# FRIDA: Reconfigurable Arrays for Dynamically Scheduled High-Level Synthesis

Louis Coulon
School of Computer and Communication Sciences, EPFL
Lausanne, Switzerland
louis.coulon@epfl.ch

Lucas Ramirez
School of Computer and Communication Sciences, EPFL
Lausanne, Switzerland
lucas.ramirez@epfl.ch

Jason Anderson
Electrical and Computer Engineering Department, University of Toronto
Toronto, Canada
janders@ece.utoronto.ca

Mirjana Stojilović
School of Computer and Communication Sciences, EPFL
Lausanne, Switzerland
mirjana.stojilovic@epfl.ch

Paolo Ienne
School of Computer and Communication Sciences, EPFL
Lausanne, Switzerland
paolo.ienne@epfl.ch

## Abstract

Reconfigurable computing fabrics include FPGAs and CGRAs. FPGAs offer flexible bit-level reconfigurability and can map almost any program via high-level synthesis (HLS) compilers, but they incur high area and speed overheads compared to ASICs. CGRAs, in contrast, provide ASIC-like performance but limited flexibility, typically supporting only feedforward programs with unambiguous memory accesses, far from the capabilities of HLS compilers. This work introduces a new class of reconfigurable arrays inspired by modern dynamically scheduled HLS (DHLS) tools. Unlike traditional HLS, DHLS compilers no longer produce explicit state machines, eliminating the need for look-up tables. Instead, they delegate scheduling decisions to a set of coarse-grained primitives. Our arrays leverage these primitives as processing elements and combine FPGA-style interconnect topology for high routing flexibility with CGRA-like bus-based interconnect. We present a framework to explore these arrays and evaluate a preliminary architecture using DHLS benchmarks. The results show an average of ~2× speed improvement, but unfortunately only a ~20% area reduction compared to an FPGA implemented on the same technology node.

## CCS Concepts

• **Hardware** → **Programmable logic elements**; *Modeling and parameter extraction*; **Hardware-software codesign**.

## Keywords

Reconfigurable Array, High-Level Synthesis, Dynamic Scheduling, Elastic Circuits, CGRA, FPGA

**ACM Reference Format:**
Louis Coulon, Lucas Ramirez, Jason Anderson, Mirjana Stojilović, and Paolo Ienne. 2025. FRIDA: Reconfigurable Arrays for Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25), February 27–March 1, 2025, Monterey, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3706628.3708880



*FPGA '25, February 27–March 1, 2025, Monterey, CA, USA*

ACM ISBN 979-8-4007-1396-5/25/02
https://doi.org/10.1145/3706628.3708880

## 1 Introduction

The most flexible reconfigurable fabrics are field-programmable gate arrays (FPGAs) [6], reconfigurable at the bit level. They are often used with high-level synthesis (HLS) compilers, capable of mapping virtually any program on such fabrics provided that enough hardware resources are available [7, 13, 25, 51]. However, unless an application benefits from bit-level reconfigurability [45], FPGAs typically suffer from a high-performance overhead compared to application-specific integrated circuits (ASICs). Indeed, FPGAs dedicate ~50% of their area to the interconnect [6], and previous research underlined a 17× area and a 3× speed gap against ASICs [29]. Additionally, earlier research efforts aimed at increasing the performance density of FPGAs have offered modest benefits [22, 50], hampering the adoption of FPGAs as computing platforms.

A less flexible but denser category of reconfigurable fabrics are coarse-grained reconfigurable arrays (CGRAs). CGRAs are comprised of coarse processing elements (PEs) connected with bus-based interconnects, allowing them to reach close to ASIC performance [10, 21, 35]. However, such performance comes at the expense of flexibility. Many CGRAs proposed so far do not support control flow and can only execute a single regular loop nest with unambiguous memory addresses [1, 10, 12, 28, 32, 36, 39, 40]; some provide support for irregular loop nests by running outer loops with a tagged dataflow [47] mechanism or rely on advanced clocking schemes [44]. Additionally, while some CGRAs support control flow [17, 20, 21, 35, 43, 52], none would be flexible enough to directly implement any circuit output by a modern HLS compiler.

With this work, we propose to design a new class of reconfigurable arrays, aligning the architecture with the output of modern HLS tools. We rely on dynamically scheduled HLS (DHLS) [26] because, unlike traditional HLS, DHLS compilers no longer produce explicit state machines [15], eliminating the need for look-up tables (LUTs). Instead, they delegate scheduling decisions to a restricted set of coarse-grained primitives, which we leverage to design the processing elements of the arrays. Our arrays have properties similar to CGRAs: processing elements consist of coarse-grained DHLS

primitives, and the interconnect is bus-based. However, our arrays feature an FPGA-style interconnect topology for high routing flexibility. We develop *FRIDA*, an end-to-end **F**ramework for **R**econf**i**gurable **Da**taflow Architecture Exploration, and use it to evaluate a preliminary architecture on a set of DHLS benchmark circuits. The results show an average of ∼2× speed improvement and a ∼20% area reduction compared to an FPGA in the same technology node.

The paper is structured as follows: Section 2 outlines the motivation for our work. Sections 3 and 4 cover the FRIDA compiler and the modeling methodology, respectively. Section 5 presents an architectural instance of the new class of reconfigurable arrays. Section 6 discusses the experimental setup and evaluation results. Finally, Section 7 connects our work to related research.

## 2 Aligning Reconfigurable Arrays with Dynamically Scheduled High-Level Synthesis

We motivate this work by starting with an observation: while FPGAs have many performance drawbacks [22, 29], especially when compared to CGRAs [35], FPGAs remain the most flexible reconfigurable computing platform. HLS compilers can map almost any program to a register-transfer level (RTL) hardware representation suitable for an FPGA implementation. Conversely, CGRAs typically require a specialized toolchain constructed from scratch. Therefore, a large flexibility and performance gap exists between FPGAs and CGRAs. With this work, we are interested in evaluating a particular category of reconfigurable arrays (RAs), which we call *DHLS-RAs*, that exhibit similar flexibility to FPGAs when used with an HLS tool. Towards this end, we adopt a compiler-centric design methodology to retain the high mappability of FPGAs while also attempting to offer the high performance of CGRAs.

CGRA research has often been bottom-up or architecture-centric. Typically, an architecture is proposed, and its performance, power, and area are analyzed with a compiler designed to fulfill the needs of the proposed architecture. As a result, initial CGRA architectures had limited flexibility. For example, ADRES [33] was a time-multiplexed architecture with nearest-neighbor connectivity. It introduced the Modulo Routing Resource Graph (MRRG) abstraction to model the device architecture, which is still used as a basis for many of today's mapping algorithms. However, as mentioned in several prior works [1, 3, 28], such restrictive architectures are challenging for compilers to map to. Consequently, previous research proposed many architectural improvements [1, 21, 28, 35, 39, 44, 47], such as independent switch blocks for routing, and algorithmic relaxations [4, 8, 37, 52] to increase the utilization and performance of CGRAs. A byproduct of such a compilation challenge is that the first CGRA architectures capable of supporting virtually any program have only been investigated recently [21].

### 2.1 Flipping the Conventional Design Approach

In this work, we flip the conventional CGRA design approach and utilize a top-down, compiler-centric strategy, leading to a new set of (flexible) architectures. Similar ideas have been investigated, such as the Reconfigurable Dataflow Arrays [35] compiled from parallel patterns [52]. However, we drive the idea further, starting from the most generic coarse-grained compilation framework: HLS. Specifically, we rely on dynamically scheduled HLS [26]. DHLS is a natural candidate for this work, as it is a form of HLS that no longer synthesizes explicit state machines and delegates scheduling decisions to coarse-grained primitives [15]. We employ DHLS primitives in the design of DHLS-RAs, thereby achieving our compiler-centric design objectives.

Besides DHLS primitives, our DHLS-RA architectures inherit flexible constructs from FPGAs. For example, our architectures incorporate flexible interconnect with many tracks and FPGA-like programmable routing switch blocks [6]. However, wherever possible, the inefficient bit-level FPGA constructs—namely, LUTs and 1-bit interconnect—are replaced with coarse-grained equivalents: coarse-grained PEs and bus-based interconnect. Such flexible constructs allow us to adopt a compilation flow similar to the classic FPGA one, *separately* solving packing, placement, and routing problems, and thus avoiding the challenging unified CGRA mapping problem [10, 33]. The coarse-grained nature of DHLS-RAs implies that computer-aided design (CAD) tools must make fewer decisions, leading to lower CAD runtimes associated with reduced problem sizes and a dramatic reduction in configuration bits.

