

The Design, Integration, and Programming of an Embedded FPGA for Edge AI

Biruk Seyoum[‡]  *Member, IEEE*, Allen Boston[‡]  *Member, IEEE*, Carmine Rizzi , Kuan-Lin Chiu , Guy Eichler 
Joseph Zuckerman , Lana Josipović  *Member, IEEE*, Pierre-Emmanuel Gaillardon  *Senior Member, IEEE*,
Luca P. Carloni  *Fellow, IEEE*

Abstract—The deployment of AI at the edge demands energy-efficient, high-performance, and adaptable computing under strict area and power budgets. Embedded FPGAs (eFPGAs) are a promising platform because they integrate reconfigurable logic directly into ASIC system-on-chip (SoC) designs. However, making eFPGAs practical for AI requires an architectural enhancement that boosts compute capability while remaining compatible with tight power, performance, and area (PPA) constraints. We present FPSA (field-programmable systolic array), a novel eFPGA-native AI primitive whose hierarchical organization reduces interconnect pressure, localizes data movement, and improves utilization. Building on FPSA, we develop an end-to-end framework that spans custom eFPGA fabric generation, HLS-based C/C++ workload mapping, and heterogeneous SoC integration, enabling system-level deployment of AI acceleration on eFPGA-based SoCs. At the primitive level, FPSA achieves $2.6\times$ higher compute density and $1.6\times$ higher frequency than state-of-the-art FPGA systolic primitives. At the system level, case studies in brain-computer interface (BCI) and high-energy physics (HEP) domains show that FPSA-enhanced eFPGAs reduce power and area per channel by up to $2.64\times$ and $4\times$, respectively, over a baseline eFPGA in a BCI SoC, and achieve $2.4\times$ higher throughput than a fixed-function ASIC baseline in a heterogeneous HEP SoC. A vision transformer case study further demonstrates FPSA's generality, achieving up to $135.6\times$ area reduction, $24.8\times$ power reduction, and $7.5\times$ higher throughput compared to a general-purpose eFPGA baseline.

Index Terms—Embedded FPGA (eFPGA), Reconfigurable Computing, Systolic array, Field-programmable systolic array (FPSA), High-level synthesis (HLS), System-on-chip (SoC)

I. INTRODUCTION

Embedded Edge-AI systems that operate under tight power and area constraints ($1\text{--}5\text{ W}$ and $<10\text{ mm}^2$) [1] must execute demanding linear-algebra workloads with high throughput and low latency, while remaining adaptable to frequently evolving models after deployment. To meet these demands, designers commonly use specialized hardware such as ASICs, FPGAs, and CGRAs. ASICs deliver high performance and energy efficiency, but incur high non-recurring engineering (NRE)

costs and offer limited flexibility [2]. Reconfigurable platforms improve adaptability, but with efficiency tradeoffs: FPGAs offer bit-level reconfigurability but suffer from logic and routing overheads on compute-intensive AI workloads [2]–[4], while CGRAs improve compute density through coarse-grained processing elements (PEs) but introduce mapping, scheduling, and communication overheads [5], [6]. These tradeoffs motivate a reconfigurable Edge-AI architecture that balances reconfigurability with the performance and energy efficiency of ASICs.

Embedded FPGAs (eFPGAs) represent a promising alternative: they integrate FPGA-like reconfigurable logic into selected regions of an ASIC SoC, enabling post-deployment flexibility without the NRE cost of full redesign [7]–[10]. However, unlike standalone FPGAs, eFPGAs operate with much tighter logic, routing, and on-chip bandwidth budgets [8], [10]. As a result, prior eFPGA deployments focused on relatively lightweight tasks such as sensor front-ends [8], lightweight co-processors [11], and CPU ISA extensions [10], [12]. Scaling to compute-intensive Edge-AI workloads remains difficult because existing eFPGA fabrics lack AI-oriented architectural support and end-to-end programming flows that fit their strict power, performance, and area (PPA) constraints.

Although prior reconfigurable-computing research has improved AI efficiency and compute density, most such advances target standalone FPGAs and do not directly translate to eFPGA fabrics. Commercial platforms often pair FPGA fabric with coarse-grained AI engines [13], but this loose coupling incurs data-movement overheads [14] and the resulting footprint is too large for PPA-constrained embedded edge SoCs. Other approaches embed AI-oriented fine-grained building blocks (*primitives*) inside the fabric [15]–[17], but these are largely proprietary and incompatible with open eFPGA flows. Academic work on standalone FPGAs has explored systolic-array-based primitives [3], [4], [18], but most of these designs are generally tuned for the abundant logic, routing, and memory bandwidth of standalone FPGAs. The key gap is therefore not the absence of AI acceleration techniques, but the lack of eFPGA-native AI architectures and end-to-end flows that enable practical deployment within strict PPA limits.

Our key insight is that large monolithic AI primitives are a poor match for resource-constrained eFPGA fabrics because their interface and execution model do not scale gracefully under tight routing and area budgets. In the case of monolithic systolic-array primitives, such as those in [3], [4], this mismatch appears in two ways: First, as array size grows, the

[‡]These authors contributed equally to this work.

B. Seyoum, K.-L. Chiu, G. Eichler, J. Zuckerman, and L. P. Carloni are with the Computer Science Department, Columbia University (email: biruk, chiu, gyeichler, jzuck, luca @cs.columbia.edu). C. Rizzi and L. Josipović are with the Department of Information Technology and Electrical Engineering, ETH Zurich (email: crizzi, josipovic @ethz.ch). A. Boston and P.-E. Gaillardon are with the Electrical and Computer Engineering Department, University of Utah (email allen.boston, pierre-emmanuel.gaillardon @utah.edu).

This work was supported in part by the Department of Energy under Award DE-SC0024458, in part by IBM Corporation, and in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

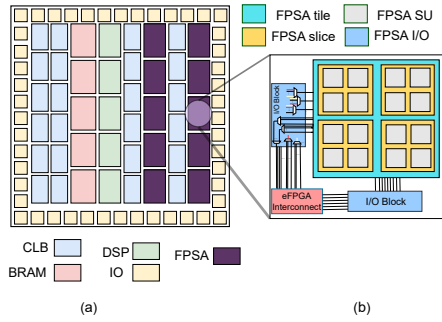


Fig. 1: (a) High-level overview of our AI-optimized eFPGA fabric, (b) high-level organization of the FPSA primitive.

primitive-to-interconnect interface scales poorly: the number of boundary signals increases, creating routing pressure at the primitive boundary and making data delivery increasingly difficult [10], [14], [19], [20]. Second, exposing the array as a single fixed compute unit leads to under-utilization on small, irregular, or mixed workloads [3], [4].

To address these challenges, we introduce the *field-programmable systolic array* (FPSA), a reconfigurable systolic compute primitive designed specifically for eFPGA integration. Fig. 1a shows an eFPGA fabric enriched with our FPSA primitives. The FPSA architecture, shown in Fig. 1b, replaces a monolithic systolic organization with a hierarchical structure of tiles, slices, and systolic units (SUs) that decomposes the array into smaller units. This hierarchy reduces primitive-interconnect I/O pressure by localizing operand distribution within the tile and enables reconfigurable intra-tile dataflow for concurrent execution on independent data units. FPSA also provides a latency-insensitive interface [21], [22] that simplifies boundary control and coordination, reducing integration overhead at the primitive boundary. Architecturally, FPSA occupies a middle-ground between fine-grained FPGA and CGRA-style word-level compute: it provides more specialized support for tensor computation than general FPGA primitives, while preserving the configurability and in-fabric integration required by eFPGAs.

Building on our FPSA primitives, we present a complete open-source framework for Edge-AI acceleration on eFPGA fabrics, with architectural innovations that enable customized eFPGAs to be integrated, programmed, and controlled as first-class components of heterogeneous SoCs alongside processors, memory, I/O, and fixed-function accelerators. To make these fabrics programmable, we develop an HLS-based mapping methodology for structured C/C++ linear-algebra kernels targeting our FPSA-augmented fabrics. This end-to-end framework enables full-system evaluation of FPSA in heterogeneous SoCs running a diverse set of real workloads that span brain-computer interface (BCI), high-energy physics (HEP), and a widely deployed vision transformer application [23].

At the primitive level, FPSA achieves $2.6\times$ higher compute density than a state-of-the-art FPGA systolic primitive [3]. These gains translate into system-level benefits: FPSA-enhanced eFPGAs reduce power and area per channel by up to $2.64\times$ and $4\times$ in BCI workloads, and improve energy efficiency by $4.2\times$ over a baseline eFPGA in a complete

HEP SoC. Furthermore, the AI-optimized eFPGAs provide up to $135.6\times$ and $24.8\times$ area and power reductions and $7.5\times$ throughput improvement against baseline for transformer-based AI workloads.

Our key contributions are as follows:

- A novel eFPGA-native AI primitive, FPSA, with a hierarchical tile/slice/SU organization that reduces interconnect pressure and improves utilization (Section III).
- An HLS-based compilation flow that maps structured C/C++ linear-algebra kernels onto FPSA-augmented eFPGA fabrics (Section IV).
- An end-to-end framework spanning FPSA primitive design, custom eFPGA fabric generation, HLS-based workload mapping, and SoC integration, enabling system-level deployment of AI acceleration in heterogeneous reconfigurable SoCs (Section V).

II. BACKGROUND AND MOTIVATION

In this section, we discuss the key differences between eFPGAs and standalone FPGAs, and motivate our work.

A. Standalone FPGAs vs. eFPGAs

Standalone FPGAs and eFPGAs both use reconfigurable logic but differ in architecture and deployment. A standalone FPGA is a full SoC with its own configuration memory, I/O, and power management, designed for general-purpose acceleration. The fabric is composed of heterogeneous primitives such as logic blocks with look-up table (LUT) arrays, on-chip block RAMs (BRAMs), DSP slices, and, in some cases, high-bandwidth memory interfaces such as DDR or HBM [13], [24]. Furthermore, the primitives in FPGAs support advanced features; for example, BRAMs include dedicated circuitry to flexibly configure different word widths and depths [20], [24].

In contrast, an eFPGA is an *intellectual property* (IP) block embedded within a larger ASIC SoC, relying on the host for interconnect, memory, and often clock and power infrastructure. To remain lightweight and integration-friendly, eFPGAs provide a smaller set of simpler primitives than standalone FPGAs [10]. Their most important architectural limitation is the routing fabric: whereas standalone FPGAs devote 50–80% of silicon area to dense, high-performance custom interconnect [10], [20], [25], [26], eFPGAs use less flexible standard-cell-based routing. This improves portability and IP integration, but reduces performance and constrains the design of specialized fine-grained primitives, requiring greater awareness of the fabric’s routing limitations.

