated per-optimization ablation, the 100%- and 10%-DSP configurations therefore serve as a sensitivity study on how the ILP adapts parallelism to the available budget. We also collect hardware resource utilization information from both HLS and PnR reports. For FINN, however, only the PnR utilization is reported, as it constructs designs by stitching together templates that are not exclusively implemented as HLS modules.

Regarding performance evaluation, for comparisons with the Vanilla implementation and other MLIR-based frameworks, the presence of large arrays passed as function parameters introduces additional overhead from the board's DRAM accesses. As a result, direct on-board execution would be affected by memory access latency. Therefore, we use the estimated clock cycles reported by HLS for performance comparison. In contrast, FINN provides an end-to-end deployment flow that runs the design directly on hardware. When comparing against FINN, we therefore measure the runtime latency during kernel execution on the board and report the average over 100 runs to obtain a stable, representative measurement. All reported speedups are computed from these two metrics accordingly.

Lastly, we evaluate the BRAM and DSP estimations produced by our ILP model against the corresponding PnR and HLS results. This comparison highlights the discrepancy between the analytical estimates and the final hardware implementation, and motivates further optimization.

B. Results: Comparing to Vanilla, HIDA and StreamHLS

Figure 6 summarizes the results of LAKSA, HIDA, StreamHLS, and the Vanilla baseline in logarithmic scale. Overall, LAKSA achieves the best performance for all workloads, especially in the default configuration without the



SL: Single Layer; SD: Strided-Dilated; TL: Two Layers; DS: Depthwise-Separable; RB: Residual Block; FeedFwd: Feed Forward.

Fig. 6: Comparison among MLIR-based frameworks – estimated latency and resource utilization

strict DSP constraint. In particular, compared with the Vanilla baseline, LAKSA consistently delivers substantial speedups while remaining within hardware limits across all reported workloads, achieving an overall geometric mean speedup of $109.04\times$ for the unconstrained configuration and $7.45\times$ for the 10%-DSP configuration. This result shows that the proposed compilation flow is effective in translating available hardware resources into performance gains.

Figure 6 also highlights important resource trade-offs in the evaluated frameworks, especially in terms of BRAM usage. Compared with HIDA, LAKSA generally uses more BRAM, while still remaining implementable for the reported workloads, likely because HIDA exposes less parallelism in the generated design. As a consequence, only relatively small buffers are needed for data storage, leading to lower BRAM usage. This highlights a common challenge of non-automated

optimization flows, where the final design quality depends strongly on manually selected parameters, as discussed in Section II. By contrast, the exceeding BRAM usage in Vanilla and StreamHLS is mainly caused by intermediate arrays and the array partitioning introduced by parallelism in the case of StreamHLS. This suggests that the BRAM overhead is not simply a drawback of LAKSA compared to HIDA, but more generally a consequence of how intermediate data and parallel access patterns are materialized in the generated hardware. Under this view, the BRAM usage of LAKSA remains significantly better controlled than that of Vanilla and StreamHLS, particularly as the input size increases, while still delivering substantially better performance compared to StreamHLS, with an overall geometric mean speedup of $48.55\times$ for the unconstrained configuration and $3.32\times$ for the 10%-DSP configuration.

Another important observation is the visible discrepancy between the HLS and post-PnR results. This discrepancy appears across multiple frameworks and resources, indicating that resource usage cannot be fully understood from HLS reports alone. In particular, the difference between HLS and post-PnR numbers suggests that backend effects such as resource binding, memory mapping and packing decisions already have a non-negligible impact on the final hardware cost. We further analyze this discrepancy together with the ILP-estimated BRAM and DSP usage of LAKSA later in Section V-D, which provides a more detailed comparison among the estimated, HLS, and post-PnR resource utilization.

For LUT and FF, the discrepancy between the HLS and post-PnR results is mainly caused by the line-buffer and stream-oriented optimizations introduced in LAKSA. These optimizations bring additional control and data-movement logic, which tends to be estimated conservatively by HLS, and therefore appears as a noticeably higher LUT/FF usage in the HLS report. During the downstream implementation, however, this logic can be represented and optimized more precisely at the netlist level, such that redundant control, routing-related overhead, and overly conservative intermediate logic estimates are reduced in the final post-PnR result. Therefore, this gap mainly reflects the limited accuracy of HLS resource estimation for stream-oriented logic, rather than a dependence on backend implementation to compensate for the design.