It is worth noting that this work's objective is not to perform an extensive evaluation of all variants of DHLS-RAs but to perform the initial study of the feasibility of such architectures by (1) developing a generic compiler framework for exploring the design space and (2) providing the reader with an example architecture with reasonably good performance, demonstrating the viability of this new class of CGRAs. In the remainder of this section, we briefly overview DHLS compilers and dive into their regularity, which we exploit to construct the proposed architectures.

### 2.2 Dynamically Scheduled HLS

Dynamically scheduled high-level synthesis refers to the HLS compilers that delegate scheduling decisions to the hardware. They produce elastic circuits, where data signals are bundled with a pair of handshake signals; data exchanges are abstracted as tokens. Elastic circuits are latency insensitive because primitives dynamically exchange tokens only when producers have valid data and consumers are ready. In this paper, we use the open-source DHLS compiler Dynamatic [18, 25, 26] and assume that the elastic circuits adhere to the SELF [15] protocol. Figure 1 shows the main primitives that may appear in a DHLS-generated circuit. Yellow rectangles represent (a subset of) the arithmetic primitives that deal with full-width operands, consuming significant area. However, other primitives, like Fork and CMerge, deal with 1-bit handshake data and consist of a few logic gates. We refer the reader to previous work [26] for a detailed overview.

### 2.3 Regularity of DHLS Circuits

In FPGAs, logic blocks are clusters of primitive elements connected by local interconnect, frequently occurring in application circuits. For example, each LUT is typically paired with one or more flip-flops (FFs) on its output, as application circuits often contain this substructure. The rationale for such clustering and local interconnect is to avoid routing LUT-to-FF signals through the high-delay global FPGA interconnect. Similar considerations arise in DHLS

circuits. Indeed, many of the steering components (e.g., Branch or Fork) are composed of a few gates, and connecting them with a global interconnect akin to FPGAs would incur prohibitive delays. Therefore, analogously to FPGAs, our proposed CGRA architectures are composed of clusters containing multiple DHLS primitives, connected by local interconnect, informed by the commonly occurring patterns of primitives output by DHLS.

Like most compilers, DHLS organizes computation in basic blocks (BBs) and produces deterministic corresponding circuits, as depicted in Figure 2. Specifically, BBs start with a CMerge and multiple Muxes to select the live-in variables, forward them for use in the basic block internal computations, and produce live-out variables through Branches. These Branch components share the same condition and are all driven by a wide Fork replicating the condition token as many times as needed. The condition is usually produced by a Compare unit. Basic blocks can be seen as a source of regularity in DHLS circuits, repeating a typical structure: Branches sharing the same condition feeding into a CMerge and a set of Muxes. Such regularity is an excellent opportunity we aim to explore when designing the PEs of our reconfigurable array (Section 5.2) to avoid prohibitive global interconnect-incurred delay overheads. As we will see, one of the key difficulties will be matching the structure of DHLS circuits with our complex PEs.

## 3 The FRIDA Compiler Framework

Figure 3 shows a high-level diagram of the FRIDA framework, its building blocks, and their interactions. As suggested in the figure, the framework relies on three circuit abstractions, and we progressively lower the circuit and architecture; in the following, we will discuss them and their rationale. Afterward, we will review the key steps of the process from a Dynamatic-generated netlist up to a fully placed and routed design.

### 3.1 Lowering

We use three main abstractions throughout the compiler: elastic channels, which represent data bundles as a single signal; decoupled data and handshake, which have explicit data signals but still keep ready and valid signals together; and, finally, one with all signals explicit (e.g., one 32-bit and two 1-bit signals for an elastic channel).

Initially, we use elastic channels and treat handshake signals implicitly, reflecting Dynamatic's internal representation. We read

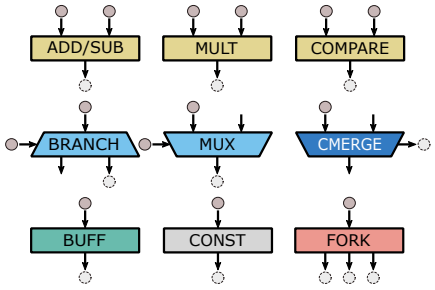


**Figure 1: Primitives commonly found in elastic circuits. Dotted and full circles represent produced and consumed tokens.**

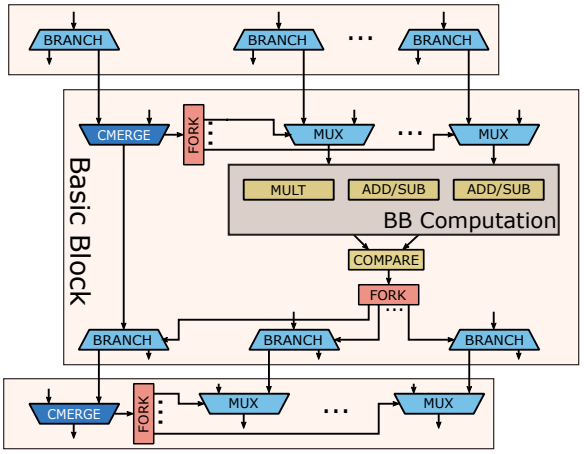


**Figure 2: Simplified DHLS circuit representation of a BB.**

Dynamatic circuits from their multi-level intermediate representation (MLIR) and express the architecture at the same level. Then, we legalize the Dynamatic circuits to the architecture (② in Figure 3) by removing memory primitives and exposing them as I/O since we do not yet support internal memory. We also convert bit widths to either one or 32, corresponding to what our architecture supports: 1-bit wide for control and predicates and 32-bit for data.

Next, we lower the architectures and circuits to a representation with decoupled data and handshake signals. We do this because some elastic primitives, such as the Fork or the Branch, do not have any gates on the data path and only implement fanout. These internal nets can be removed from the primitives since such fanout is easily implemented by an FPGA-like interconnect. Decoupling the data from the handshake provides a finer-grained representation reflecting such needs and avoids, for example, including 32-bit data I/Os on primitives that do not require them. We use this representation for both Buffering and Packing. The packing pass produces a packed netlist, namely, a mapping between application netlist nodes, architecture clusters, and primitives.

Finally, we lower the architecture, circuit, and packing data structures to an explicit representation where all data, valid, and ready signals are explicitly exposed. This representation is suitable for timing annotations targeting any port type and supporting accurate configuration multiplexer insertion, which may sometimes be present only in valid or ready paths. It also represents the exact circuit that is placed and routed (P&R).

### 3.2 Capturing the Architecture in a DSL

The first need of the framework is to capture the desired DHLS-RA architecture, much as FPGA architecture-exploration frameworks, like VPR [34], do. VPR uses an XML-based hierarchical format to describe architectures. The hierarchy consists of a tree of modules analogous to a typical Verilog hierarchy. Each module has an interface that exposes a set of pins, a set of children modules, and an interconnect block specifying how parent and children module pins are interlinked. We want to explore arbitrary clusters in this work, so we developed a Scala-based Domain-Specific Language (DSL) to express architecture hierarchies.

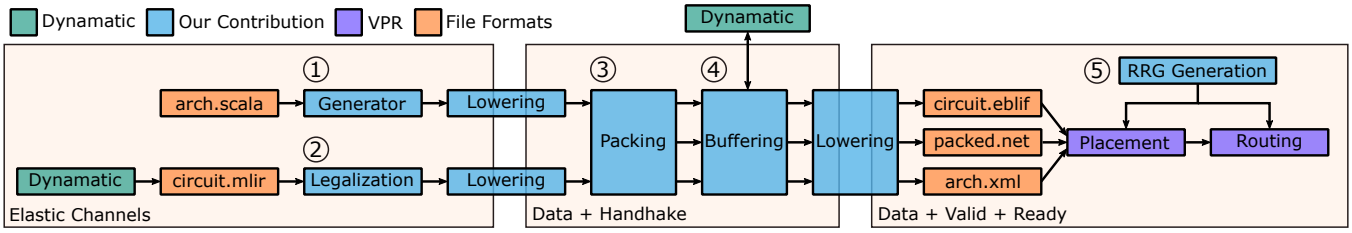


**Figure 3: Compilation flow of the FRIDA framework.**

Our DSL uses three operators that can combine primitives to form any arbitrary cluster hierarchically. We have and (&), which places the primitives in parallel, or (|), which also puts the primitives in parallel but with port sharing, and and-then (→), which chains primitives. Figure 4 shows how these operators can combine two primitives, A and B, each with a single input and output port. We express all the clusters of our arrays (our PEs) using this DSL; the Generator (①) in Figure 3) produces the hierarchical representation from the DSL.

## 3.3 Packing

The packing problem involves finding a covering for the application netlist with patterns corresponding to architecture clusters. In other words, one must decide which clusters and how many to use to implement the complete netlist. While many such coverings are typically possible, we are only interested in the ones that minimize circuit area and critical path.