B. Systolic Array Computation on eFPGAs

Systolic arrays (SAs) use parallel PEs to perform multiply-and-accumulate (MAC) operations while forwarding operands across the array each cycle, composing into higher-level kernels such as general matrix multiplication (GEMM) and matrix-vector multiplication (MVM) [27], [28]. Although eFPGAs can support SA computation, mapping SAs efficiently onto FPGA fabrics exposes fundamental limitations. Even small arrays require substantial control and data-orchestration

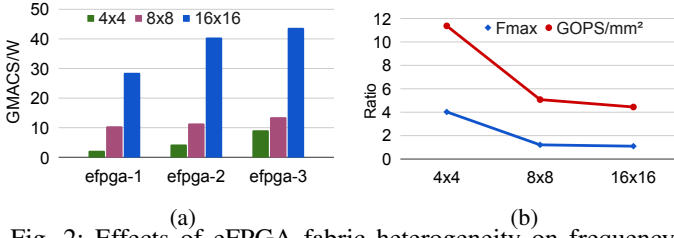


Fig. 2: Effects of eFPGA fabric heterogeneity on frequency, area, and power for systolic GEMM implementations.

logic, which quickly becomes a performance bottleneck [17], [27]; larger arrays further intensify routing congestion and buffering overheads [3]. These challenges are especially severe in the resource-constrained eFPGAs, as illustrated below.

Analysis. To quantify these limitations, we implement three systolic-array accelerators (4×4 , 8×8 , and 16×16 PEs, INT16 precision) using OpenFPGA [19] on three eFPGA fabrics of increasing heterogeneity: LUT-only (efpga_1), LUT+BRAM (efpga_2), and LUT+BRAM+DSP (efpga_3). Fig. 2a shows that *power efficiency* (operations per watt, $\frac{GMACS}{W}$) improves with fabric heterogeneity across all SA sizes, consistent with prior findings on standalone FPGAs [20]. Fig. 2b shows the improvement ratio between efpga_3 and efpga_1 with respect to SA size: heterogeneity improves F_{max} by up to $4\times$ for the 4×4 SA but only $1.07\times$ for the 16×16 SA, and improves *compute density* (measured in $\frac{GMACS}{sec \cdot mm^2}$) by $11.3\times$ at 4×4 , dropping to $5\times$ at 8×8 and $4\times$ at 16×16 , a superlinear area penalty. These results show that heterogeneity alone cannot sustain scaling: as array size grows, routing and data-delivery overheads dominate, making even modest SA sizes (e.g., 16×16) difficult to support efficiently in eFPGAs.

C. High-performance eFPGAs for Edge AI

Making eFPGAs practical for Edge AI requires overcoming three challenges. First, compute density and energy efficiency of SA computations must be improved within strict PPA constraints: off-fabric SA engines (e.g., Versal-style [13]) incur communication and area overheads incompatible with edge budgets, while existing fine-grained hard macro approaches targeting standalone FPGAs [3], [17] are too large or inefficient to translate well to eFPGA fabrics, as we show experimentally in Section VI-B. Second, efficiently mapping applications onto AI-optimized eFPGAs requires methodologies that can handle the complex control and dataflow of systolic architectures [27], [29], a capability that existing primitives and tools lack. Third, the full design flow from primitive design and fabric integration to SoC embedding is fragmented and hard to reproduce, limiting practical deployment. Our work addresses all three challenges through FPSA and its supporting end-to-end flow.

III. FPSA MICROARCHITECTURE

Designing a new eFPGA primitive requires a proper balance across area, performance, energy efficiency, flexibility, I/O bandwidth, and routability, all within the tight PPA constraints of the designated application domain [20], [25]. Our FPSA architecture navigates these tradeoffs for the eFPGA context.

A. Composable Systolic Hierarchy

Fig. 1 illustrates the logical layout of a heterogeneous eFPGA fabric featuring **FPSA tiles** alongside CLBs, BRAMs, and DSPs. Arranged in columns at the granularity of a BRAM or DSP block, each tile adopts a hierarchical organization: a 2D array of **FPSA slices**, each integrating a 2D array of **FPSA systolic units (SUs)** that form the core PEs.

Prior work typically relies on monolithic systolic compute units without an explicit hierarchy between PEs [3], [4]. Consequently, when target workloads are smaller than the physical SA size within the tile, PE utilization collapses because many PEs sit idle (e.g., executing a 4×4 GEMM on a tile with 8×8 array leaves $3/4$ PEs idle). In contrast, the FPSA slice hierarchy groups PEs into independently operable slices (Fig. 1b), enabling local operand distribution across PE groups and concurrent execution of multiple independent data paths of the same operation type, yielding SIMD-style execution within the systolic array. Increasing the number of PEs in a monolithic SA raises the I/O pin count between the primitive and the eFPGA interconnect logic (blue box in Fig. 1b) [14], creating routing pressure that can either cause congestion or force an area- and power-inefficient routing architecture [14], [25]. FPSA addresses this differently: rather than relying on the eFPGA routing fabric to distribute operands among PEs, it internalizes this distribution within the tile through reconfigurable intra-tile routing. This shifts the data-delivery burden away from the surrounding routing fabric, keeping the external interface lean while simplifying integration. These mechanisms are further detailed in Section III-B. At every level of the hierarchy, the FPSA architecture provides buffering, control logic, and data orchestration for better performance when composing primitives to execute systolic tasks.

FPSA Systolic Units (SUs). FPSA SUs are the fundamental PEs responsible for the execution of MAC operations. In a systolic-based GEMM, each PE performs an output-stationary MAC on one row and one column. Fig. 3a shows the SU datapath and its control logic. The SU receives input values through *west* and *north* and has a controller whose inputs are the matrix/vector size, the tile coordinates, and the *op_mode* signal to dynamically configure the operation of its datapath.

Each SU accepts two 16-bit input values (west, north) and can operate in INT16 or INT8 mode. This is enabled by a *fracturable multiplier* [20] that performs a single INT16 multiplication or two INT8 multiplications in parallel [30], with the mode selected at runtime through the *op_mode* signal. Products are accumulated into a 38-bit register. Floating-point (FP) support is excluded from the current design, as associated area and power overheads are prohibitive within the tight PPA budgets of eFPGAs, and most Edge AI workloads primarily rely on low-precision integer arithmetic [31].

FPSA Slices. An FPSA slice wraps an $N \times N$ array of SUs with integrated I/O and control logic. Fig. 3b shows a 2×2 FPSA slice. The input ports *north_in* and *west_in* feed the first row and column of the SU grid. The slice output logic correctly reorders and shifts SU outputs across different modes of operation. The configurable slice controller manages the muxing signals to flexibly support data transfer in dif-

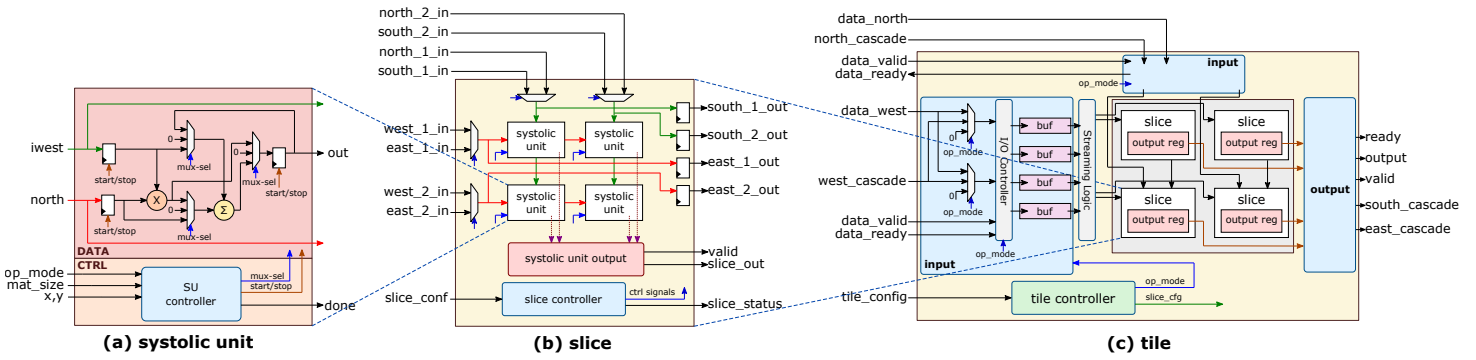


Fig. 3: The detailed microarchitecture of FPSA. (a) the FPSA SU, (b) the FPSA slice, and (c) the FPSA tile.

ferent op_mode . By encapsulating control, data orchestration, and buffering at the slice level, FPSA reduces per-SU logic overhead, thus improving energy efficiency.

FPSA Tiles. FPSA tiles are coarse-grained structures within the FPGA fabric that are programmed directly to accelerate computations. Each tile contains a $K \times K$ array of slices along with an I/O controller, streaming logic, and line-buffers.

Fig. 3c shows a typical tile. Here, N and K denote the design-time parameters for the number of SUs per slice and slices per tile, respectively, so each tile contains $N^2 \times K^2$ SUs. For example, in Fig. 3c, $N = K = 2$. Each tile has two 64-bit inputs (to receive data from outside of the primitive), one 76-bit output (equivalent to 2 SU outputs), a control/configuration interface, and I/O ports for cascading tiles. Given the limited on-chip memory resources (BRAMs) in eFPGAs, we integrate on-primitive line buffers to enable input reuse. For example, in the case of GEMM computations, a row of vectors can be read once and reused to multiply it with many columns, thus reducing off-chip traffic. Additionally, line buffers help improve the I/O bandwidth, as discussed in Section III-D.

B. Benefits of the Hierarchical Architecture

The FPSA hierarchical architecture introduces two key benefits. First, by leveraging configurable intra-tile routing (Fig.3c), it expands the number of internal SUs that can receive independent operands with minimal I/O cost between the primitive and the interconnect. In conventional SAs with monolithic organization [3], [4], [28], [32], only the top- and left-most SUs receive external inputs, and directly feeding additional SUs (PEs) would cause I/O count to grow rapidly with the array size. This poor I/O scalability complicates routing logic and is particularly undesirable in resource-constrained eFPGAs. In contrast, the FPSA architecture combines tile-level I/O logic (Fig.3c) with the slice hierarchy to locally distribute operands among slices, achieving linear I/O growth. This has two advantages: (1) it reduces the complexity of the eFPGA routing fabric by requiring fewer routing channels, thereby lowering both power consumption and implementation cost; and (2) it improves CAD implementation scalability by enhancing routability as the FPSA size (number of SUs) increases, thereby reducing place-and-route runtime.