For HIDA and StreamHLS, a contributing factor to the weaker performance is the mismatch between the intended loop pipelining and the actual memory organization. Although these frameworks set the pipeline II to 1, the corresponding memory system is not always fully adapted to the chosen parallelism. More specifically, the arrays are not partitioned consistently with the parallel execution pattern, so the required memory bandwidth cannot always be provided in time. As a result, the target II may not be fully sustained in the generated hardware, which in turn limits throughput and contributes to the lower speedups observed for HIDA and StreamHLS. At the same time, the better performance of LAKSA is not solely due to the design itself, but also to its ability to identify more suitable parallelism for the target hardware and workload. In this sense, the advantage comes less from a manually tuned implementation and more from selecting a better operating point in the hardware-software design space automatically. This enables the generated design to utilize the available resources more effectively and achieve higher performance.

The LAKSA-10% configuration serves a different purpose. It restricts the available DSP budget to only a small fraction of the device capacity, effectively emulating deployment on a much smaller FPGA and thereby probing the device adaptability of LAKSA. Under such a stringent constraint, the objective is no longer to maximize performance for every workload, but to assess whether the framework can still identify effective parallelization strategies within a severely limited arithmetic budget. As expected, the reduced DSP budget lowers the achievable speedup compared with the unconstrained LAKSA configuration. This behavior is consistent across workloads, although the degree of degradation varies depending on their computational characteristics. Nevertheless, even under this

strict constraint, LAKSA-10% continues to deliver strong performance across the evaluated workloads and remains competitive with, or superior to, the compared baselines. These results indicate that LAKSA can still derive efficient hardware mappings under tight resource budgets, demonstrating that the framework adapts gracefully to smaller, more resource-constrained edge FPGA platforms without requiring any change to the compilation flow.

Overall, Figure 6 shows that LAKSA provides the best performance-resource trade-off among the evaluated works.

C. Results: Comparing to FINN

Figure 7 compares LAKSA and FINN in terms of resource utilization and latency across several representative workloads. Utilization is shown on a logarithmic scale, and the reported speedup is computed as the ratio of FINN's latency to that of LAKSA. For the workloads where latency is available for both frameworks, LAKSA consistently achieves lower latency, with average speedups ranging from $1.11\times$ to $4.92\times$. This result indicates that LAKSA is able to expose substantially more compute parallelism on the target FPGA. For completeness, latency is not reported for `Linear` and `FeedFwd` in FINN, as our FINN flow did not produce valid end-to-end latency measurements for these two workloads under the current evaluation setup. For resource utilization, we report the post-PnR results, since FINN constructs the accelerator by composing multiple HLS modules, making the final implementation results the most meaningful basis for comparison.

The most direct architectural difference is visible in DSP usage. Across all workloads in Figure 7, FINN uses zero DSPs, whereas LAKSA still relies heavily on DSP-based computation. This difference strongly suggests that FINN maps these low-bit operators to logic-oriented arithmetic rather than dedicated multiplier blocks. In practice, such implementations can be realized through bit-level logic, partial-product reduction, and other LUT-based arithmetic structures, instead of consuming DSP resources. By contrast, LAKSA currently preserves a more DSP-centric mapping for multiplication and accumulation, which allows it to sustain higher arithmetic parallelism at the cost of significantly higher DSP usage. The latency advantage of LAKSA therefore comes from a fundamentally different resource trade-off, namely using dedicated arithmetic hardware to reduce execution time, while FINN shifts more of the arithmetic cost into logic. At the same time, these results also highlight that efficient DSP packing is an important optimization opportunity for LAKSA. Since these workloads use low-bit operators, packing multiple operations into a smaller number of DSPs could potentially reduce DSP consumption significantly while preserving much of the performance advantage.