**Limitations of Current Packing Algorithms.** Packing problems are present for FPGAs [31], where the packer decides which LUTs to group into CLBs, and, more recently, in CGRAs [37], to accommodate increasingly complex CGRA architectures. However, none of the existing packing algorithms scale to the complexity of the clusters investigated in this work (see Section 5.2).

A packing algorithm for FPGAs is incorporated into VPR [31]. With it, primitives with hard-wired connectivity in the architecture (e.g., carry-chain functionality or local links between LUTs and FFs) require the use of *packing patterns* [31]. Packing patterns are annotations on the architecture file enumerating all mappable subgraphs of the architecture that the packing algorithm should seek out in the application netlist. Therefore, each configuration multiplexer has a multiplicative effect on the size of the set of searched architecture subgraphs. Such an exhaustive enumeration scheme does not scale to the large number of configuration multiplexers in our clusters (see Section 5.2).

On the CGRA side, CLUMAP [37] proposes a packing heuristic dedicated to complex CGRA architectures. However, it enumerates all possible placements to validate the candidate mappings, which would be infeasible in our case. Many other CGRA mapping algorithms have been proposed in recent years [4, 8, 10]. Still, most focus on a unified mapping problem encompassing packing, placement, and routing. Therefore, they are not aligned with this work's objectives (Section 2).

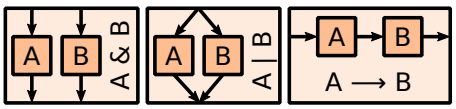


**Figure 4: The &, | and → operators we use to construct our clusters, applied on two primitives, A and B, each with a single input and output.**

**A Custom Packer.** We developed our packer (③ in Figure 3), which is suitable for the architectures we investigate in this work. As with many similar algorithms in technology mapping or instruction selection, we proceed in two phases: the first enumerates (overlapping) mappings associating subgraphs of the application netlist to matching architecture clusters; the second one selects a "good" set of mappings covering the application netlist. However, contrary to similar algorithms, the flexible DHLS-RA clusters contain many configuration multiplexers, resulting in the packing problem having placement and routing subproblems. While a detailed explanation of the algorithm is beyond the scope of this paper, we briefly outline its design below.

In the first phase (matching), we enumerate candidate mappings using decision trees, taking inspiration from the VF2 algorithm [14]. Decision tree nodes correspond to assignments of application netlist nodes to architecture nodes, and a path from a decision tree root to any internal tree node corresponds to a partial mapping. We denote paths from decision tree roots to an arbitrary node $d$ as $M(d)$. We obtain the children of any decision tree node by looking at the circuit and architecture neighborhood and trying all possible combinations. We prune the decision tree, ensuring that any path $M(d)$ results in a valid mapping. We test whether a mapping is valid using an exact ILP formulation inspired by CGRAs [10]. We also prune the decision tree by ensuring that no two paths $M(d_1)$ and $M(d_2)$ result in the same mapping. We build a decision tree for each application netlist node and each architecture node capable of implementing it. We collect the mappings corresponding to all paths from all roots to all leaves of the resulting decision trees.

For the second phase (covering), we use a simple greedy heuristic to select mappings covering the input circuit. We sort the candidate mappings according to their size and select them in decreasing order. When a mapping is selected, we remove the application netlist nodes from all other mappings. If the removal of application netlist nodes invalidates the mapping, we keep it in the list but only consider it for selection once further removals make it valid again.

## 3.4 Buffering

The buffer insertion pass (④ in Figure 3) is critical for performance in DHLS tools. It ensures the correctness of circuits by placing a

buffer in each cycle and impacts performance by appropriately placing buffers to maximize throughput. Buffering in DHLS has been extensively investigated by previous work [27, 41] targeting FPGAs. However, the style of our architectures imposes new constraints on existing algorithms.

First, buffers are too expensive to be present on all paths in the architecture clusters. Consequently, we extended previous buffering algorithms [41] to consider the presence and size of buffers in the architecture. After packing, we run the buffering algorithms to traverse all edges of the packed netlist and look for corresponding buffer slots in the architecture. We constrain buffer depths (at most) to the depth of architecture buffers for edges mapped inside a cluster. On the other hand, we leave edges across clusters unconstrained because additional clusters can be added to implement buffers of any depth, potentially breaking larger buffers into chains of buffers of smaller depths.

Additionally, global routing (that is, connections across clusters) and internal configuration multiplexers contribute significantly to the delay and cannot be ignored when buffering. Thus, we also extended the work of Rizzi et al. [41] to support delay annotations on the input-packed netlist.

### 3.5 Placement and Routing

We rely on VPR 8 for placement and routing, given that the interconnect of our DHLS-RAs is FPGA-like. More precisely, a DHLS-RA interconnect consists of two disjoint FPGA routing networks: one is used to route data signals and consists of 32-bit buses, while the other routes handshake and condition signals and is made of 1-bit wires. We get VPR to generate, separately, the Routing Resource Graphs (RRGs) corresponding to the two disjoint networks. Since they are disjoint, we can merge the two RRGs into one (⑤ in Figure 3) and pass this back to VPR. Unaware of the bit width of the individual resources, VPR sees this as a regular RRG where every routing resource is 1-bit-wide, yet it can correctly place and route the circuit. To reflect the timing differences between 1-bit-wide and 32-bit-wide MUXes, we generate two sets of programmable routing multiplexers with appropriate delay properties.

## 4 Modeling

This section details the process of modeling primitives, interconnects, and tile areas, emphasizing the methods used for synthesis, area estimation, and timing analysis.

### 4.1 Modeling the Primitives

We wrote a library of Chisel [11] generators implementing all DHLS primitives. We use the library to generate a set of nonparameterized Verilog implementations of these primitives. Given a particular technology node, each Verilog implementation goes through logic synthesis into standard cells (Synopsys Design Compiler 2019.12) and P&R (Cadence Innovus 20.1) twice: first, with a zero delay constraint on all timings arcs and, then, with a delay constraint set to the negative slack relaxed by 20%. For each primitive, we extract accurate timing and area information from post-P&R reports. We expose the timing information to VPR with the appropriate timing annotations on the primitive of the architecture description XML [46] and compute cluster areas as described in Section 4.3.

### 4.2 Modeling the Interconnect

We model the *connection block* (CB) and *switch block* (SB) routing multiplexers with standard cells, similar to the other primitives of the architecture. We assume no capacitive output load for CB multiplexers because the tiles do not have local crossbars—that is, CB multiplexers directly drive the inputs of tile components. The length of a wire spanning a single grid location (LEN-1) is estimated as the square root of the area of the largest tile. The capacitive load of the SB multiplexers is calculated as the capacitance of the wire they drive plus an additional 20% accounting for the capacitance of the input pins of the multiplexers at the end of the wire, similar to previous work [24]. We synthesize and P&R a library of multiplexers, varying the number of inputs, the length of the driven wire, and the data width (one or 32 bits).

### 4.3 Modeling Tile Areas

To compute tile areas, we need the areas of the primitives they contain, their corresponding CB and SB multiplexers, and the required configuration storage. The number and size of CB multiplexers are determined from the architectural parameters (number and width of inputs to the tiles). The number and size of SB multiplexers are obtained from the RRG produced by VPR and then averaged across all switch blocks, except those at the perimeter of the array, to account for the variability intrinsic to the VPR RRG generator [5]. The capacitive loads on routing multiplexers determine the size of the SBs. We use an iterative process to set the capacitive load on SB multiplexers. We first estimate the capacitive load using only the tile area without the SB. Then, we recompute the load, considering the tile area with the initial SB size. We repeatedly adjust the load of the SB multiplexers until the area converges. We estimate the area of a configuration bit to be the minimum size FF. Finally, a tile area is the sum of the areas of its primitives, configuration bits, and CB and SB multiplexers.

## 5 Architecture Case Study

Our DHLS-RA architectures are regular arrays of various tiles, approximately of the same area. Each tile is constructed by one or more replicas of a cluster, where clusters are developed from design patterns we observe in the DHLS-generated netlists. In particular, our case study uses three clusters, depicted in Figures 5, 6, and 7. In the following, we outline the strategy we used to build the architecture before diving into the details of the three clusters.

### 5.1 Architectural Design Patterns

Our main design goal when building clusters is to reduce fragmentation: we want to avoid as much as possible that small primitives, such as handshake management ones, end up as the only components mapped to a cluster. As the handshake primitives consist of only a few tens of logic gates, the remaining unused parts of a cluster and CB and SB multiplexers would lead to prohibitive area overheads. Therefore, we design clusters to be as flexible as possible and absorb a wide variety of application subgraphs.