Second, analogous to how FPGA slices allow multiple small logic functions to share a single CLB, the hierarchical

organization improves utilization by enabling multiple slices within a tile to operate concurrently on independent data units (e.g., different vectors or matrices) when full-tile systolic execution is unnecessary. For example, as shown in Fig.4b and Fig.4c, a single FPSA tile concurrently executes MVM and MAC operations on independent matrices and vectors. This flexibility distinguishes FPSA from prior approaches.

C. FPSA I/O

All FPSA tile I/O ports use a latency-insensitive (LI) interface with *valid* and *ready* signals [22], [33], [34] for three reasons: LI interfaces decouple internal synchronous computation from variable-latency communication with the surrounding system (NoC, memory, DMA) [35], separating the eFPGA fabric from the rest of the SoC while enabling safe backpressure, composability (chaining multiple FPSA tiles), and reuse across integration contexts; they ease compatibility with HLS tools, which typically support fine-grained [36] or coarse-grained [37] LI designs; and, adding minimal area overhead ($< 1\%$ in our case), they let primitives interact with BRAM memories and communicate through DMA [10].

D. Operation Modes And Input Loading

Operation Modes. The flexible FPSA architecture supports multiple operation modes to handle a variety of computations that are essential for AI workloads. These include GEMM, MVM, MAC, element-wise addition/subtraction (ELT), and convolution (CONV).

Fig.4 illustrates the functional organization of a tile in these operation modes. In default mode ($op_mode = 0$), FPSA executes output-stationary GEMM on $(N \cdot K) \times (N \cdot K)$ matrices, with INT8 allowing twice the matrix size. Data flows through the tile, with matrix rows shifting horizontally and columns shifting vertically. In the MVM mode ($op_mode = 1$) shown in Fig.4b, the tile processes an $N \times N$ matrix against N vectors concurrently, enabling efficient computation of dense layers. The ELT mode ($op_mode = 3$) requires full data loading before the element-wise operations begin. FPSA also includes dedicated CONV mode ($op_mode = 4$). Finally, Fig. 4(c) depicts the tile fractured into $N \times N$ individual slices to compute MAC. However, in MAC mode, the utilization of multipliers in a tile is reduced by a factor of $1/(K \times K)$ since only a single SU in a slice has direct access to external inputs.

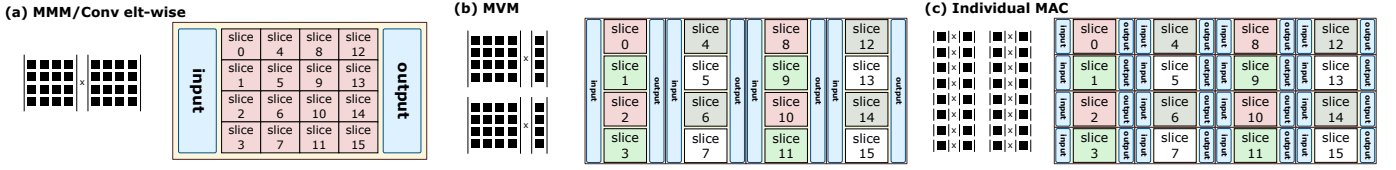


Fig. 4: An example showing different logical organizations of FPSA in different `op_mode`. (a) a full 4x4 GEMM (b) fractured MVM computation, and (c) MAC mode where FPSA is used as 16 DSPs.

The utilization in MAC mode can be improved by feeding each SU directly from external memory. However, this would increase both the number of external I/O ports and the line buffers connected to them, multiplying tile I/O count by the number of SUs and undermining FPSA's I/O reduction benefit. Exploring hybrid broadcast schemes that balance I/O count and preload latency remains an area of future work.

Although FPSA excels at compute-intensive workloads with regular data movement (e.g., GEMM, MVM, and convolution) that naturally map to its systolic architecture, workloads with dynamic control or irregular and sparse data access (e.g., decision trees) are less well suited. This limitation is a known tradeoff of systolic designs [3], [27].

Bandwidth and Input Loading. FPSAs receive data either from on-chip BRAM FIFOs or through DMA ports connected to the ASIC. Multi-ported BRAMs with configurable word widths and depths could improve bandwidth and simplify input loading, but require costly hardened macros or combinations of standard BRAMs and soft-logic multiplexing [20], [38]. Both options are area-prohibitive under the $<10\text{ mm}^2$ budget [10]. Moreover, FPSA's 64-bit input ports can fetch at most 8 INT8 operands per cycle, which is insufficient to supply all SUs simultaneously. FPSA addresses both limitations by combining simple single-ported BRAMs with internal I/O control logic, including line buffers and streaming logic (the blue boxes in Fig. 3C). This organization enables a virtual bandwidth-expansion strategy that overlaps data loading with computation [17], achieving equivalent throughput at a fraction of the area cost. The approach pre-loads multiple vectors into line buffers positioned between the input ports and the tile's topmost and leftmost slices shown in Fig. 3c, and streams them across blocks. When data arrives from BRAM FIFOs, DMA, or upstream FPSA tiles and `valid` is asserted, the I/O control logic loads the buffers. Once enough data is buffered to sustain computation without stalls, it asserts an internal `start` signal to begin streaming and execution. The required pre-load depends on the `op_mode` value: for example, in INT16 GEMM mode with $N=2$ and $K=2$, only two 64-bit reads are needed. The streaming logic also sequences inputs appropriately for single-tile and multi-tile modes. Internal input buffering offers additional advantages by allowing operand reuse, thus reducing energy consumption. It also reduces the capacity and number of BRAM needed to supply input; this is a valuable property in eFPGAs, where BRAMs are costly [10].

Cascading FPSA Tiles. FPSA tiles can also be cascaded to handle larger GEMM computations. In this mode, the inputs of a tile can originate from memory or upstream tiles in the cascade. In this case, each tile uses global x-y coordinates to

Listing 1: User input pseudocode.

```
1 void main(A[ROWS][W], B[W][COLS], OUT[ROWS][COLS]) {
2   int SA_ROWS = 4, SA_COLS = 4;
3   fpsa_mat_mul(A, B, OUT, SA_ROWS, SA_COLS); }
```

determine its logical position within the tile matrix.

IV. HLS-BASED MAPPING ON AN FPSA eFPGA

This section outlines our HLS compilation flow for mapping structured linear-algebra kernels onto FPSA-based eFPGA fabrics, focusing on GEMM as the representative case; the other use cases shown in Fig.4 are handled analogously.

A. Systolic GEMM Accelerators

Without explicit architectural support, conventional HLS tools fail to map GEMM-style loop nests onto efficient SA pipelines [27], [29], [37]. Instead, they implement the computation using standard operators (e.g., multipliers, adders), which further degrade compute density on resource-constrained eFPGAs. To solve this problem, we develop an MLIR-based [39] mapping flow, illustrated in Fig. 5(a), that translates structured C/C++ GEMM specifications into hardware mapped onto FPSA-enhanced eFPGAs through three steps: (1) **Matrix partitioning**: partition matrices and map GEMM computations across SUs; (2) **Tile packing**: organize SUs into tiles and orchestrate inter-tile communication; (3) **Intra-tile data reuse optimization**: identify reuse opportunities between adjacent tiles and cascade their connections. The resulting code explicitly encodes the dataflow, spatial organization, and temporal execution of GEMM on the SA architecture; Section V-B describes our extension of the HLS compiler [36] to translate this into RTL, shown in Fig. 5(c).

B. Matrix Partitioning

Our input is a structured C/C++ description of a GEMM as shown in Listing 1, parameterized by the matrix dimensions ($ROWS$, $COLS$, W) and systolic-array configuration (SA_{rows} , SA_{cols}), which define the dimensions of the 2D grid of SUs (see Fig.5(c)). We divide the input matrices into smaller sub-blocks: matrix A is split *row-wise* and matrix B *column-wise*. We calculate the number of rows of A ($ROWS_{SU}$) and columns of B ($COLS_{SU}$) to assign to each SU as:

$$ROWS_{SU} = \frac{ROWS}{SA_{rows}}, \quad COLS_{SU} = \frac{COLS}{SA_{cols}} \quad (1)$$

Fig.5b and Fig.5c illustrate an example where A is $16 \times W$ and B is $W \times 20$, mapped onto a 4×4 systolic array, with

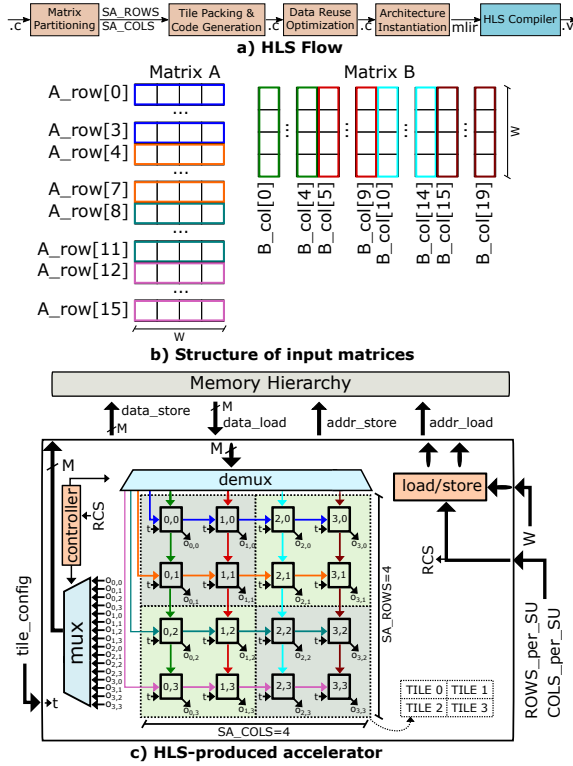


Fig. 5: Overview of the FPSA HLS mapping flow. (a) The three-step compilation pipeline from C/C++ input to synthesizable RTL (b) Row- and column-wise partitioning of input matrices A and B across the SU grid (c) The generated systolic accelerator: each rectangle represents an SU labeled by its (x,y) coordinates, edge colors denote data partitions, and dashed boxes outline FPSA tile boundaries.

each $SU(x,y)$ assigned $ROWS_{SU}$ rows of A and $COLS_{SU}$ columns of B indexed by:

$$\begin{aligned} \mathcal{R}_y &= y \cdot ROWS_{SU} + i, \\ \mathcal{C}_x &= x \cdot COLS_{SU} + j, \end{aligned}$$

where $i \in [0, ROWS_{SU} - 1]$ and $j \in [0, COLS_{SU} - 1]$. Listing 2 shows the generated mapping for $SU(1,2)$, with indexing functions $\mathcal{R}_2$ and $\mathcal{C}_1$ in lines 6 and 7. For $ROWS_{SU} = 4$ and $COLS_{SU} = 5$, $SU(1,2)$ is assigned rows 8–11 of A and columns 5–9 of B , performing $4 \cdot 5 \cdot W = 20W$ MAC operations with the result stored at $(\mathcal{R}_2, \mathcal{C}_1)$. Our partitioning ensures that all SUs in the same row share the same subset of A 's rows and that all SUs in the same column share the same subset of B 's columns, enabling balanced parallel execution and the data reuse opportunities exploited in the next steps.