The behavior of LUT and FF is more nuanced. Although LAKSA uses far more DSPs, it does not always use more logic resources than FINN. For example, in `SL-224`, `SD-224`, `TL-224`, `DS-224` (input size 224×224), and `FeedFwd`, LAKSA actually uses fewer LUTs than FINN. A similar trend can also be observed for FFs in some cases. This behavior is consistent with the fact that a larger fraction

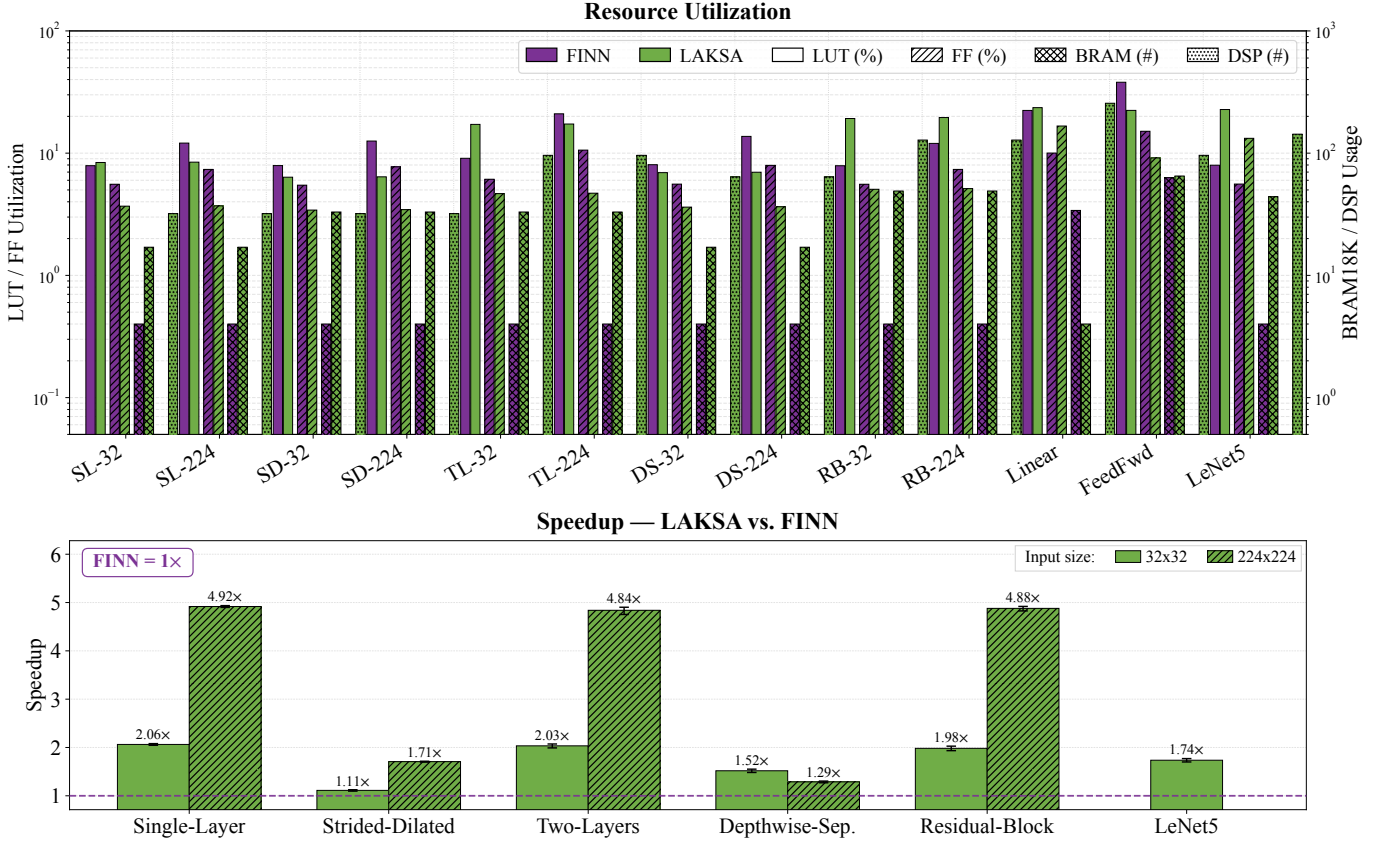


Fig. 7: Comparison between LAKSA and FINN – resource utilization and latency

of arithmetic in LAKSA is absorbed by DSPs rather than materialized as LUT-based datapaths. However, LAKSA can still consume more LUTs and FFs for some workloads, such as RB-32/224 and LeNet5. These cases likely reflect the overhead of the streaming dataflow realization itself, including additional routing logic, synchronization, FIFO control, and inter-stage buffering required to maintain high parallelism, with part of this overhead stemming from the use of HLS-generated FIFO implementations in LAKSA, as opposed to the more specialized HDL-based implementation adopted in FINN. In other words, the trade-off is not simply DSP versus LUT, but rather a combination of DSP-backed compute and extra stream orchestration overhead.