We ensure that most primitives can behave as *identity primitives*, i.e., they can be inserted on any edge of the circuit without changing its functionality. For example, Forks are naturally identity

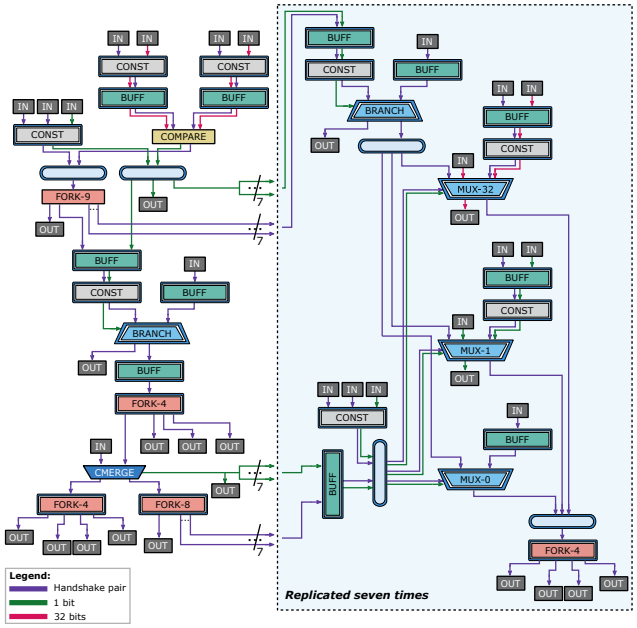


**Figure 5: The Control cluster is the largest cluster. Its structure is similar to the one of Figure 2 with eight basic block live-ins/live-outs and using the design goals of Section 5.1.**

primitives: when they have one output, they transparently forward tokens and can be inserted on any edge. Similar behavior can be obtained from Branches and Muxes by setting the appropriate constant on their condition port. Therefore, we systematically put bypassable constants before Branches and Muxes in the clusters and let the compiler insert identity primitives in the mappings, together with potentially associated constants, whenever necessary.

We overprovision cheap components to prevent, as much as possible, that packing results in clusters used only for small primitives, such as Forks, Sinks, Sources, or Constants. Therefore, we place small, bypassable primitives on every architecture edge they may be present on. In particular, we place Forks before all output handshake ports in the architecture to accommodate values consumed by multiple primitives. Note that data replication—i.e., fanout on 32-bit-wide data signals—is handled by the interconnect, thanks to the separation of the data from the handshake signals.

We try to balance the performance advantages of flexible buffer placement with the desire to have small tile area footprints. Hence, we placed buffers in locations of maximal utility and limited impact on the area. We put the small 0-bit (handshake only) buffers on most edges that can accommodate them. However, we limit the placement of 1-bit and, more specifically, 32-bit buffers to places where they have a reduced area impact, before bigger primitives such as arithmetic-logic units (ALUs) or multipliers, or where they have the highest impact on mapping density.

## 5.2 Constructing the Clusters

The Control cluster, illustrated in Figure 5, is responsible for handshake management and hardens links between consecutive basic blocks, as suggested in Section 2.3. To this end, its architecture contains a Compare and a Fork feeding a condition into a set of Branches, implementing the producer basic block of Figure 2, as well as a CMerge feeding another condition through a Fork to a set of Muxes, implementing the consumer basic block of Figure 2. As explained in the previous section, we provision the tile with ample buffers and bypassable constants. We also make the Branch and Mux primitives bypassable thanks to bypassable constants on their condition input. We set the number of live-outs the tile can accommodate to eight, which means the Control cluster has eight Branches and seven Muxes, sufficient in our experience for most applications. When this is not enough, we note that the Fork after the Compare and the one after the CMerge are of sizes nine and eight, respectively: one of their outputs is exposed to the general interconnect and can reach other tiles. When a primitive can be implemented with multiple bit widths—Muxes, for example—we provide three variants and the configuration multiplexers controlling which one to use. Finally, we place full-width buffers before multiplexers since they are often present in elastic circuits. In Figure 5, we depict bypassable components by enclosing them in a blue frame and denote configuration multiplexers with an oval shape (abstracting their exact location and topology). For clarity, we draw valid and ready signals bundled together and directed as the dataflow even though, in practice, they have opposite directions.

The Ancillary cluster, depicted in Figure 6, is designed as a fallback and cheaper alternative for application netlist subgraphs not efficiently fitting the Control cluster. Its structure is similar to a single instance of the replicated Control cluster column, highlighted in blue in Figure 5: it starts with a Branch connected to Muxes of varying bit widths, driving an output Fork. A CMerge is also available to feed the condition inputs to the Muxes. Contrary to the Control cluster, we size the output Forkto eight, doubling its size, and place

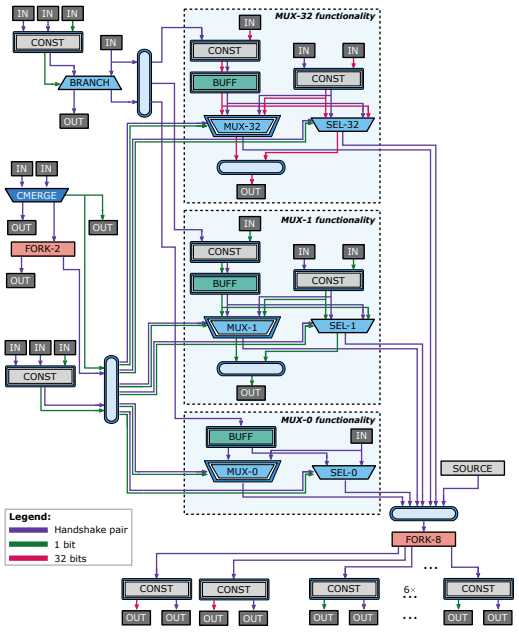


**Figure 6: The Ancillary cluster is a smaller but more flexible version of the Control cluster.**

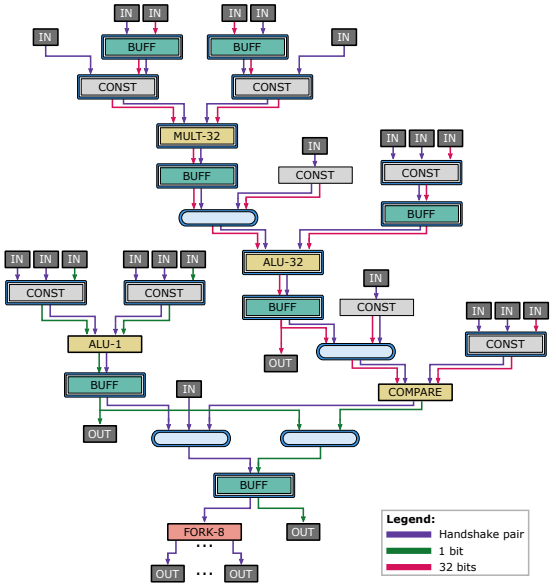


**Figure 7: The Arithmetic Cluster. A 32-bit multiplier followed by a 32-bit ALU or a 1-bit ALU.**

bypassable constants of varying width on the Fork outputs. We also put full-width buffers before one of the Mux inputs, similar to the Control cluster.

The Arithmetic cluster, shown in Figure 7, is responsible for hardening the compute-intensive patterns in elastic circuits. We simplify the structure and harden the classic multiply-accumulate pattern by implementing a multiplier with local links to an ALU. We also allow comparisons after the ALU and provide a small 1-bit ALU. Since the Arithmetic cluster contains large primitives, we place full-width buffers before and after the multiplier and the ALU.

We note that the Ancillary and Arithmetic clusters are universal. They contain all the necessary primitives to implement elastic circuits; primitives are individually reachable and can, therefore, implement together any circuit without the need for the Control cluster. A simple look at the three clusters' figures should clarify the packing phase's challenges (Section 3.3). Namely, packers contain several tens of nodes with fairly irregular connectivity, very rich configuration possibilities, and an almost universal possibility to bypass primitives: all elements that grow exponentially the number of possible configurations and, thus, call for our algorithm, very efficient in pruning the solution space.

## 5.3 Floorplan and Interconnect

Each tile of the DHLS-RA is constructed by replicating the clusters presented in the previous section, trying to match the area of the largest. Concretely, we do not replicate the Control cluster, as it is the largest, but we replicate the Arithmetic cluster twice and the Ancillary cluster three times. Then, we construct DHLS-RAs by tiling $2 \times 2$ patterns containing two Ancillary, one Control, and one Arithmetic tile, as illustrated in Figure 8. We note that, for simplicity, we have not yet implemented embedded memories, and we thus connect memories through the I/O pins.