C. Tile Packing and Code Generation

The previous step specifies computation at the SU level; now we organize SUs into tiles of $N \cdot K \times N \cdot K$ SUs (see Section III-A). For example, with $N = 2$, $K = 1$, four tiles of 2×2 SUs cover the full accelerator (Fig.5(c)). We transform the SU-level code in Listing 2 into tile-level code in Listing 3 by performing the following steps for each tile:

Listing 2: Pseudocode for instantiation of $SU(1,2)$.

```
1 void fpsa_SU(in_north, in_west, out_east, out_south, out_res);
2 void main(A_row[ROWS], B_col[COLS], OUT[ROWS][COLS], ROWS_SU, COLS_SU){
3   for(i=0; i < ROWS_SU; i++){
4     for(j=0; j < COLS_SU; j++){
5       ... // Code for SU(1,2), replicated for all SUs:
6       R_2 = 2*ROWS_SU + i; // Indexing function for row R_2
7       C_1 = 1*COLS_SU + j; // Indexing function for column C_1
8       SU_1_2_north = B_col[C_1]; // Read column C_1 of B
9       SU_1_2_west = A_row[R_2]; // Read row R_2 of A
10      fpsa_SU(SU_1_2_north, SU_1_2_west, SU_1_2_east, SU_1_2_south, SU_1_2_result); // Instantiate SU
11      OUT[R_2][C_1] = SU_1_2_result; }}} //Store result
```

Listing 3: Pseudocode for 2×2 FPSA tile instantiation.

```
1 void fpsa_tile_2x2(in_north, in_west, out_east, out_south, out_res);
2 void main(A_row[ROWS], B_col[COLS], OUT[ROWS][COLS], ROWS_SU, COLS_SU){
3   for(i=0; i < ROWS_SU; i++){
4     for(j=0; j < COLS_SU; j++){
5       ... // Code for tile T2, replicated for all tiles:
6       C_0, C_1, R_2, R_3 = ... // Indexing functions (details omitted)
7       SU_0_2_north = B_col[C_0];
8       SU_1_2_north = B_col[C_1];
9       // concatenate north inputs
10      T2_north = [SU_0_2_north, SU_1_2_north];
11      SU_0_2_west = A_row[R_2];
12      SU_0_3_west = A_row[R_3];
13      // concatenate west inputs
14      T2_west = [SU_0_2_west, SU_0_3_west];
15      // Instantiate tile T2
16      fpsa_tile_2x2(T2_north, T2_west, T2_east, T2_south, T2_res);
17      OUT[R_2][C_0] = T2_res[0]; //store SU(0,2) result
18      OUT[R_2][C_1] = T2_res[1]; //store SU(1,2) result
19      OUT[R_3][C_0] = T2_res[2]; //store SU(0,3) result
20      OUT[R_3][C_1] = T2_res[3]; //store SU(1,3) result
21    }}}}
```

- 1) We replace the function calls of individual SUs with a single `fpsa_tile` call; in Listing 3, e.g., SUs (0,2), (1,2), (0,3), (1,3) are replaced with tile T2.
- 2) We concatenate north inputs of the topmost SU row into a single tile north input; e.g., inputs of SUs (0,2) and (1,2) form `T2_north`.
- 3) We concatenate west inputs of the leftmost SU column into a single tile west input; e.g., the inputs of SUs (0,2) and (0,3) form `T2_west`.
- 4) We assign SU outputs to corresponding partitions of the tile output in row-major order (lines 13–16 in Listing 3).
- 5) We remove redundant memory accesses for data flowing directly between internal SUs; e.g., since $SU(1,2)$ receives its west input from $SU(0,3)$, $SU_{1_2_west}$ is eliminated.

D. Inter-Tile Data Reuse Optimization

In the final step, we optimize data reuse across tiles using FPSA's built-in cascading (Section III-D). The procedure is conceptually similar to the tile-packing step: (1) replace each non-topmost tile's north input with the south output of its preceding tile; (2) replace each non-leftmost tile's west input with the east output of its preceding tile; (3) remove all memory accesses no longer assigned to any tile. For example, since Tile 2 receives its north input from Tile 0, `T2_north`

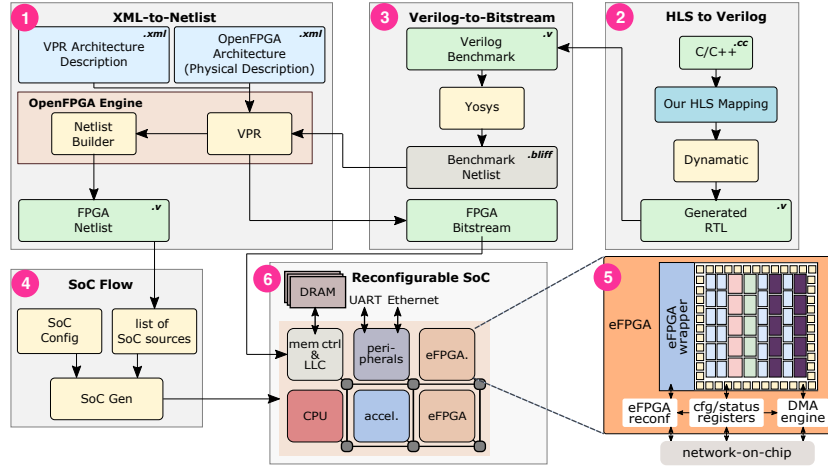


Fig. 6: The framework implementation leverages three open-source projects: OpenFPGA, Dynamic HLS, and ESP.

is replaced by `T0_south` and the memory accesses to `B_col` (lines 7–8 in Listing 3) are removed. This ensures each data element is read from memory exactly once and forwarded via FPSA’s LI interface, eliminating additional soft logic for synchronization. The resulting code directly encodes memory access patterns, tile-level mapping, and data reuse, and is passed to the HLS compiler for RTL generation (Section V-B).

V. THE IMPLEMENTATION OF THE FRAMEWORK

This section presents our end-to-end framework, from FPSA-enhanced eFPGA design to SoC integration and compilation of the accelerators in Section IV into deployable bitstreams. Fig.6 shows the complete flow.

A. eFPGA Fabric Design

Like BRAMs and DSPs in commercial FPGAs, FPSA is a design-time primitive that must be integrated into the eFPGA fabric before fabrication. Although several open-source frameworks [10], [19], [40], [41] and CAD tools [42], [43] simplify eFPGA design, creating fabrics with custom blocks such as FPSA remains challenging due to complex physical design flows, timing issues from combinational loops, and the difficulty of accurately estimating PPA without performing an expensive full physical design. Fig. 6 shows our fabric generation flow. Our framework automates fabric generation by using OpenFPGA [19], an open-source tool that generates eFPGA prototypes from high-level architectural descriptions (Step 1). OpenFPGA integrates Yosys [42] for synthesis and VPR [43] for place-and-route of accelerators targeting the fabric. We extend OpenFPGA with custom architecture files (based on Lattice iCE40 [44]) and Yosys scripts for accurate technology mapping of eFPGA resources and FPSA instances from HLS code. Finally, we enhance OpenFPGA with a performance estimation method that combines static timing analysis (STA) with VPR back-annotation [45], [46], allowing rapid and reliable PPA exploration without the cost of the full physical design.

B. Architecture Instantiation through HLS

To enable efficient hardware mapping, we adopt a dynamic HLS flow that naturally aligns with the LI interfaces used by both FPSA tiles and SoC system components (NoC, DMA, CPU, etc.). Unlike statically scheduled HLS designs, which require additional wrapper logic, the dynamic approach provides seamless communication and automatic timing alignment, allowing FPSA tiles to integrate cleanly with other HLS-generated modules and the SoC [36], [37].

We build upon Dynamic [36], an open-source HLS compiler that translates C code into LI circuits, to automatically generate the following components from the code obtained by our mapper in Section IV: (1) *memory ports* (labeled `data_load`, `data_store` in Fig.5c) of width M to access the input and output matrices; (2) an *FPSA tile* for each `fpsa_tile` function call; (3) *mux/demux logic* when multiple systolic tile ports share `data_load` or `data_store`; (4) a *load/store request unit* that computes memory requests driving `addr_load` and `addr_store` based on the data parameters discussed in Section IV-B; (5) *controller logic* that manages data transfers according to the access pattern defined by the load/store request unit and drives the mux/demux selection.

We extend Dynamic’s handshake IR with the units above and add the corresponding parametric RTL component descriptions to Dynamic’s RTL library. We enhance Dynamic’s memory interface to generate burst-mode memory requests with transfers adjusted for port width M : zero-padded if $M > W$, or split across multiple requests if $M < W$. Dynamic then automatically generates these components from the input C code, connects them with LI signals, and produces the Verilog circuit description corresponding to the circuit in Fig.5c.

C. HLS-to-Bitstream Generation

Steps 2 and 3 in Fig. 6 represent the generation of RTL from HLS and the bitstream generation from Verilog specifications, respectively. The HLS flow receives C/C++ code as input and generates the accelerator RTL, with custom primitives instantiated via Dynamic. OpenFPGA then generates the

bitstream from this HLS-produced accelerator RTL, employing Yosys and VPR to pair the generated fabric with bitstream generation for custom eFPGA fabrics. We modified the Yosys script to extract and map each resource in the synthesizable accelerator Verilog (i.e., LUTs, DFFs, and custom primitives) directly to its corresponding fabric resource in the architecture.

D. eFPGA-SoC Integration

We integrate our eFPGAs into heterogeneous SoCs using ESP [47]–[49], an open-source tile-based platform built on a 2D-mesh NoC. To enable system-level deployment of FPSA-enhanced eFPGAs, we extend ESP with a new eFPGA tile containing eFPGA-specific interfaces, a DMA engine, and runtime reconfiguration support. As shown in step 5 in Fig. 6, the tile wraps the eFPGA fabric generated by the OpenFPGA flow and adds the logic required to synchronize clock-domain crossings between the fabric and the surrounding SoC. This software-actuated DMA engine, connected to the ESP NoC, fetches bitstreams from on-chip memory or external DRAM to program the fabric, and is later reused for runtime operand and result transfers, as described next.