The BRAM results show a more nuanced discrepancy between the two frameworks. Although LAKSA in most cases uses more BRAM than FINN, the gap remains moderate for most workloads, and interestingly, for *Linear*, less BRAM is consumed. This behavior is reasonable because LAKSA and FINN use on-chip memory differently.

Another interesting observation is the different sensitivity to input size. For LAKSA, moving from 32×32 to 224×224 changes LUT, FF, BRAM, and DSP only marginally for the kernels we evaluate. This suggests that the generated hardware structure is determined primarily by the selected parallelization and buffering strategy, while the larger input mainly increases the latency rather than the amount of instantiated hardware. In contrast, FINN shows more noticeable growth in LUT and FF for the larger-input variants. This indicates that, for these

workloads, the generated logic in FINN is more sensitive to tensor size, even though its DSP usage remains zero.

Overall, Figure 7 highlights a clear design trade-off between the two frameworks. LAKSA achieves consistently lower latency by exploiting aggressive parallelism and DSP-based arithmetic, but this advantage comes with substantially higher DSP usage due to the higher parallelism exposed by LAKSA. These observations also suggest that enabling DSP packing and reuse would be a particularly meaningful direction for reducing the resource gap of LAKSA without giving up its performance advantage.

D. Results: Estimation vs. Reality

Figure 8 compares the estimated resource usage from the LAKSA ILP model and the HLS report with the final post-PnR implementation results across different kernels and optimization variants. Both Est and HLS bars are shown on a logarithmic scale and normalized to the post-PnR result (dashed red line at $1\times$), so that values above one indicate overestimation and values below one indicate underestimation. Several clear trends can be observed from the figure. First, the ILP model generally captures the correct resource trend across kernels and variants, namely that larger or less constrained designs tend to consume more BRAM and DSP, while the 10%-DSP-constrained variants consistently reduce DSP usage. This indicates that the model is effective as a hardware-aware optimization guide, even though its absolute accuracy is limited.