Figure 8 gives a simplified view into the DHLS-RAs interconnect, focusing on the programmable routing switch blocks (SBs). Connection blocks (CBs) are omitted for clarity. Wires span a single tile (LEN-1) or two (LEN-2). The tile inputs are distributed on their left and bottom sides, while the outputs are on their top and right sides. Ready signals follow the opposite convention to keep them on the same side of the tile as the corresponding valid signal. CBs are cluster-specific, while SBs are the same throughout the array.

The SBs are of the common Wilton type [48]. We empirically size the interconnect parameters to balance between the application routability and the footprint of the SBs and CBs. LEN-1 and LEN-2 wires are uniformly distributed. The 32-bit network's routing channel width is 10 for 320 wires overall. CB multiplexers connect to 50% of the routing wires, and output ports connect to a single routing multiplexer in the neighboring SB. For the 1-bit network, the routing channel width is 60. 1-bit CB multiplexers connect to 25% of the routing wires, and output ports connect to 25% of the routing multiplexers in the neighboring SB.

# 6 Evaluation

We evaluate the sample DHLS-RA architecture on the benchmarks shipped with the open-source Dynamatic DHLS tool [18]. We exclude benchmarks containing operations not currently supported by the case study of Section 5, such as floating point operators and dividers. We generate FPGA results using Dynamatic v2.0.0 with ordinary options and AMD Vivado 2024.1 as a backend. In particular, Dynamatic produces circuits with a minimal bit width for the FPGA, whereas our DHLS-RA always manipulates 32 bits. We target an AMD Kintex-7 FPGA (`xc7k160tfbg484-2`).

For our DHLS-RA, we use the TSMC 28 nm technology and its standard cell library. The AMD 7-series FPGAs [2] are manufactured in the same technology node, allowing for a fair comparison.

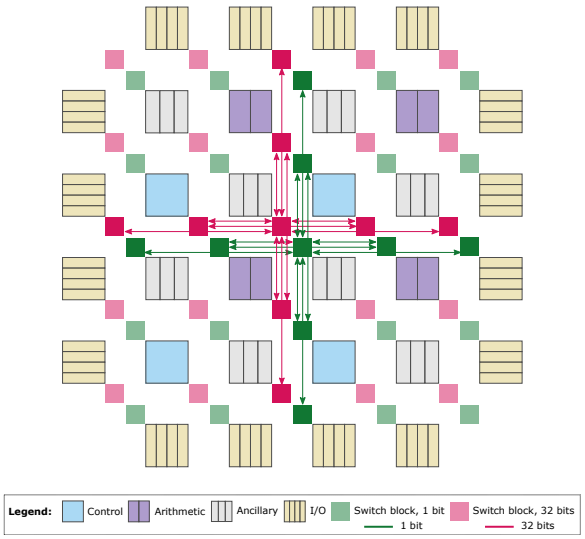


**Figure 8: The DHLS-RA with three tile types in a $2 \times 2$ periodic pattern and a simplified view of the FPGA-style routing with disjoint 1-bit (green) and 32-bit (red) routing networks. Our experiments use an array made of $10 \times 10$ tiles.**

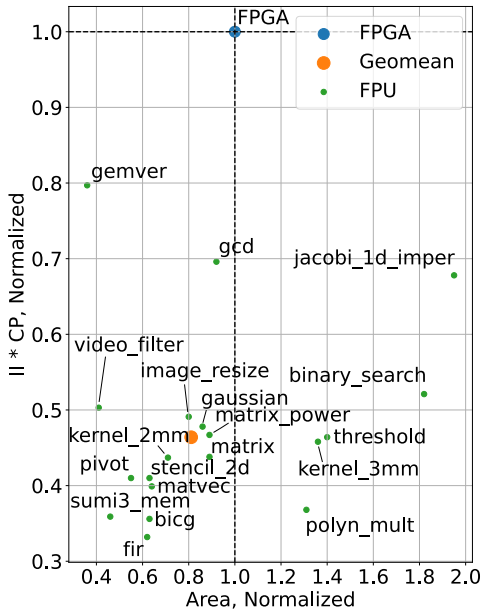


**Figure 9: Performance comparison of benchmarks implemented on the DHLS-RA against FPGA implementations. Results are normalized against the FPGA performance.**

**Table 1: Critical Path and II achieved for all benchmarks.**

| Benchmark | CP (ns) Ours | CP (ns) FPGA | II Ours | II FPGA | II × CP Ratio |
|---|---|---|---|---|---|
| **fir** | 1.72 | 5.16 | 1 | 1 | 0.332 |
| **bicg** | 1.79 | 5.02 | 1 | 1 | 0.356 |
| **sumi3_mem** | 1.78 | 4.98 | 1 | 1 | 0.359 |
| **polyn_mult** | 2.20 | 5.97 | 1 / 1 | 1 / 1 | 0.368 |
| **matvec** | 2.09 | 5.23 | 1 | 1 | 0.399 |
| **pivot** | 2.12 | 5.16 | 1 | 1 | 0.410 |
| **stencil_2d** | 2.21 | 5.38 | 1 | 1 | 0.410 |
| **kernel_2mm** | 2.32 | 5.30 | 1 / 1 | 1 / 1 | 0.437 |
| **matrix** | 2.11 | 4.82 | 1 | 1 | 0.438 |
| **kernel_3mm** | 2.12 | 4.63 | 1 / 1 / 1 | 1 / 1 / 1 | 0.458 |
| **threshold** | 2.04 | 4.40 | 3 | 3 | 0.464 |
| **matrix_power** | 2.15 | 4.60 | 1 | 1 | 0.467 |
| **gaussian** | 2.19 | 4.57 | 1 / 2 | 1 / 2 | 0.478 |
| **image_resize** | 2.09 | 4.26 | 1 | 1 | 0.491 |
| **video_filter** | 2.41 | 4.79 | 1 | 1 | 0.503 |
| **binary_search** | 2.61 | 5.00 | 3 / 3 / 2 | 3 / 3 / 2 | 0.521 |
| **jacobi_1d_imper** | 2.71 | 3.99 | 1 / 1 | 1 / 1 | 0.678 |
| **gcd** | 3.34 | 7.19 | 1 / 3 / 2 | 1 / 2 / 1 | 0.696 |
| **gemver** | 3.32 | 6.26 | 2 / 1 / 2 / 1 | 1 / 1 / 1 / 1 | 0.797 |
| **Geomean** | **2.24** | **5.04** | | | **0.471** |

We characterized FRIDA's components in the technology corner 1.0 V/85 °C/TT that, to our best understanding, corresponds to the FPGA operating conditions [2].

We sweep the critical path objective to minimize all loop kernels' average initiation interval (II) for DHLS-RA and FPGA implementations. Then, we report the area and speed corresponding to the sweep point for each benchmark, minimizing the product of average II and critical path.

Project information can be found on GitHub [19].

## 6.1 FPGA Resource Area Estimation

We estimated the AMD 7-series tile areas, including local and general interconnect, using a publicly available die photo of an Artix-7 FPGA [42]. We captured each tile type's relative width and height from the die photo. The 7-series BRAM tiles contain 36 Kib of SRAM, organized as two 18 Kib subarrays. High-density SRAM cells are reported to occupy 0.127 $\mu m^2$ in TSMC's 28 nm process [49]. To our knowledge, in TSMC 28 nm, the foundry's dual-ported cells are between 1.9× and 2.5× the size of a high-density single-ported 6T cell. Therefore, since 7-series FPGAs use dual-ported BRAMs, we assume that the cell used in the FPGA is around 0.25 $\mu m^2$. This, coupled with the relative dimensions of the tiles, allows estimating the silicon area in $\mu m^2$ for the CLB, DSP, and BRAM tiles.

## 6.2 DHLS-RA Performance

Figure 9 depicts the overall performance of the instance of DHLS-RAs we study, normalized against equivalent FPGA implementations. In both cases, we report the area as the sum of *used* architecture elements. For the FPGA, we report area results computed as the sum of used DSPs and CLBs, including associated SBs. For our work, we report the sum of used clusters (Control, Arithmetic, Ancillary), counting the whole SB area even if only one of the tile's replicated clusters is in use. We achieve a geometric mean critical path × average II and area reduction of about ~2× and ~20%, respectively. These results underline the opportunities provided by DHLS-RAs, exhibiting similar programmability as the FPGA but with higher performance. The results are promising, especially considering the DHLS-RA instance presented in this paper represents our initial attempt. We believe that further design space exploration could lead to architectures with enhanced speed and area efficiency. We now provide a more detailed discussion of the results.

## 6.3 Speed Analysis

Table 1 summarizes the ideal II and critical path achieved on all benchmarks. Each reported II value corresponds to the best II achievable without stalls for choice-free dataflow circuits (CFDFCs) [27], representing control-free loops in the control flow graph. We achieve better critical paths and average II × CP products than the FPGA in all benchmarks. However, we note that the two worse-performing benchmarks, gemver and gcd, achieve higher II than on the FPGA. Since we have similar or better timing properties than the FPGA (our multiplier is combinational, while with Dynamatic, it is pipelined), these results suggest that further tuning of the buffer placement strategy could lead to improved performance.