E. Runtime Execution

FPSA is exposed to the reconfigurable SoC (step 6 in Fig. 6) as a custom accelerator: HLS-synthesized RTL, generated as described in Section V-C, that wraps the FPSA tile array and implements lightweight control and datapath logic to interface with the ESP NoC sockets [47], [50]. The programming flow proceeds in three steps: (1) the host CPU configures the accelerator’s memory-mapped registers with the matrix dimensions and FPSA operation mode; (2) the CPU preloads weights and activations into the shared local memory; and (3) the CPU starts the accelerator by writing to its `start` register. Once started, the accelerator autonomously streams operands from shared local memory into the FPSA tile array, writes the resulting output tensor back to shared local memory via the same DMA engine, and raises an interrupt to notify the CPU upon completion. Data reuse across chained tiles is handled entirely through FPSA’s latency-insensitive cascading interface, requiring no additional synchronization logic.

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate four key aspects of our work: (i) the quality of the FPSA architecture with a one-to-one comparison against two state-of-the-art (SoA) AI primitives [3], [17] and a regular DSP similar to those used in Intel Agilex devices [51]; (ii) a comparison between FPSA-enhanced eFPGAs and baseline eFPGAs without FPSA across a set of AI benchmarks; (iii) a complete heterogeneous reconfigurable SoC featuring eFPGAs for BCI and HEP Edge AI applications; and (iv) a fabric-level design-space exploration using a modern vision transformer (ViT-Tiny) workload, examining the area-throughput tradeoff under a fixed silicon budget.

TABLE I: Per-primitive area and power characterization used for area/power estimation across all fabric configurations.

eFPGA component	Area (μm^2)	Power (mW)
LUT	131	0.004
CLB	4878	0.16
BRAM	5961	2.52
DSP	2483	0.36
Connection box	79	0.0016
Switch box	140	0.0789

TABLE II: Comparison summary between FPSA variants.

Variant	F_{max} (MHz)	Primitive type	MAC/ sec	GMAC/ cycle	Area (μm^2)	Power (mW)	Density $\frac{GMAC}{sec \cdot mm^2}$	Power-eff $\frac{GMAC}{sec \cdot W}$
FPSA-small	1000	tile slice SU	16	16	16637 2887 634	8.01 1.23 0.76	961.7	1997
FPSA-medium	909	tile slice SU	64	57.28	68142 2824 616	29.3 1.12 0.66	840	1973
FPSA-large	833	tile slice SU	256	212	288252 2887 617	111.5 1.03 0.6	737	1905

A. Experimental Setup

Prototyping eFPGA fabrics. We prototype our custom eFPGA fabrics using OpenFPGA’s [19] XML-to-Prototype flow, which generates synthesizable Verilog from the architecture descriptions introduced in Section V. All fabrics and primitives are implemented in a 12nm FinFET technology process and synthesized with Cadence Genus to obtain area and power estimates. We use Synopsys PrimeTime for static timing analysis and back-annotate the resulting timing estimates into the architecture description, enabling VPR to model critical-path delays. Table I summarizes the resulting primitive- and fabric-level implementation data.

B. FPSA Architectural Evaluation

We evaluate three FPSA variants: FPSA-small ($N=2$, $K=2$), FPSA-medium ($N=4$, $K=2$), and FPSA-large ($N=8$, $K=2$), corresponding to 4×4 , 8×8 , and 16×16 SU arrays, respectively. K is fixed at 2 across all variants as a design-time choice that balances tile compute capacity against area and I/O overhead; N is varied to characterize the slice-level compute-density tradeoff. We compare FPSA against three representative baselines. First, we use the tensor-slice (TS) architecture from [3], which is architecturally equivalent to FPSA-small and therefore supports a direct one-to-one comparison. For fairness, we synthesize TS without FP16 support and compare both designs only in fixed-point modes. Second, we compare against an open-source Intel-Agilex-like DSP implementation [52], synthesized without floating-point support and evaluated only at INT16 precision. Third, we compare against Intel Stratix NX tensor blocks (STBs) [17]. Because public STB implementations do not provide complete area and timing data, this comparison is analytical and uses published estimates scaled to 12nm following the approach of Stillmaker and Baas [53]. Our comparisons with academic and commercial primitives (Tensor Slices, Agilex DSP, and STBs) are primitive-to-primitive evaluations that exclude routing overhead on both sides, and should be interpreted as assessments of raw compute-primitive efficiency. We target edge SoCs operating with a 1–5 W power envelope and a $<10 \text{ mm}^2$ area budget; GPUs and high-end standalone platforms, such as Versal-class devices, occupy a fundamentally different design point and fall outside our comparison’s scope.

TABLE III: Comparison of FPSPA against SoA primitives.

Primitive	F_{max} (MHz)	MAC/cycle	INT16 precision				Power-eff $\frac{GMAC}{sec \cdot mm^2}$
			GMAC/sec	Area (μm^2)	Power (mW)	Density $\frac{GMAC}{sec \cdot mm^2}$	
FPSPA-small	1000	16	16	16637	8.01	961.7	1997
TS [3]	625	16	10	27534	8.61	363.1	1161
Agilex-DSP	1176	2	2.35	3081	1.14	759	2058

Comparing FPSPA Variants. Table II summarizes the FPSPA variants. FPSPA-small is the fastest, reaching 1 GHz, while FPSPA-large is only 16% slower despite providing 16 $\times$ more compute. Area and power scale close to linearly with compute capacity: FPSPA-large incurs 17.3 $\times$ the area and 13.9 $\times$ the power of FPSPA-small. Compute and power density slightly decline for larger variants, primarily due to buffering and I/O overhead needed to sustain higher throughput (Fig. 3c).

Intra-Tile Hierarchy Overhead. To quantify the cost of the slice hierarchy, we compare the total slice area with the effective compute area (i.e., the area occupied by the SUs alone) using the data reported in Table II. The maximum overhead is 13% in FPSPA-large, indicating that additional flexibility and local routing support come at a modest area cost. In general, FPSPA scales efficiently while maintaining a high compute density.

Comparison with Agilex-DSP. Table III compares FPSPA-small with an Agilex-like DSP block. Although the DSP is 5.3 $\times$ smaller and achieves a slightly higher frequency (1.1 $\times$), it supports only two INT16 MACs per cycle and therefore reaches just 2.35 GMAC/s. FPSPA-small delivers 16 GMAC/s, a 6.8 $\times$ throughput improvement, and achieves 1.26 $\times$ higher compute density at comparable power efficiency. This result highlights that packing more tensor-oriented compute into the primitive can improve compute density despite the larger block size.

Comparison with Tensor Slice (TS). Tensor Slices is the architectural baseline closest to FPSPA, with FPSPA-small as its direct architectural equivalent¹. As shown in Table III, FPSPA achieves 60% higher maximum frequency, occupies 39% less area, and delivers 2.6 $\times$ higher peak compute density. This gap stems from architectural style: TS implements its operating modes through a centralized monolithic organization with more mode-dependent control and output-handling logic around the systolic core, whereas FPSPA's hierarchical tile/slice/SU structure uses simpler local forwarding and more regular per-slice control, reducing internal muxing and non-compute overhead, shortening critical paths, and lowering area. The slice hierarchy's area overhead does reduce FPSPA's compute-area efficiency slightly below TS's (87% in Table III versus 90% [3]), but it is precisely this overhead that affords FPSPA its greater flexibility and integrated control logic, and in turn, its higher overall performance.

At the fabric level, chaining systolic arrays requires precise timing alignment across tiles. FPSPA addresses this with latency-insensitive interfaces and valid/ready handshakes that provide robust flow control, whereas TS lacks such mechanisms and therefore requires more complex coordination logic. Notably, a recent TS extension added buffers to improve performance [4], further supporting the core design premise

TABLE IV: Comparison between Intel STBs and FPSPA.

Feature	STB [17]	FPSPA-small
F_{max} (MHz)	793	1100
Area (μm^2)	10077	16637
INT8 multipliers	30	64
Compute density ($GMAC/s \cdot mm^2$)	2360	4231
Precisions	INT8, INT4	INT8, INT16
Scalar Mode (MAC/cycle)	3	16
Vector Mode (MAC/cycle)	6	16
Tensor Mode GEMM (MAC/cycle)	30 (100%)	64 (100%)
Tensor Mode MVM (MAC/cycle)	10 (33%)	32 (50%)

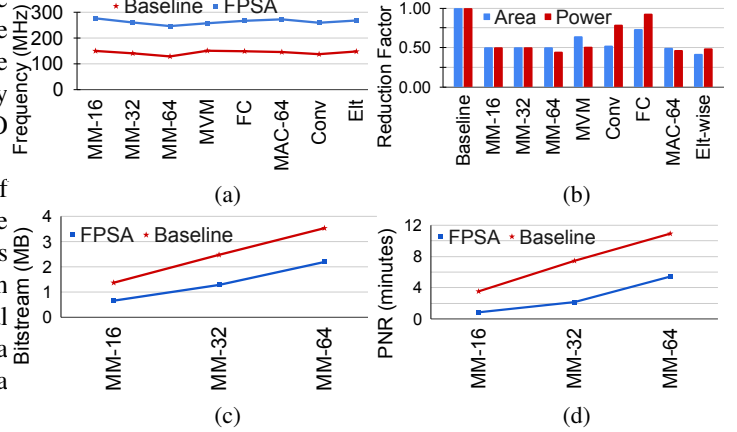


Fig. 7: FPSPA-enhanced vs. baseline eFPGA fabrics across the AI benchmark suite: (a) frequency, (b) area and power reduction, (c) bitstream size, and (d) place-and-route time.

adopted in FPSPA. FPSPA also has a smaller fabric footprint: each TS tile spans six CLB columns, while FPSPA requires only four (Table I)². This compact footprint enables 50% more FPSPA tiles in the same fabric area, or alternatively leaves additional area for general-purpose logic, which is important in constrained eFPGAs.

Key takeaway. *These results support our central claim that FPSPA's hierarchical organization is a better fit for resource-constrained eFPGAs than monolithic primitives: localized control and forwarding win frequency, area, and density at the primitive level, while a smaller CLB-column footprint with fewer demanded routing channels wins fabric-level scalability.*

Comparison with Intel Stratix Tensor Blocks. Stratix Tensor Blocks (STBs) [17] replace DSPs in Intel FPGAs with 30 INT8 MACs organized as three dot-product units. We compare them analytically against the architecturally equivalent FPSPA-small using optimistic STB area and frequency estimates from prior work, scaled to 12nm [14], [53]. As shown in Table IV, STBs are 28% smaller, but FPSPA achieves 38% higher F_{max} and 79% higher peak compute density. Across operating modes, FPSPA also sustains higher effective utilization. In the MAC-like modes, corresponding to STB scalar and vector execution, FPSPA delivers 5.3 $\times$ and 2.6 $\times$ higher throughput, respectively. In tensor mode, it maintains full utilization for GEMM while achieving higher MVM utilization than STB (50% versus 33%) [54].

²Column span is estimated from each tile's total area (Table II, Table III) relative to the CLB area in Table I, combined with the fabric's column pitch.