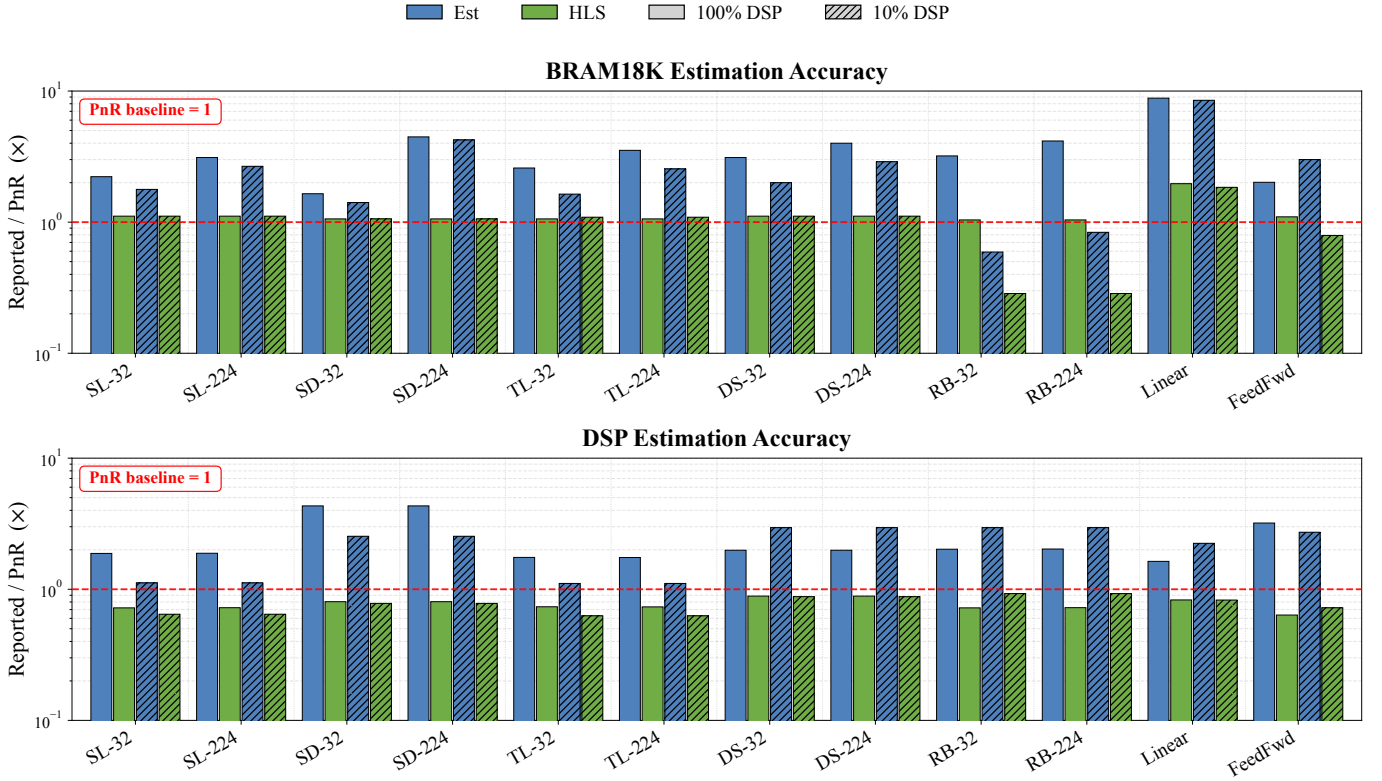


Fig. 8: Comparison of BRAM and DSP utilization – estimation vs. reality

At the same time, the gap between the three results is often substantial. For many kernels, the ILP estimation is noticeably more conservative than both the HLS and post-PnR results, especially for BRAM. This behavior is particularly evident in the *Linear* and *FeedFwd* kernels, where the estimated BRAM usage ratio reaches around $9\times$ and $3\times$ the post-PnR value, respectively. A similar trend can also be observed for DSP, although the discrepancy is usually smaller than for BRAM in the constrained variants. These results suggest that the ILP model approximates resource pressure at a coarse granularity and therefore behaves more like a safe upper-bound estimate than an exact predictor, as discussed in Section IV-C.

Another important observation is that the HLS report is often closer to the final implementation than the ILP estimate, but it still does not fully match the post-PnR numbers. In particular, the post-PnR DSP usage is consistently higher than the HLS report across all kernels. This indicates that part of the actual DSP cost is introduced or exposed only after downstream synthesis and physical implementation. By contrast, BRAM usage is in many cases slightly lower in post-PnR than in HLS, which suggests that the backend may further optimize memory packing or mapping for some kernels. However, this behavior is not uniform, as shown by the 10%-DSP variants of *RB-32* and *RB-224*, where post-PnR BRAM usage becomes significantly larger than both the ILP and HLS results.

The existence of these discrepancies is expected because the three results are produced at different abstraction levels. The ILP model is built at compile time and mainly captures resource usage through analytical formulations of parallelism

and buffering requirements. As a result, it cannot fully model implementation-specific effects such as memory bank merging and DSP resource reuse. The HLS report already includes scheduling and binding decisions, but it is still generated before logic synthesis and placement-and-routing, and therefore cannot fully predict backend transformations as well. These backend effects become more pronounced for structurally complex kernels.

Overall, Figure 8 shows that the main value of the ILP model is not to reproduce the exact post-PnR numbers, but to provide a reliable early-stage estimate of resource trends for optimization and design-space exploration.

VI. CONCLUSION AND FUTURE WORK

In this work, we presented LAKSA, an MLIR-based compilation framework that addresses key design challenges in FPGA accelerator generation for resource-constrained devices by combining hardware-aware optimizations drawn from template-based approaches with automated optimization parameter selection at a high level of abstraction as in compiler-based approaches. These components enable LAKSA to systematically expose hardware parallelism while respecting hardware resource limits, achieving a $3.32\times$ speedup over StreamHLS under strict resource constraints. Beyond performance, LAKSA demonstrates that vendor-specific optimizations need not be encoded in fixed operator templates, but can instead be captured and applied analytically within the compiler, resulting in a more systematic, extensible, and template-independent optimization flow, achieving an average of $1.74\times$ speedup in an end-to-end evaluation against FINN.