## 6.4 Area Analysis

Figure 10 shows the breakdown of the tiles area, while Table 3 summarizes the area achieved by all the benchmarks. We achieved a geometric mean area reduction of ~20%, falling short of recovering the FPGAs' 18× area gap against ASICs underlined by Kuon et al. [29]. Two disadvantages of our DHLS-RAs against FPGAs arise from our modeling strategy: (1) our multiplexers (constituting roughly 45% of the total area) are designed in standard cells instead

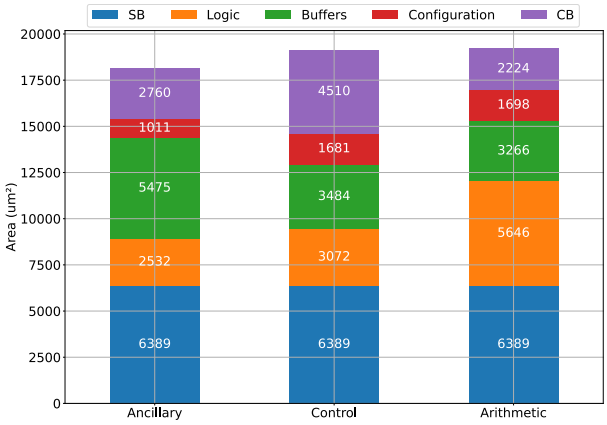


**Figure 10: Tile area breakdown.**

of being laid out by hand and implemented using trees of pass transistors or transmission gates [9], and (2) we synthesize architecture primitives independently, limiting the scope of logic synthesis.

Furthermore, low architecture utilization limits the area gains achievable by the presented DHLS-RA instance. Indeed, contrary to FPGAs, we build our clusters from hardened primitives and achieve generality by making all such primitives bypassable. For example, we supply the tiles with ample buffer resources to provide the buffer placement with enough flexibility to achieve low II values and critical paths. As a result, our mappings only utilize a (small) subset of the primitives available in clusters, whereas in FPGAs, the slice utilization is typically very high. Additionally, the hardened primitives prevent us from applying the bitwidth optimization pass standard in HLS compilers, leading to further area degradation. We quantify the above by looking at two benchmarks with similar FPGA areas but opposite area gains on our fabric: fir, where our fabric is almost half the area of the FPGA, and jacobi_1d_imper, where we are twice the FPGA size.

We report the impact of the bitwidth optimization pass on the first two columns of Table 2. While fir barely changes with and without bitwidth optimization, the implementation of jacobi with unoptimized bitwidth is 2.2× larger than the optimized version, making it 12% larger than our implementation (Table 3). Therefore, the bitwidth optimization pass is responsible for half of the difference in area gains between fir and jacobi. Then, we report architecture utilization in the following columns, computed as the sum of the area of used cluster primitives over the sum of the area of all cluster primitives in used clusters, ignoring, in both cases, any bypass and routing multiplexers. Both benchmarks exhibit significantly lower utilization than LUTs in typical FPGAs: 25% for fir and 15% for Jacobi. In both cases, underutilization originates from the many empty buffer slots, architecture features such as partially

**Table 2: Bitwidth Optimization (BO) and Area Efficiency.**

| Benchmark | Our Area (μm²) | FPGA Area (μm²) | | Area Efficiency | | | |
|---|---|---|---|---|---|---|---|
| | w/o BO | w/ BO | w/o BO | Anc. | Ari. | Con. | Overall |
| **fir** | 61,017 | 97,872 | 103,398 | 4% | 35% | 12% | 25% |
| **jacobi_1d_imper** | 241,078 | 123,414 | 273,537 | 7% | 21% | 16% | 15% |

**Table 3: Area results achieved for all benchmarks.**

| Benchmark | Ours (# Clusters) | | | FPGA | | Area (μm²) | | Area |
|---|---|---|---|---|---|---|---|---|
| | Anc. | Ari. | Con. | Slices | DSPs | Ours | FPGA | Ratio |
| **gemver** | 41 | 18 | 8 | 1676 | 22 | 623,220 | 1,721,004 | 0.36 |
| **video_filter** | 20 | 14 | 2 | 771 | 15 | 344,831 | 831,051 | 0.41 |
| **sumi3_mem** | 4 | 3 | 1 | 133 | 6 | 79,089 | 170,877 | 0.46 |
| **pivot** | 7 | 4 | 2 | 263 | 3 | 147,727 | 266,415 | 0.55 |
| **fir** | 1 | 3 | 1 | 80 | 3 | 61,017 | 97,872 | 0.62 |
| **bicg** | 12 | 5 | 2 | 294 | 7 | 205,276 | 327,222 | 0.63 |
| **stencil_2d** | 12 | 7 | 3 | 350 | 3 | 218,132 | 246,542 | 0.63 |
| **matvec** | 8 | 4 | 2 | 221 | 4 | 151,650 | 235,797 | 0.64 |
| **kernel_2mm** | 37 | 21 | 6 | 801 | 12 | 595,209 | 834,489 | 0.71 |
| **image_resize** | 10 | 5 | 2 | 209 | 2 | 165,912 | 208,617 | 0.80 |
| **gaussian** | 15 | 10 | 3 | 329 | 3 | 280,666 | 327,201 | 0.86 |
| **matrix_power** | 14 | 7 | 2 | 258 | 3 | 232,257 | 261,810 | 0.89 |
| **matrix** | 10 | 7 | 3 | 211 | 6 | 216,590 | 242,715 | 0.89 |
| **gcd** | 36 | 21 | 6 | 703 | 0 | 597,589 | 647,463 | 0.92 |
| **polyn_mult** | 25 | 12 | 4 | 266 | 6 | 383,181 | 293,370 | 1.31 |
| **kernel_3mm** | 45 | 27 | 10 | 567 | 9 | 810,168 | 594,783 | 1.36 |
| **threshold** | 8 | 3 | 3 | 122 | 0 | 157,866 | 112,362 | 1.40 |
| **binary_search** | 33 | 7 | 7 | 259 | 0 | 432,989 | 238,539 | 1.82 |
| **jacobi_1d_imper** | 13 | 8 | 3 | 134 | 0 | 241,078 | 123,414 | 1.95 |
| **Geomean** | | | | | | | | **0.81** |

using the eight Control cluster live-outs (Figure 2), not using the multiplier in the Arithmetic cluster (contributing to a third of its area), or using the Ancillary cluster to map a single primitive. In fir, the single Ancillary cluster contains a single fork. In jacobi, nine ancillaries contain single buffers, and two contain single forks. If these dangling components were absorbed in neighboring tiles, either through nontrivial architecture improvements or a better buffer placement algorithm favoring slots within tiles, jacobi would be at 65% of the area of the FPGA implementation without bitwidth optimization, closing the gap with fir.

## 7 Related Work

Table 4 overviews relevant previous (elastic) CGRA architectures. We underline their programmability in the (supported) *Program Constructs* rows, from the least flexible ones, supporting only *Control-Free* execution, to the most flexible ones, supporting arbitrary DHLS circuits and related features such as automatic pipelining. We also compare the interconnect flexibility of our architecture against previous work. The *PE-Only* row pertains to nearest-neighbor architectures where, for a given PE, it can be used as an interconnect or an ALU operation, not both. The *Standalone SB* row reflects architectures where routing can "pass through" with simultaneous PE use. Across all the architectures in the table, DHLS-RAs stand out as the most flexible architecture. Finally, we list the PE area and the speed results reported in previous CGRA works. In the case of DHLS-RA, we report the area of the largest, arithmetic tile, and the average speed.