¹We thank the authors of [3] for providing us their code for comparison.

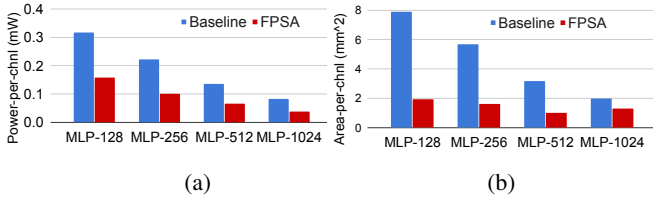


Fig. 8: (a) Power-per-channel and (b) Area-per-channel comparison of baseline vs FPSA.

C. AI-optimized vs. Baseline eFPGAs

We compare a baseline eFPGA fabric composed only of traditional primitives (LUTs, FFs, BRAMs, and DSPs) against fabrics enhanced with FPSA-small tiles to quantify the fabric-level benefit of FPSA.

AI Benchmarks. To evaluate the architectural impact of FPSA on AI kernels, we construct a benchmark suite similar to Koios [55] and re-implement representative kernels in C++ for the flow described in Section V. The suite includes GEMM (16×16 , 32×32 , 64×64 ; INT16/INT8), MVM ($64 \times 64 \times 64 \times 32$), element-wise add/sub (64×64), convolution (32×32 input, $4 \times 3 \times 3$ filters, stride 1), fully-connected ($42 \times 16 \times 42 \times 16$), and simple MACs on 64×64 operands. Each benchmark is implemented once on the baseline eFPGA and once on an FPSA-enhanced fabric; larger matrix kernels are mapped by chaining multiple FPSAs.

Evaluation. Fig. 7 summarizes the results. Relative to the baseline fabric, FPSA improves frequency by up to 90% (MM-64), reduces area and power by up to 51% and 54%, respectively, lowers bitstream size, and cuts place-and-route time by 50% on average. These gains arise because FPSA replaces LUT- and routing-intensive tensor datapaths with hardened systolic compute, reducing both critical path overhead and configuration cost. The measured F_{max} is slightly lower than the primitive-only values in Table III due to the added routing overhead of the surrounding FPGA fabric [20].

Non-AI Benchmarks. To assess the cost of specialization, we also map several non-AI VTR benchmarks [41], following Taka et al. [4]. Compared to the baseline fabric, the FPSA-enhanced eFPGA shows only a 2% average drop in F_{max} , indicating that FPSA improves AI-oriented fabric efficiency without significantly compromising the baseline eFPGA's general-purpose utility.

D. Case Studies: BCI and HEP

Reconfigurable BCI Applications. BCIs are a compelling target for FPSA-enhanced eFPGAs because both sensing scale and effective workload size vary over time. As neural interfaces integrate more recording channels [56], [57], DNN-based decoding models must scale accordingly to preserve application accuracy [58]. At the same time, neural activity is sparse and spatially correlated [59], [60], so processing all channels uniformly is often unnecessary. This makes BCI a natural setting for reconfigurable AI acceleration: the system should adapt the deployed model to the subset of active channels while remaining within strict implant- or edge-class power and area budgets [61].

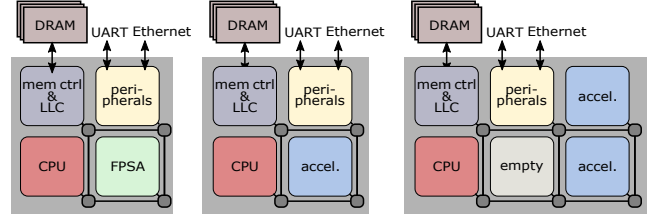


Fig. 9: Heterogeneous SoCs used for evaluation.

To evaluate this capability, we implement a multi-layer perceptron (MLP) for speech synthesis from electrocorticography brain signals [62]. The original model, designed for 128 sensing channels sampled at 2 kHz, consists of four layers and generates 40 labels corresponding to speech frequencies used in audio reconstruction. We first map this baseline model (MLP-128) onto both the baseline and FPSA-enhanced eFPGAs, then scale the workload to larger channel counts (MLP-256, MLP-512, and MLP-1024) by proportionally increasing network depth and width. We evaluate *power per channel* and *area per channel*, which are critical metrics in BCI systems [57], [63].

Fig. 8a and 8b show that both BCI metrics scale sublinearly with model size and channel count, but FPSA consistently improves per-channel efficiency across all model sizes. Compared with the baseline eFPGA, the FPSA-enhanced design reduces power per channel by up to $2.64\times$ and area per channel by up to $4\times$. These gains are especially important for BCI systems, where compute must coexist with tight sensing, communication, and thermal budgets.

Key takeaway. More broadly, these results show that FPSA enables eFPGAs to support dynamically resizable neural workloads with substantially better system-level efficiency.

Reconfigurable HEP Application. Modern AI-based HEP applications process massive detector data streams under strict latency constraints, while radiation damage, temperature fluctuations, hardware aging, and mechanical shifts can alter detector behavior, requiring recalibration and adaptation of the AI model [64]. This makes HEP a natural target for reconfigurable acceleration.

To evaluate this setting, we consider two DNN models: nn-1 [65], a lightweight anomaly-detection model evaluated on the MicroBooNE dataset [66], and nn-2, a Lenet-like [67], pattern recognition classifier. We design four heterogeneous SoCs with a common CPU (CVA6 [68]), memory hierarchy, and I/O, but different AI acceleration subsystems: a baseline eFPGA (SoC 1), an FPSA-enhanced eFPGA (SoC 2), a fixed-function nn-1 accelerator with nn-2 running in software on the CPU (SoC 3), and two lightweight fixed-function accelerators, one per model (SoC 4). In SoC 1 and SoC 2, the two models are swapped at runtime on the same eFPGA fabric. In all cases, nn-1 and nn-2 are executed sequentially, representing a common HEP inference pipeline. We evaluate end-to-end throughput, compute density, and energy efficiency.

Fig. 10a and 10b compare the FPSA-based SoC against two fixed-function baselines. Relative to SoC 3, the FPSA-based SoC achieves $2.4\times$ higher throughput by executing both

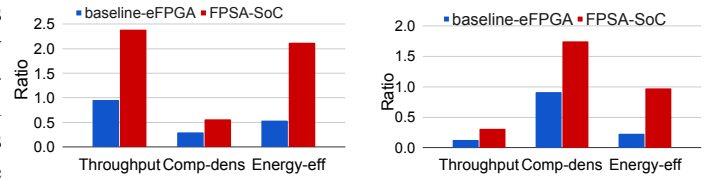
models efficiently on the same reconfigurable fabric. This result does not suggest that reconfigurable logic is intrinsically more efficient than ASIC logic; rather, it highlights a system-level advantage of reconfigurability for sequential multi-model workloads. SoC 4 is a stronger baseline because it integrates one dedicated ASIC accelerator per model, eliminating the software bottleneck. Even against this more favorable configuration, the FPSA-based SoC still achieves $1.67\times$ compute density at comparable energy efficiency. Interestingly, comparing the two reconfigurable SoCs directly, the AI-optimized SoC 2 against the baseline SoC 1, yields the largest payoff of all: $4.2\times$ higher energy efficiency and $2.1\times$ higher compute density (Fig. 10b). **Key takeaway.** *These results show that a single FPSA-enhanced eFPGA can consolidate the functionality of multiple specialized accelerators while remaining competitive with fixed-function designs.*

E. Modern Edge-AI Inference Workloads

To demonstrate that FPSA generalizes to modern edge-AI inference workloads and to provide practical guidance for fabric designers, we evaluate the dense linear components of ViT-Tiny [23], a widely deployed vision transformer for edge inference, as a case study. We use this workload to answer a concrete design question: *given a fixed silicon compute budget, should a designer build the fabric using many FPSA-small tiles or fewer FPSA-large tiles?* We first characterize the ViT-Tiny workload and describe its mapping onto FPSA. We then define a set of realistic fabric configurations under a fixed area budget and evaluate each configuration in terms of frequency, area, power, and throughput. All configurations are synthesized using a 12nm FinFET technology process via Cadence Genus for area and power estimation. $F_{\max}$ is obtained from VPR [43] place-and-route runs.

Workload Characterization. ViT-Tiny processes 224×224 images as sequences of 197 tokens with embedding dimension 192, across 12 transformer layers [23]. Both weights and activations are quantized to 8-bit fixed-point (INT8), leveraging FPSA's fracturable multiplier mode to maximize compute density within the edge power budget. Each layer contains six dense linear operations: three independent Q, K, and V projections ($197\times 192 \times 192\times 192$ each), an output projection ($197\times 192 \times 192\times 192$), and two feed-forward layers, FC1 ($197\times 192 \times 192\times 768$) and FC2 ($197\times 768 \times 768\times 192$). Each operation reduces to a GEMM of the form $\mathbf{A}\times\mathbf{B}$, where $\mathbf{A}$ encodes the token sequence and $\mathbf{B}$ the weight matrix, and maps directly onto FPSA using the HLS flow from Section IV. Across all 12 layers, these operations account for approximately 1.05 B MACs and dominate the computational cost of ViT-Tiny inference. Nonlinear operations (GELU activation, softmax) are executed on the host CPU or another accelerator and are excluded from the GEMM-execution target, consistent with FPSA's focus on dense tensor acceleration.

Mapping Strategy. Each linear operation of the transformer is mapped onto FPSA using the three-step HLS flow described in Section IV: matrix partitioning, tile packing, and inter-tile data reuse optimization. At INT8 precision, FPSA's fracturable multiplier doubles the effective SU count per dimension,



(a) Normalized to ASIC SoC 3

(b) Normalized to ASIC SoC 4

Fig. 10: System-level evaluation: eFPGA vs ASIC SoCs.

yielding effective sub-matrix sizes of 8×8 , 16×16 , and 32×32 for FPSA-small, medium, and large, respectively. Following the partitioning scheme of Section IV-B, $\mathbf{A}$ is divided row-wise and $\mathbf{B}$ column-wise across the SU grid. The resulting number of sequential tile invocations per operation on a single tile is reported in Table V for each operation and FPSA variant. We illustrate how these counts are derived with an example.