We see several promising directions for future work. First, incorporating predictors such as Balor [30] could improve QoR estimation and enable better parallelism decisions. Second, ideas from FIFOAdvisor [31] could improve FIFO sizing, and BRAM-based FIFO implementations should be considered explicitly within the ILP model for deeper buffers. Third, throughput estimation methods from FINN could help balance dataflow nodes and improve end-to-end pipeline efficiency. More broadly, LAKSA is designed as an open framework that can be embedded into other flows and absorb techniques from them. For instance, its streaming architecture is naturally compatible with AutoSA-style systolic generation, extending applicability to Transformer-like and GEMM-dominated workloads. Larger designs may additionally benefit from physical optimizations such as AutoBridge [32] to achieve higher operating frequency. Finally, improved DSP packing [33], [34] represents another valuable direction to reduce DSP consumption while preserving performance. Finally, we see great potential in interfacing with upstream MLIR DSLs flows beyond ML workloads such as [35], [36] from the EVEREST project [37] and how multiple implementation variants can dynamically be deployed for adaptivity in the computing continuum [38].

ACKNOWLEDGMENTS

This work was funded by the EU Horizon Europe Programme under grant agreement No 101135183 (MYRTUS). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

During the preparation of this manuscript, the authors used Claude (Anthropic) solely to improve the wording, grammar, and readability of the text. The authors reviewed and edited all resulting text and take full responsibility for the content of this publication.