To our knowledge, the CGRA framework closest to our work is RipTide [21]. RipTide has a similar goal of offering high programmability, and the authors integrate the necessary primitives within the fabric to run arbitrary programs. However, RipTide targets low-power applications and has a 50 MHz operating frequency on a smaller technology node (Intel 22FFL). Therefore, its intended design space is orthogonal to the one of this paper: our frequencies reported in Table 1 are an order of magnitude higher. Additionally, the authors design their own HLS framework from scratch and propose a new set of operators, including Carry, Invariant, and

**Table 4: Comparison of FRIDA against other CGRA frameworks.**

| | | Riken [1] | (Elastic) HyCube [40] | (Elastic) ADRES [40] | CLUMAP [37] | Plasticine [35, 52] | SPU [16] | UE-CGRA [44] | REVEL [47] | Elastic-CGRA [23] | RipTide [21] | **Ours** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Framework | ✗ | CGRA-ME | CGRA-ME | CGRA-ME | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | FRIDA |
| Program Constructs | Integer (INT) / FP | FP | INT | INT | Both | Both | FP | INT | Both | INT | INT | INT |
| | Control-Free | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Predication | ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| | Nested Loops | ✗ | ✗ | ✗ | ✗ | ✓[1] | ✗ | ✓[2] | ✓[3] | ✓ | ✓ | ✓ |
| | Arbitrary Functions | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓[4] | ✓[5] | ✓ |
| | Arbitrary (D)HLS Circuits | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| Inter-connect | PE-Only | N | N | Y | N | N | N | N | N | N | N | N |
| | Standalone SB | Y | Y | N | Y | Y | Y | Y | Y | N | Y | N |
| | FPGA-Like | N | N | N | N | N | N | N | N | Y | N | Y |
| Perf-ormance | Technology Node (nm) | NandGate 45 | FreePDK 45 | FreePDK 45 | - | TSMC28 | UMC 28 | TSMC28 | 28 | TSMC 65 | Intel 22 FFL | TSMC28 |
| | PE Area ($\mu m^2$) | 27,736 | 9,487 | 6,095 | - | 850,000 | 11,550 | 4,000 | - | 50,000 | 2,035 | 19,223 |
| | CP (ns) | - | 3.07 | 4.66 | - | 1.00 | - | 1.33 | 0.80 | 5.00 | 20.00 | 2.23 |
| | Frequency (MHz) | - | 326 | 215 | - | 1,000 | - | 750 | 1,250 | 200 | 50 | 448 |

[1] Counter Based [2] Support for Branches and Merges [3] Tagged Dataflow for Outer Loops [4] No Loop Pipelining [5] Forced Buffer Placements

Stream, that are not necessary for this work. They do not support automatic pipelining, an essential HLS feature for extracting performance [27, 41], and buffer values on the output channels of PEs, whereas FRIDA integrates with DHLS buffering algorithms [41]. Finally, they adopt mapping algorithms similar to what has been proposed for CGRAs [10], with a unified place and route problem. At the same time, we strive to mimic the FPGA compilation flow with distinct packing, placement, and routing phases.

More broadly, a class of CGRAs similar to this work are elastic CGRAs, where data signals are coupled with handshake signals similar to DHLS circuits. Except for RipTide, all previously proposed elastic CGRAs are feedforward only. The authors of CLUMAP [37] use predication on inner-loop branches to form hyperblocks. In the RIKEN CGRA [3] or in elasticized versions of HyCUBE and ADRES [38, 40], the arrays accept single regular loop nests. Another earlier work [23] presented an elastic CGRA similar to FPGAs. However, the PEs they propose and their compiler are much more naïve than what we investigate in this work and predate most recent DHLS research. Specifically, they require the output of each basic block to pass through a single branch that acts as a synchronizer, preventing loop iterations from being pipelined.

Previous research also emphasized the need for more flexible interconnects in statically scheduled CGRAs. HyCube [28] introduced crossbar switch-blocks and showed 1.5× to 3× improvement in performance per Watt against CGRAs with more restrictive interconnect. TRAM, THRAM [30, 36] and the RIKEN CGRA [3] also demonstrate similar results.

Finally, we compare the performance of DHLS-RAs against prior CGRAs. We note that CGRA's design goals and capabilities are seldom aligned. In Table 4, for example, two opposite design points are RipTide and Plasticine. RipTide has small integer PEs of 2,035 $\mu m^2$ and targets a low 50 MHz operating frequency, while Plasticine has big floating point SIMD PEs of 0.85 $mm^2$ and targets a high 1 GHz operating frequency. Therefore, in the following paragraph, we only give a high-level comparison of the architectures.

ADRES and HyCube (in 45 nm) report PE areas of 6,000 $\mu m^2$ and 9,500 $\mu m^2$, respectively. If (1) we halve the PE area to account for the technology difference and (2) double it because ADRES and HyCube have one ALU per PE while we have two, we see that their PE area is similar to ours if one removes routing resources and buffers (Figure 10). We get similar results with UE-CGRAs reporting an area of 4,000 $\mu m^2$ per PE. Riken is an example of a larger CGRA, with 27,000 $\mu m^2$ per PE in 45 nm, but supports FP. RipTide has a considerably smaller PE (2,035 $\mu m^2$), but at 22 nm, it does not seem fundamentally different. Presumably, all these designs use the same or similar arithmetic libraries (we use Synopsys DesignWare components). Therefore, differences mostly depend on reconfiguration capabilities and the level of generality the designs achieve. Timing-wise, we exhibit similar operating frequencies to most, again, probably because of the similar arithmetic libraries. Unfortunately, almost none of the past work on CGRAs reports II results; yet, a significant amount of our area budget (Figure 10) is to include configurable buffers whose purpose is to achieve the lowest possible II. Finally, some CGRAs (e.g., Plasticine) are so different from our design that it seems impossible to put them in perspective.

## 8 Conclusions and Future Work

In this paper, we proposed a new class of (flexible) elastic CGRAs inspired by DHLS compilers. The tiles of the proposed CGRAs are clusters of hardened primitives—the same primitives that appear in circuits output by a DHLS, aligning the architectures with the compiler output. To explore such architectures and target applications to them, we introduced a framework, FRIDA, composed of front-end DHLS, DSL for architecture specification, custom packer, buffer insertion, and placement and routing with VPR. We presented an instance of such architecture and evaluated its speed and area relative to a commercial FPGA realized in the same process technology. Our evaluation showed average speed and area improvements of ~2× and ~20%, respectively, across a suite of benchmark applications. While the results are encouraging and demonstrate the capability of this new class of CGRAs and our FRIDA compiler, the ~20% area improvements fall short of recovering the FPGAs' 18× area gap against ASICs underlined by Kuon et al. [29]. The limited area gains originate from both modeling and architecture considerations. To an extent, we think many of the problems limiting our area efficiency can be addressed by future work.

## 9 Acknowledgements

This project was partially supported by the Swiss National Science Foundation (grant No. 219299).

## References


[1] Boma Adhi, Carlos Cortes, Yiyu Tan, Takuya Kojima, Artur Podobas, and Kentaro Sano. 2022. The Cost of Flexibility: Embedded versus Discrete Routers in CGRAs for HPC. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Heidelberg, Germany, 347–356. https://doi.org/10.1109/CLUSTER51413.2022.00046

[2] AMD Xilinx. 2021. *Kintex 7 Datasheet*. https://docs.xilinx.com/v/u/en-US/ds182_Kintex_7_Data_Sheet Accessed: 2024-01-13.

[3] H. Jason Anderson, Boma Adhi, Carlos Cortes, Emanuele Del Sozzo, Omar Ragheb, and Kentaro Sano. 2023. Exploration of Compute vs. Interconnect Tradeoffs in CGRAs for HPC. In *Proceedings of the 13th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies* (Kusatsu, Japan) *(HEART '23)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/3597031.3597055

[4] Rami Beidas and Jason H. Anderson. 2022. CGRA Mapping Using Zero-Suppressed Binary Decision Diagrams. In *Proceedings of 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC'22)*. IEEE, Taipei, Taiwan, 616–622. https://doi.org/10.1109/ASP-DAC52403.2022.9712571

[5] Vaughn Betz and Jonathan Rose. 2000. Automatic Generation of FPGA Routing Architectures from High-Level Descriptions. In *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '00)*. Association for Computing Machinery, New York, NY, USA, 175–184. https://doi.org/10.1145/329166.329203

[6] Andrew Boutros and Vaughn Betz. 2021. FPGA Architecture: Principles and Progression. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 4–29. https://doi.org/10.1109/MCAS.2021.3071607

[7] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[8] Liang Chen and Tulika Mitra. 2012. Graph Minor Approach for Application Mapping on CGRAs. In *Proceedings of 2012 International Conference on Field-Programmable Technology (FPT'12)*. IEEE, Seoul, Korea (South), 285–292. https://doi.org/10.1109/FPT.2012.6412149

[9] Charles Chiasson and Vaughn Betz. 2013. Should FPGAs Abandon the Pass-Gate?. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications (FPL'13)*. IEEE, Porto, Portugal, 1–8. https://doi.org/10.1109/FPL.2013.6645511

[10] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A Unified Framework for CGRA Modelling and Exploration. In *Proceedings of 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP'17)*. IEEE, Seattle, USA, 184–189. https://doi.org/10.1109/ASAP.2017.7995277

[11] ChipsAlliance. 2024. *Chisel Language*. https://www.chisel-lang.org/ Accessed: 2024-10-08.