For example, mapping the Q projection ($197\times 192 \times 192\times 192$) onto FPSA-large at INT8 precision (effective 32×32 grid) gives $ROWS_{SU} = \lceil 197/32 \rceil = 7$ and $COLS_{SU} = \lceil 192/32 \rceil = 6$ via Equation 1 (Section IV), yielding $7 \times 6 = 42$ sequential tile invocations on a single FPSA-large tile, with $\mathbf{A}$ and $\mathbf{B}$ sub-blocks reloaded from memory at every invocation, consistent with Table V. By combining FPSA's cascading mechanism (Section III-D) with our inter-tile data reuse optimization (Section IV-D), multiple tiles can be chained to process sub-matrix pairs in parallel, proportionally reducing the number of sequential invocations and amortizing memory-access costs across tiles: a full 7×6 cascaded array collapses all 42 invocations into one, while intermediate configurations such as 7×3 or 7×2 trade area for proportionally fewer invocations. Therefore, the number of chained tiles defines a PPA tradeoff: more tiles reduce sequential invocations but consume greater fabric area. For example, fully parallelizing all invocations within a single transformer layer would require 378 FPSA-large tiles, occupying 108.9mm^2 and exceeding the $<10\text{mm}^2$ area budget by more than $10\times$. Therefore, for practical deployments, fabric designers must balance parallelism against area. We examine this tradeoff next through a controlled design-space exploration (DSE) that fixes total compute capacity while varying tile granularity and tile count.

Fabric DSE. As shown in Section VI-B, FPSA-large offers $16\times$ the compute capacity of FPSA-small but occupies $17.3\times$ more area and delivers $13.9\times$ better power efficiency. Hence, a fabric of many small tiles and a fabric of few large tiles may offer equivalent raw compute capacity yet differ substantially in area footprint, power consumption, and achievable $F_{\max}$, making tile granularity an important architectural decision rather than a simple scaling choice. To explore this tradeoff systematically, we define two *iso-compute classes* in which total FPSA compute capacity (total SU number) is held equal across variants, achieved by scaling tile count inversely with tile size, so that observed differences in $F_{\max}$, area, power, and throughput are attributable solely to tile granularity and the complexity of the surrounding routing and control logic.

Class 1 targets a small fabric budget: **C1-S** uses 4×4 FPSA-

small tiles (16 tiles), **C1-M** uses 2×2 FPSA-medium tiles (4 tiles), and **C1-L** deploys a single FPSA-large tile. Class 2 targets a larger budget: **C2-S** uses 8×8 FPSA-small tiles (64 tiles), **C2-M** uses 4×4 FPSA-medium tiles (16 tiles), and **C2-L** uses 2×2 FPSA-large tiles (4 tiles). Each class also includes a baseline eFPGA fabric with equivalent LUTs, BRAMs, and DSPs but no FPSA tiles as a reference. To ensure a fair iso-compute comparison, we configure Yosys [42] to pack multiple INT8 multiplications per DSP block, matching the total number of multiply-accumulate units (SUs) in each class: Class 1 baselines pack 2 INT8 multiplications per DSP, and Class 2 baselines pack 4. This ensures that observed differences in area, power, $F_{\max}$, and throughput between the baseline and FPSA-enhanced configurations reflect architectural efficiency rather than differences in raw compute parallelism.

Comparative Evaluation. Table VI reports the resource utilization, area, power, $F_{\max}$, throughput, and throughput density for all eight configurations. The baseline eFPGA fabrics exceed the $<10\text{mm}^2$ area budget by more than $9\times$ in both classes (90.49mm^2 and 100.2mm^2 , respectively) and consume over 3 W of power; this leaves negligible headroom for the remaining SoC components within a 1–5 W edge power budget. Despite this large resource footprint, the baseline achieves only 34.2 GMAC/s and 62.1 GMAC/s in Class 1 and Class 2, respectively, yielding a throughput density of 0.378 and 0.62 GMAC/s/ mm^2 . In contrast, all FPSA-enhanced configurations fit comfortably within both area and power constraints, with C1-L achieving the smallest footprint at 0.67mm^2 and lowest power at 131 mW, a $135.6\times$ area reduction and $24.8\times$ power reduction over its class baseline, while delivering 256.5 GMAC/s more than $7.5\times$ higher than the baseline. The PPA gap between the baseline and all FPSA-enhanced configurations confirms that general-purpose fabrics are insufficient for edge transformer inference and that AI-specialized primitives such as FPSA are essential for simultaneously meeting area, power, and timing constraints.

Within each iso-compute class, larger FPSA tiles consistently outperform smaller ones across all metrics. In Class 1, moving from C1-S to C1-L reduces area by $3.45\times$ and power by $1.93\times$, while increasing $F_{\max}$ by $1.44\times$ and throughput density by $4.95\times$ (from 77.3 to 382.8 GMAC/s/ mm^2). This is because, although FPSA-small achieves $1.17\times$ higher frequency in isolation (Table II), connecting 16 tiles through external routing shifts the critical path to the inter-tile logic, reducing $F_{\max}$ and diminishing the throughput advantage of smaller tiles. Class 2 shows consistent trends, with C2-L achieving $1.93\times$ smaller area, $1.34\times$ lower power, and $2.66\times$ higher throughput density than C2-S. Across classes, Class 2 exhibits $\sim 1.3\times$ lower $F_{\max}$ and $2.6\text{--}3.7\times$ higher power than Class 1, reflecting longer routing paths in larger arrays; even so, C2-L consumes only 488 mW and achieves 765.0 GMAC/s, well within edge constraints.

Key takeaway. *These results suggest that, for ViT-Tiny-class workloads, a smaller number of large FPSA tiles tends to outperform a larger number of small tiles across area, power, and frequency, making FPSA-large a favorable choice for edge transformer inference under tight area budgets.*

TABLE V: Number of invocations per transformer layer per single FPSA tile for each ViT-Tiny dense linear operation.

Operation	MACs	FPSA-small	FPSA-medium	FPSA-large
Q Projection	7.3 M	600	156	42
K Projection	7.3 M	600	156	42
V Projection	7.3 M	600	156	42
Output Proj	7.3 M	600	156	42
MLP FC1	29.0 M	2,400	624	168
MLP FC2	29.0 M	600	156	42
Total / layer	87.2 M	5,400	1,404	378
Total / inference	$\sim 1.05\text{ B}$	64,800	16,848	4,536

VII. RELATED WORK

Heterogeneous SoCs Integrating eFPGAs. eFPGA platforms offer lightweight on-the-fly reconfigurability and customization of hardware accelerators, interfaces, or other digital logic functions and can be tailored to meet specific PPA requirements of the target application, providing designers with greater flexibility and scalability compared to fixed-function ASIC designs [8], [69], with preliminary works showing the advantages of this integration in heterogeneous SoC architectures [7], [9]. However, the increase in heterogeneity translates into a higher design effort [70]. Open-source SoC platforms have emerged to mitigate design costs [71], [72], but most provide little to no support for eFPGA integration, and prior work has rarely examined full eFPGA–SoC integration.

Systolic Architectures in HLS. HLS tools have greatly enhanced productivity by raising the level of abstraction [70], [73]; several prior studies optimize systolic array architectures using high-level [27], [28], [74] or domain-specific languages [75]. AutoSA [27], one of the most recent contributions, maps nested loop sections in C-level code to systolic units and applies loop optimizations to improve PE data distribution. The ARIES framework [76] introduces a compiler IR for AI engines but is tied to a specific architecture; in contrast, our open-source, vendor-independent HLS methodology lets users define custom modules, making it more general, flexible, and extensible across applications.

VIII. CONCLUSION

This work establishes the viability of eFPGAs for Edge-AI acceleration under tight power, performance, and area constraints. Our main contribution is FPSA, an eFPGA-native AI primitive whose hierarchical organization reduces interconnect pressure, improves utilization, and enables efficient tensor computation in resource-constrained eFPGA fabrics. Combined with an HLS-based compilation flow and full SoC integration support, our approach shows that architectural specialization and system-level integration can make eFPGAs a practical platform for real-world deployment.

ACKNOWLEDGMENT

The authors used a generative AI tool (Claude, Anthropic) for proofreading and LaTeX/BibTeX error-checking. It was not used for research ideas, technical content, or results; all claims and conclusions are the authors' own.

TABLE VI: Resource utilization, area, and power for iso-compute fabric configurations.

Config	CLBs	DSPs	BRAMs	FPSA Tiles	Area (mm ²)	Area Reduc.	Power (mW)	Power Reduc.	Norm. F _{max}	Tput (GMAC/s)	Tput Density (GMAC/s/mm ²)
Class 1 — Small Budget											
C1-Baseline	18,417	128	56	0	90.49	1×	3,250	1×	1×	34.2	0.378
C1-S	388	0	24	16	2.30	39.3×	253	12.8×	2.00×	177.8	77.3
C1-M	132	0	8	4	0.97	93.7×	155	20.9×	2.47×	219.9	226.7
C1-L	74	0	3	1	0.67	135.6×	131	24.8×	2.88×	256.5	382.8
Class 2 — Large Budget											
C2-Baseline	20,314	256	50	0	100.2	1×	3,590	1×	1×	62.1	0.62
C2-S	480	0	24	64	3.55	28.2×	653	5.4×	1.72×	555.4	156.5
C2-M	349	0	24	16	2.94	34×	571	6.2×	2.08×	672.2	228.6
C2-L	132	0	8	4	1.84	54.4×	488	7.3×	2.36×	765.0	415.8