REFERENCES

- [1] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "FPGA HLS Today: Successes, Challenges, and Opportunities," vol. 15, no. 4, pp. 51:1–51:42.
- [2] C. Xu, S. Jiang, G. Luo, G. Sun, N. An, G. Huang, and X. Liu, "The Case for FPGA-Based Edge Computing," vol. 21, no. 7, pp. 2610–2619.
- [3] F. N. Peccia, S. Pavlitska, T. Fleck, and O. Bringmann, "Efficient Edge AI: Deploying Convolutional Neural Networks on FPGA with the Gemini Accelerator," in *2024 27th Euromicro Conference on Digital System Design (DSD)*, pp. 418–426.
- [4] V. Kathail, "Xilinx Vitis Unified Software Platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Association for Computing Machinery, pp. 173–174.
- [5] Altera. High Level Synthesis Compiler — Quartus® Prime Design Software. [Online]. Available: <https://www.altera.com/products/development-tools/quartus-prime/hls-compiler>
- [6] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. Association for Computing Machinery, pp. 65–74.
- [7] M. Blott, T. B. Preußner, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," vol. 11, no. 3, pp. 16:1–16:23.
- [8] J.-F. Schulte, B. Ramhorst, C. Sun, J. Mitrevski, N. Ghielmetti, E. Lupi, D. Danopoulos, V. Loncar, J. Duarte, D. Burnette, L. Laatu, S. Tzelepis, K. Axiotis, Q. Berthet, H. Wang, P. White, S. Demirsoy, M. Colombo, T. Aarrestad, S. Summers, M. Pierini, G. D. Guglielmo, J. Ngadiuba, J. Campos, B. Hawks, A. Gandrakota, F. Fahim, N. Tran, G. Constantines, Z. Que, W. Luk, A. Tapper, D. Hoang, N. Paladino, P. Harris, B.-C. Lai, M. Valentin, R. Forelli, S. Ogreni, L. Gerlach, R. Flynn, M. Liu, D. Diaz, E. Khoda, M. Quinlan, R. Solares, S. Parajuli, M. Neubauer, C. Herwig, H. F. Tsoi, D. Rankin, S.-C. Hsu, and S. Hauck. Hls4ml: A Flexible, Open-Source Platform for Deep Learning Acceleration on Reconfigurable Hardware.
- [9] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. Association for Computing Machinery, pp. 242–251.
- [10] 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. Association for Computing Machinery, pp. 93–104.
- [11] H. Ye, C. Hao, J. Cheng, H. Jeong, H. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 741–755.
- [12] H. Ye, H. Jun, and D. Chen, "HIDA: A Hierarchical Dataflow Compiler for High-Level Synthesis," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24, vol. 1. Association for Computing Machinery, pp. 215–230.
- [13] H. Chen, N. Zhang, S. Xiang, Z. Zeng, M. Dai, and Z. Zhang, "Allo: A Programming Model for Composable Accelerator Design," vol. 8, pp. 171:593–171:620.
- [14] W. Zhang, J. Zhao, G. Shen, Q. Chen, C. Chen, and M. Guo, "An Optimizing Framework on MLIR for Efficient FPGA-based Accelerator Generation," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 75–90.
- [15] 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. Association for Computing Machinery, pp. 103–114.
- [16] H. Ye and D. Chen, "StreamTensor: Make Tensors Stream in Dataflow Accelerators for LLMs," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. Association for Computing Machinery, pp. 201–216.
- [17] A. Montgomerie-Corcoran, Z. Yu, and C.-S. Bouganis, "SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 418–424.
- [18] F. Manca, F. Ratto, and F. Palumbo, "ONNX-To-Hardware Design Flow for Adaptive Neural-Network Inference on FPGAs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, L. Carro, F. Regazzoni, and C. Pilato, Eds. Springer Nature Switzerland, pp. 85–96.
- [19] B. Ringlein, F. Abel, D. Diamantopoulos, B. Weiss, C. Hagleitner, and D. Fey, "DOSA: Organic Compilation for Neural Network Inference on Distributed FPGAs," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, pp. 43–50.
- [20] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on a low-level intermediate representation," vol. 22, no. 04, p. 1250010.
- [21] S. Verdoolaege, "Isl: An Integer Set Library for the Polyhedral Model," in *Mathematical Software – ICMS 2010*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds. Springer, pp. 299–302.
- [22] G.-M. Liang, C.-Y. Yuan, M.-S. Yu, T.-L. Chen, K.-H. Chen, and J.-K. Lee, "The Support of MLIR HLS Adaptor for LLVM IR," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22. Association for Computing Machinery, pp. 1–8.
- [23] S. Pouget, L.-N. Pouchet, and J. Cong, "A Unified Framework for Automated Code Transformation and Pragma Insertion," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '25. Association for Computing Machinery, pp. 187–198.
- [24] Xilinx. (2026) Merlin-compiler. Xilinx. [Online]. Available: <https://github.com/Xilinx/merlin-compiler>