[12] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A Fully Pipelined and Dynamically Composable Architecture of CGRA. In *Proceedings of 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. IEEE, Boston, MA, USA, 9–16. https://doi.org/10.1109/FCCM.2014.12

[13] Jason Cong and Zhiru Zhang. 2006. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *Proceedings of the 43rd Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '06)*. Association for Computing Machinery, New York, NY, USA, 433–438. https://doi.org/10.1145/1146909.1147025

[14] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub)graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[15] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of Synchronous Elastic Architectures. In *Proceedings of the 43rd Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '06)*. Association for Computing Machinery, New York, NY, USA, 657–662. https://doi.org/10.1145/1146909.1147077

[16] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[17] Jinyi Deng, Xinru Tang, Jiahao Zhang, Yuxuan Li, Linyun Zhang, Boxiao Han, Hongjun He, Fengbin Tu, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. 2023. Towards Efficient Control Flow Handling in Spatial Architecture via Architecting the Control Flow Plane. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1395–1408. https://doi.org/10.1145/3613424.3614246

[18] EPFL-LAP. 2024. *Dynamatic Compiler*. https://github.com/EPFL-LAP/dynamatic Accessed: 2024-10-03.

[19] EPFL-LAP. 2024. *FRIDA Compiler*. https://github.com/EPFL-LAP/FRIDA Accessed: 2024-18-12.

[20] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: an Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE, Valencia, Spain, 1027–1040. https://doi.org/10.1109/ISCA52012.2021.00084

[21] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *Proceedings of 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*. IEEE, Chicago, IL, USA, 546–564. https://doi.org/10.1109/MICRO56248.2022.00046

[22] David Grant, Chris Wang, and Guy G.F. Lemieux. 2011. A CAD Framework for Malibu: An FPGA with Time-Multiplexed Coarse-Grained Elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/1950413.1950441

[23] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 171–180. https://doi.org/10.1145/2435264.2435296

[24] Safeen Huda, Jason Anderson, and Hirotaka Tamura. 2014. Optimizing Effective Interconnect Capacitance for FPGA Power Reduction. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '14)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/2554688.2554788

[25] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/3174243.3174264

[26] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2021. Synthesizing General-Purpose Code Into Dynamically Scheduled Circuits. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 97–118. https://doi.org/10.1109/MCAS.2021.3071631

[27] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 186–196. https://doi.org/10.1145/3373087.3375314

[28] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference (DAC'17)* (New York, NY, USA). Association for Computing Machinery, Austin, TX, USA, 1–6. https://doi.org/10.1145/3061639.3062262

[29] Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2007), 203–215. https://doi.org/10.1109/TCAD.2006.884574

[30] Jingyuan Li, Yunhui Qiu, Guowei Zhu, Qilong Zhu, Wenbo Yin, and Lingli Wang. 2023. THRAM: A Template-based Heterogeneous CGRA Modeling Framework Supporting Fast DSE. In *Proceedings of 2023 IEEE International Symposium on Circuits and Systems (ISCAS'23)* (Monterey, CA, USA). IEEE, Monterey, CA, USA, 1–5. https://doi.org/10.1109/ISCAS46773.2023.10182204

[31] Jason Luu. 2014. *Architecture-Aware Packing and CAD Infrastructure for Field-Programmable Gate Arrays*. Ph.D. Thesis. University of Toronto, Toronto, Canada.

[32] Bingfeng Mei, Mladen Bereković, and Jean-Yves Mignolet. 2007. ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors. In *Fine- and Coarse-Grain Reconfigurable Computing*, Stamatis Vassiliadis and Dimitrios Soudris (Eds.). Springer, Dordrecht, Netherlands, 255–297. https://doi.org/10.1007/978-1-4020-6505-7_6

[33] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL'03)*. Springer, Berlin, Heidelberg, Germany, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[34] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. 2020. VTR 8: High-performance CAD and Customizable FPGA Architecture Modelling. *ACM Transactions on Reconfigurable Technology and Systems* 13, 2, Article 9 (June 2020), 55 pages. https://doi.org/10.1145/3388617

[35] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings*

*of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Association for Computing Machinery, Toronto ON Canada, 389–402. https://doi.org/10.1145/3079856.3080256

[36] Yunhui Qiu, Yuhang Cao, Yuan Dai, Wenbo Yin, and Lingli Wang. 2022. TRAM: An Open-Source Template-Based Reconfigurable Architecture Modeling Framework. In *Proceedings of 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL'22)*. IEEE, Belfast, United Kingdom, 61–69. https://doi.org/10.1109/FPL57034.2022.00021

[37] Omar Ragheb and Jason H. Anderson. 2024. CLUMAP: Clustered Mapper for CGRAs with Predication. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 101, 6 pages. https://doi.org/10.1145/3649329.3658269

[38] Omar Ragheb, Rami Beidas, and Jason Anderson. 2023. Statically Scheduled vs. Elastic CGRA Architectures: Impact on Mapping Feasibility. In *Proceedings odf 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'23)*. IEEE, St. Petersburg, FL, USA, 468–475. https://doi.org/10.1109/IPDPSW59300.2023.00079

[39] Omar Ragheb, Stephen Wicklund, Matthew Walker, Rami Beidas, Adham Ragab, Tianyi Yu, and Jason Anderson. 2024. CGRA-ME 2.0: A Research Framework for Next-Generation CGRA Architectures and CAD. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'24)*. IEEE, San Francisco, CA, United States, 642–649. https://doi.org/10.1109/IPDPSW63119.2024.00124

[40] Omar Ragheb, Tianyi Yu, David Ma, and Jason Anderson. 2022. Modeling and Exploration of Elastic CGRAs. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL'22)*. IEEE, Belfast, United Kingdom, 404–410. https://doi.org/10.1109/FPL57034.2022.00067

[41] Carmine Rizzi, Andrea Guerrieri, Paolo Ienne, and Lana Josipović. 2022. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications (FPL'22)*. IEEE, Belfast, United Kingdom, 375–383. https://doi.org/10.1109/FPL57034.2022.00063

[42] Siliconpr0n Website. 2024. *AMD Artix-7 FPGA: Die Photo*. https://siliconpr0n.org/map/xilinx/xc7a50t/single/xilinx_xc7a50t_mcmaster_s1-2_vc60x_roi1.jpg Accessed: 2024-01-15.

[43] Steven Swanson, Ken Michelson, Andrew Schwerin, and Andrew Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. (MICRO'03)*. IEEE, San Diego, CA, USA, 291–302. https://doi.org/10.1109/MICRO.2003.1253203

[44] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-Elastic CGRAs for Irregular Loop Specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*. IEEE, Seoul, Korea (South), 412–425. https://doi.org/10.1109/HPCA51647.2021.00042

[45] Yaman Umuroglu, Yash Akhauri, Nicholas James Fraser, and Michaela Blott. 2020. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In *Proceedings of 2020 30th International Conference on Field-Programmable Logic and Applications (FPL'20)*. IEEE, Gothenburg, Sweden, 291–297. https://doi.org/10.1109/FPL50879.2020.00055

[46] VPR Website. 2024. *VPR Architecture Description Format*. University of Toronto. https://docs.verilogtorouting.org/en/latest/arch/ Accessed: 2024-10-03.

[47] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. IEEE, San Diego, CA, USA, 703–716. https://doi.org/10.1109/HPCA47549.2020.00063

[48] Steven J. E. Wilton. 1997. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. Ph. D. Dissertation. University of Toronto.

[49] Shien-Yang Wu, J.J. Liaw, C.Y. Lin, M.C. Chiang, C.K. Yang, J.Y. Cheng, M.H. Tsai, M.Y. Liu, P.H. Wu, C.H. Chang, L.C. Hu, C.I. Lin, H.F. Chen, S.Y. Chang, S.H. Wang, P.Y. Tong, Y.L. Hsieh, K.H. Pan, C.H. Hsieh, C.H. Chen, C.H. Yao, C.C. Chen, T.L. Lee, C.W. Chang, H.J. Lin, S.C. Chen, J.H. Shieh, M.H. Tsai, S.M. Jang, K.S. Chen, Y. Ku, Y.C See, and W.J. Lo. 2009. A Highly Manufacturable 28nm CMOS Low Power Platform Technology with fully Functional 64Mb SRAM using Dual/Tripe Gate Oxide Process. In *2009 Symposium on VLSI Technology*. IEEE, Kyoto, Japan, 210–211.

[50] A. Ye and J. Rose. 2006. Using Bus-Based Connections to Improve Field-Programmable Gate-Array Density for Implementing Datapath Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14, 5 (2006), 462–473. https://doi.org/10.1109/TVLSI.2006.876095

[51] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA'22)*. IEEE, Seoul, Korea (South), 741–755. https://doi.org/10.1109/HPCA53966.2022.00060

[52] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE, Valencia, Spain, 1041–1054. https://doi.org/10.1109/ISCA52012.2021.00085