REFERENCES

- [1] R. Singh *et al.*, “Edge AI: A Survey,” *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023.
- [2] I. Kuon *et al.*, “Measuring the Gap Between FPGAs and ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.
- [3] A. Arora *et al.*, “Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 23–33.
- [4] E. Taka *et al.*, “Systolic Sparse Tensor Slices: FPGA Building Blocks for Sparse and Dense AI Acceleration,” in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2025, pp. 159–171.
- [5] G. Gobieski *et al.*, “Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture,” in *IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.
- [6] L. Liu *et al.*, “A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications,” *ACM Computing Surveys*, vol. 52, no. 6, 2019.
- [7] F. Gao *et al.*, “DECADES: A 67mm², 1.46TOPS, 55 Giga Cache-Coherent 64-bit RISC-V Instructions per Second, Heterogeneous Many-core SoC with 109 Tiles including Accelerators, Intelligent Storage, and eFPGA in 12nm FinFET,” in *IEEE Custom Integrated Circuits Conference (CICC)*, 2023, pp. 1–2.
- [8] P. D. Schiavone *et al.*, “Arnold: An eFPGA-Augmented RISC-V SoC for Flexible and Low-Power IoT End Nodes,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4.
- [9] S. K. Lee *et al.*, “SMIV: A 16-nm 25-mm² SoC for IoT with Arm Cortex-A53, eFPGA, and Coherent Accelerators,” *IEEE Journal of Solid-State Circuits*, vol. 57, no. 2, 2022.
- [10] D. Koch *et al.*, “FABulous: An Embedded FPGA Framework,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 45–56.
- [11] A. Li *et al.*, “Duet: Creating Harmony Between Processors and Embedded FPGAs,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 745–758.
- [12] R. Razdan *et al.*, “A High-Performance Microarchitecture with Hardware-Programmable Functional Units,” in *Proceedings of the 27th Annual International Symposium on Microarchitecture*.
- [13] “Versal AI Engine,” <https://www.xilinx.com/products/intellectual-property/versal-ai-engine.html>, 2023.
- [14] A. Arora *et al.*, “Tensor Slices: FPGA Building Blocks for the Deep Learning Era,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–34, 2022.
- [15] “FlexLogix InferX AI,” <https://flex-logix.com/inferx-ai/inferx-ai-hardware/>, 2023.
- [16] “Achronix,” <https://www.achronix.com/product/speedster7t-fpgas>, 2023.
- [17] M. Langhammer *et al.*, “Stratix 10 NX Architecture and Applications,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 57–67.
- [18] A. Arora *et al.*, “Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks,” in *IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2020, pp. 53–60.
- [19] X. Tang *et al.*, “OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs,” *IEEE Micro*, vol. 40, no. 4, 2020.
- [20] A. Boutros *et al.*, “FPGA Architecture: Principles and Progression,” *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.
- [21] L. Carloni *et al.*, “Theory of Latency-Insensitive Design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.
- [22] L. P. Carloni, “From Latency-Insensitive Design to Communication-Based System-Level Design,” *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2133–2151, 2015.
- [23] H. Touvron *et al.*, “Training Data-Efficient Image Transformers & Distillation Through Attention,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 10 347–10 357.
- [24] “VTR Xilinx 7 Series Architecture File,” https://github.com/verilog-to-routing/vtr-verilog-to-routing/blob/master/vtr_flow/arch/xilinx/simple-7series.xml, 2023.
- [25] A. Boutros *et al.*, “Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 35–42.
- [26] M. Eldafrawy *et al.*, “FPGA Logic Block Architectures for Efficient Deep Learning Inference,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 3, pp. 1–34, 2020.
- [27] J. Wang *et al.*, “AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, 2021, pp. 93–104.
- [28] J. Cong *et al.*, “PolySA: Polyhedral-Based Systolic Array Auto-Compilation,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.
- [29] Y. Chen *et al.*, “High-Frequency Systolic Array-Based Transformer Accelerator on Field Programmable Gate Arrays,” *Electronics*, vol. 12, no. 4.
- [30] K. E. Murray *et al.*, “Optimizing FPGA Logic Block Architectures for Arithmetic,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 6, pp. 1378–1391, 2020.
- [31] H. Sharma *et al.*, “Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks,” in *ACM/IEEE 45th International Symposium on Computer Architecture (ISCA)*.
- [32] L. Jia *et al.*, “Generating Systolic Array Accelerators with Reusable Blocks,” *IEEE Micro*, vol. 40, no. 4, pp. 85–92, 2020.
- [33] K. E. Fleming *et al.*, “Leveraging Latency-Insensitivity to Ease Multiple FPGA Design,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2012, pp. 175–184.
- [34] K. E. Murray *et al.*, “Quantifying the Cost and Benefit of Latency Insensitive Communication on FPGAs,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2014.
- [35] E. G. Cota *et al.*, “An Analysis of Accelerator Coupling in Heterogeneous Architectures,” in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.
- [36] L. Josipović *et al.*, “Dynamically Scheduled High-Level Synthesis,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 127–136.
- [37] “Vitis HLS,” <https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html>, 2023.
- [38] AMD-Xilinx, “Accelerating DNNs with Xilinx Alveo Accelerator Cards,” <https://docs.amd.com/v/u/en-US/wp504-accel-dnns>.
- [39] MLIR, Multi-Level IR Compiler Framework, 2020. [Online]. Available: <https://mlir.llvm.org/>
- [40] A. Li *et al.*, “PRGA: An Open-Source FPGA Research and Prototyping Framework,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.
- [41] J. Rose *et al.*, “The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2012, pp. 77–86.
- [42] C. Wolf *et al.*, “Yosys—A Free Verilog Synthesis Suite,” in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.
- [43] V. Betz *et al.*, “VPR: A New Packing, Placement and Routing Tool for FPGA Research,” in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. Berlin, Heidelberg: Springer-Verlag, 1997, pp. 213–222.

- [44] “iCE40 Ultra / Ultra Lite,” <https://www.latticesemi.com/Products/FPGAandCPLD/iCE40Ultra>, 2025, accessed: 2025-08-20.
- [45] J. H. Kim *et al.*, “Synthesizable Standard Cell FPGA Fabrics Targetable by the Verilog-to-Routing CAD Flow,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 2, 2017.
- [46] B. Grady *et al.*, “Physical Design Considerations for Synthesizable Standard-Cell-Based FPGAs,” in *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, NY, USA, 2019.
- [47] P. Mantovani *et al.*, “Agile SoC Development with Open ESP,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.
- [48] G. Tombesi *et al.*, “FLIP2M: Flexible Intra-Layer Parallelism and Inter-Layer Pipelining for Multi-Model AR/VR Workloads,” *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 5s, pp. 1–27, 2025.
- [49] B. Seyoum *et al.*, “PR-ESP: An Open-Source Platform for Design and Programming of Partially Reconfigurable SoCs,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
- [50] L. P. Carloni, “The case for embedded scalable platforms,” in *53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.
- [51] “Intel Agilex Variable Precision DSP Blocks User Guide,” https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/agilex/ug-ag-dsp.pdf, 2022.
- [52] “Intel-Agilex-Like DSP Implementation,” https://github.com/samidhm/DSP_Slice/tree/main/DSP_Slice/Agilex, 2022.
- [53] A. Stillmaker *et al.*, “Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180 nm to 7 nm,” *Integration*, vol. 58.
- [54] M. O. Stan, “HPIPE-NX: Leveraging Tensor Blocks for High-Performance CNN Inference Acceleration on FPGAs,” Master’s thesis, University of Toronto (Canada), 2022.
- [55] A. Arora *et al.*, “Koios: A Deep Learning Benchmark Suite for FPGA Architecture and CAD Research,” in *31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 355–362.
- [56] I. H. Stevenson *et al.*, “How Advances in Neural Recording Affect Data Analysis,” *Nature Neuroscience*, vol. 14, no. 2, pp. 139–142, 2011.
- [57] N. Zeng *et al.*, “A Wireless, Mechanically Flexible, 25 μ m-Thick, 65,536-Channel Subdural Surface Recording and Stimulating Microelectrode Array with Integrated Antennas,” in *2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*.
- [58] S. Wojtowysch *et al.*, “Can Shallow Neural Networks Beat the Curse of Dimensionality? A Mean Field Training Perspective,” *IEEE Transactions on Artificial Intelligence*, vol. 1, no. 2, pp. 121–129, 2021.
- [59] A. E. Urai *et al.*, “Large-Scale Neural Recordings Call for New Insights to Link Brain and Behavior,” *Nature Neuroscience*, vol. 25, no. 1.
- [60] M. S. Lewicki, “A Review of Methods for Spike Sorting: The Detection and Classification of Neural Action Potentials,” *Network: Computation in Neural Systems*, vol. 9, no. 4, p. R53, 1998.
- [61] P. D. Wolf *et al.*, “Thermal Considerations for the Design of an Implanted Cortical Brain-Machine Interface,” *Indwelling Neural Implants: Strategies for Contending with the In Vivo Environment*.
- [62] J. Berezutskaya *et al.*, “Direct Speech Reconstruction from Sensorimotor Brain Activity with Optimized Deep Learning Models,” *Journal of Neural Engineering*, vol. 20, no. 5, p. 056010, 2023.
- [63] G. Eichler *et al.*, “MINDFUL: Safe, Implantable, Large-Scale Brain-Computer Interfaces from a System-Level Design Perspective,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 1672–1689.
- [64] F. Pop, “High Performance Numerical Computing for High Energy Physics: A New Challenge for Big Data Science,” *Advances in High Energy Physics*, vol. 2014, no. 1, p. 507690, 2014.
- [65] J. Gou *et al.*, “Knowledge Distillation: A Survey,” *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.
- [66] G. Cerati, “MicroBooNE Public Data Sets: A Collaborative Tool for LArTPC Software Development,” in *EPJ Web of Conferences*, vol. 295. EDP Sciences, 2024, p. 08012.
- [67] Y. LeCun *et al.*, “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [68] F. Zaruba *et al.*, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.
- [69] P. N. Whatmough *et al.*, “A 16nm 25mm² SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators,” in *Symposium on VLSI Circuits*, 2019.
- [70] B. Khailany *et al.*, “A Modular Digital VLSI Flow for High-Productivity SoC Design,” in *Design Automation Conference (DAC)*, 2018, pp. 1–6.
- [71] J. Balkind *et al.*, “OpenPiton: An Open Source Manycore Research

Framework,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 217–232.

- [72] A. Amid *et al.*, “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs,” *IEEE Micro*.
- [73] F. Ferrandi *et al.*, “Invited: Bambu: An Open-Source Research Framework for the High-Level Synthesis of Complex Applications,” in *58th ACM/IEEE Design Automation Conference (DAC)*, 2021.
- [74] U. Bondhugula *et al.*, “Automatic Mapping of Nested Loops to FPGAs,” in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007, pp. 101–111.
- [75] N. Srivastava *et al.*, “T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations,” in *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 181–189.
- [76] J. Zhuang *et al.*, “ARIES: An Agile MLIR-Based Compilation Flow for Reconfigurable Devices with AI Engines,” in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2025, pp. 92–102.



Biruk Seyoum received his Ph.D. in Computer Science from Scuola Superiore Sant’Anna, Italy, in 2021. He is an Associate Research Scientist in Computer Science at Columbia University.



Allen Boston is pursuing a Ph.D. in computer engineering at the University of Utah, working on FPGA architecture, CAD tools, and AI-assisted chip design.



Carmine Rizzi received his B.S. from Politecnico di Torino, Italy, and M.S. from École Polytechnique, France. He is pursuing a Ph.D. in electrical engineering at ETH Zurich.



Kuan-Lin Chiu received his Ph.D. in computer science from Columbia University. His research focuses on hardware accelerators, data movement, and hardware/software co-design for edge computing.



Guy Eichler received his Ph.D. in computer science from Columbia University, where he is a Postdoctoral Research Scientist. His research focuses on AI hardware acceleration for brain-computer interfaces.



Joseph Zuckerman received his Ph.D. in computer science from Columbia University. He led development of ESP, an open-source SoC design platform.



Lana Josipović received her Ph.D. from EPFL. She is an Assistant Professor at ETH Zurich, working on reconfigurable computing and electronic design automation.



Pierre-Emmanuel Gaillardon is a Professor at the University of Utah, where he leads the Laboratory for NanoIntegrated Systems. His research spans FPGA architecture and emerging device technologies.



Luca P. Carloni is Professor and Chair of Computer Science at Columbia University. He received the Ph.D. from UC Berkeley and is a Fellow of the ACM.