- [25] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14.
- [26] LLVM Project. (2026) CIRCT: Circuit IR compilers and tools. [Online]. Available: <https://circt.llvm.org/>
- [27] S. Soldavini, F. Suchert, S. Curzel, M. Fiorito, K. Friebe, F. Ferrandi, R. Cmar, J. Castrillon, and C. Pilato, "Etna: MLIR-Based System-Level Design and Optimization for Transparent Application Execution on CPU-FPGA Nodes," in *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 224–224.
- [28] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 657–662.
- [29] Xilinx. (2026) Kria KV260 Vision AI Starter Kit. AMD. [Online]. Available: <https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html>
- [30] E. Murphy and L. Josipović, "Balor: HLS Source Code Evaluator Based on Custom Graphs and Hierarchical GNNs," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24. Association for Computing Machinery, pp. 1–9.
- [31] S. Abi-Karam, R. Sarkar, S. Basalama, J. Cong, and C. Hao, FIFOAdvisor: A DSE Framework for Automated FIFO Sizing of High-Level Synthesis Designs.
- [32] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. Association for Computing Machinery, pp. 81–92.
- [33] J. Sommer, M. A. Özkan, O. Keszocze, and J. Teich, "DSP-Packing: Squeezing Low-precision Arithmetic into FPGA DSP Blocks," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 160–166.
- [34] R. Bosio, F. Minnella, T. Urso, M. R. Casu, L. Lavagno, M. T. Lazarescu, and P. Pasini, "NN2FPGA: Optimizing CNN Inference on FPGAs With Binary Integer Programming," vol. 44, no. 5, pp. 1807–1818.
- [35] S. Soldavini, K. F. A. Friebe, M. Tibaldi, G. Hempel, J. Castrillon, and C. Pilato, "Automatic creation of high-bandwidth memory architectures from domain-specific languages: The case of computational fluid dynamics," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 16, no. 2, Mar. 2023. [Online]. Available: <https://doi.org/10.1145/3563553>
- [36] K. F. A. Friebe, J. A. Ohlmann, and J. Castrillon, "Demonstrating a future for mlir-native dsl compilers on a numpy-like example," 2026. [Online]. Available: <https://arxiv.org/abs/2604.19906>
- [37] C. Pilato, S. Bohm, F. Brocheton, J. Castrillon, R. Cevasco, V. Cima, R. Cmar, D. Diamantopoulos, F. Ferrandi, J. Martinovic, G. Palermo, M. Paolino, A. Parodi, L. Pittaluga, D. Raho, F. Regazzoni, K. Slaninova, and C. Hagleitner, "EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms," in *Proceedings of the 2021 Design, Automation and Test in Europe Conference (DATE)*, ser. DATE'21, Feb. 2021, pp. 1320–1325. [Online]. Available: <https://ieeexplore.ieee.org/document/9473940>
- [38] F. Palumbo, M. K. Zedda, T. Fanni, A. Bagnato, L. Castello, J. Castrillon, R. D. Ponte, Y. Deng, B. Driessen, M. Fadda, T. H. du Fretay, J. de Oliveira Filho, V. Rao, F. Regazzoni, A. Rodríguez, M. Schranz, and G. Sedda, "MYRTUS: Multi-layer 360 dynamic orchestration and interoperable design environment for compute-continuum systems," in *Proceedings of the 21st ACM International Conference on Computing Frontiers (CF'24)*, ser. CF '24 Companion. New York, NY, USA: Association for Computing Machinery (ACM), May 2024, p. 101–106.



Jiahong Bi received his B.E. degree from Xidian University, Xi'an, China, in 2019 and the M.S. degree from TU Dresden, Germany, in 2023. He is currently working and pursuing Ph.D. degree at the Chair for Compiler Construction, TU Dresden, under Prof. Dr.-Ing. Jeronimo Castrillon's supervision. His research interests include domain-specific language for non-trivial hardware (e.g., FPGA, AMD AIE, Cerebras WSE) with automatic design-space exploration and compilation.



Lars Schütze received the B.Sc. and M.Sc. degrees in computer science, and the Ph.D. degree from TU Dresden, Germany, in 2024. He is currently a postdoctoral researcher at the Chair for Compiler Construction, TU Dresden. His doctoral research studied role-based programming languages and improved their runtime efficiency by applying code generation and partial evaluation at runtime. He has a background in UML modeling and metamodeling, optimizing compilers, and just-in-time compilation, and was involved in the verification of OCL constraints in models and programs within the DresdenOCL project. His current research interests include domain-specific languages and compilers, particularly for hybrid quantum-classical computing. He has co-chaired international conferences and workshops, serving as Euro-Par workshop co-chair and as co-chair of the COP workshop co-located with ECOOP.



Jeronimo Castrillon is a professor in the Department of Computer Science at the TU Dresden, where he is also affiliated with the Center for Advancing Electronics Dresden (CfAED), the Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig (ScaDS.AI), and the Barkhausen Institut. He is the head of the Chair for Compiler Construction, with research focus on programming and design methodologies as well as computer architectures for complex emerging computing systems.

He received a degree in Electronic Engineering from the Pontificia Bolivariana University in Colombia in 2004, his masters degree from the University of Lugano in Switzerland in 2006 and his Ph.D. degree (Dr.-Ing.) with honors from the RWTH Aachen University in Germany in 2013. In 2014, Prof. Castrillon co-founded Silexica GmbH/Inc, a company that provided programming tools for heterogeneous architectures, now with Xilinx/AMD. In 2023, he co-founded Xronos, a company that develops tools and services for safe cyber-physical systems. Prof. Castrillon has more than 150 international publications, is Senior Member IEEE, Senior Member ACM, and is a member of the editorial board of the ACM TODAES and IEEE TC. In 2025 he received a Consolidator Grant from the European Research Council (ERC).